Fast Curvature Matrix-Vector Products for Second-Order Gradient Descent

Technical Report IDSIA-12-00
to appear in Neural Computation

Nicol N. Schraudolph
nic@inf.ethz.ch

Institute of Computational Sciences
Eidgenössische Technische Hochschule
CH-8092 Zürich, Switzerland
http://www.icos.ethz.ch/

November 15, 2000
revised October 30, 2001

Abstract

We propose a generic method for iteratively approximating various second-order gradient steps — Newton, Gauss-Newton, Levenberg-Marquardt, and natural gradient — in *linear* time per iteration, using special curvature matrix-vector products that can be computed in $O(n)$. Two recent acceleration techniques for online learning, *matrix momentum* and *stochastic meta-descent* (SMD), in fact implement this approach. Since both were originally derived by very different routes, this offers fresh insight into their operation, resulting in further improvements to SMD.

1 Introduction

Second-order gradient descent methods typically multiply the local gradient by the inverse of a matrix $\bar{C}$ of local curvature information. Depending on the specific method used, this $n \times n$ matrix (for a system with n parameters) may be the Hessian (Newton's method), an approximation or modification thereof (Gauss-Newton, Levenberg-Marquardt, *etc.*), or the Fisher information (*natural gradient* – Amari, 1998). These methods may converge rapidly, but are computationally quite expensive: the time complexity of common methods to invert $\bar{C}$ is $O(n^3)$, and iterative approximations cost at least $O(n^2)$ per iteration if they compute $\bar{C}^{-1}$ directly, since that is the time required to just access the n^2 elements of this matrix.

Note, however, that second-order gradient methods do *not* require $\bar{C}^{-1}$ explicitly: all they need is its product with the gradient. This is exploited by Yang and Amari (1998) to efficiently compute the natural gradient for multi-layer perceptrons with a single output and one hidden layer: assuming i.i.d. Gaussian input, they explicitly derive the form of the Fisher information matrix and its inverse for their system, and find that the latter's product with the gradient can be computed in just $O(n)$ steps. However, the resulting algorithm is rather complicated, and does not lend itself to being extended to more complex adaptive systems (such as multi-layer perceptrons with more than one output or hidden layer), curvature matrices other than the Fisher information, or inputs that are far from i.i.d. Gaussian.

In order to set up a general framework that admits such extensions (and indeed applies to *any* twice differentiable adaptive system), we abandon the notion of calculating the exact second-order gradient step in favor of an iterative approximation. The following iteration efficiently approaches $\vec{v} = \bar{C}^{-1}\vec{u}$ for an arbitrary vector $\vec{u}$ (Press et al., 1992, page 57):

$$\vec{v}_0 = 0\,; \quad (\forall\, t \geq 0)\;\; \vec{v}_{t+1} = \vec{v}_t + D\,(\vec{u} - \bar{C}\vec{v}_t) \tag{1}$$

where D is a conditioning matrix chosen close to $\bar{C}^{-1}$ if possible. Note that if we restrict D to be diagonal, all operations in (1) can be performed in $O(n)$ time, except (one would suppose) for the matrix-vector product $\bar{C}\vec{v}_t$.

In fact there is an $O(n)$ method for calculating the product of an $n{\times}n$ matrix with an arbitrary vector — if the matrix happens to be the Hessian of a system whose gradient can be calculated in $O(n)$, as is the case for most adaptive architectures encountered in practice. This fast Hessian-vector product (Pearlmutter, 1994; Werbos, 1988; Møller, 1993b) can be used in conjunction with (1) to create an efficient, iterative $O(n)$ implementation of Newton's method.

Unfortunately Newton's method has severe stability problems when used in nonlinear systems, stemming from the fact that the Hessian may be ill-conditioned and does not guarantee positive definiteness. Practical second-order methods therefore prefer measures of curvature that are better behaved, such as the outer product (Gauss-Newton) approximation of the Hessian, a model-trust region modification of the same (Levenberg, 1944; Marquardt, 1963), or the Fisher information.

Below we define these matrices in a maximum likelihood framework for regression and classification, and describe $O(n)$ algorithms for computing the product of any of them with an arbitrary vector for neural network architectures. These curvature matrix-vector products are, in fact, cheaper still than the fast Hessian-vector product, and can be used in conjunction with (1) to implement rapid, iterative, optionally stochastic $O(n)$ variants of second-order gradient descent methods. The resulting algorithms are very general, practical (*i.e.*, sufficiently robust and efficient), far less expensive than the conventional $O(n^2)$ and $O(n^3)$ approaches, and — with the aid of *automatic differentiation* software tools — comparatively easy to implement (see Section 4).

We then examine two learning algorithms that use this approach: *matrix momentum* (Orr, 1995; Orr and Leen, 1997) and *stochastic meta-descent* (Schrau-

dolph, 1999b,c; Schraudolph and Giannakopoulos, 2000). Since both methods were derived by entirely different routes, viewing them as implementations of iteration (1) will provide additional insight into their operation, and suggest new ways to improve them.

2 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 $\vec{x}$ to the corresponding output $\vec{y}$, for a given vector $\vec{w}$ 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 $\vec{y} = \mathcal{N}(\vec{w})$. 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 $\vec{z} = \mathcal{M}(\vec{y}) = \mathcal{M}(\mathcal{N}(\vec{w}))$. For networks with linear outputs, $\mathcal{M}$ will be the identity map.

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 $\mathcal{L}(\vec{z}) \equiv -\log \Pr(\vec{z})$ of the output $\vec{z}$ under a suitable statistical model (Bishop, 1995). For supervised learning, $\mathcal{L}$ may also implicitly depend on given targets $\vec{z}^*$ for the outputs. Formally, the loss can now be regarded as a function $\mathcal{L}(\mathcal{M}(\mathcal{N}(\vec{w})))$ of the weights, for a given set of inputs and (if supervised) targets.

Jacobian and gradient. 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 — the vector $\vec{g}$ of derivatives of the loss with respect to the weights — is given by

$$\vec{g} \;\equiv\; \frac{\partial}{\partial \vec{w}} \mathcal{L}(\mathcal{M}(\mathcal{N}(\vec{w}))) \;=\; J'_{\mathcal{L}\circ\mathcal{M}\circ\mathcal{N}} \;=\; J'_{\mathcal{N}} J'_{\mathcal{M}} J'_{\mathcal{L}}\,, \tag{2}$$

where $\circ$ denotes function composition, and $'$ the matrix transpose.

Matching loss functions. We say that the loss function $\mathcal{L}$ *matches* the output nonlinearity $\mathcal{M}$ iff $J'_{\mathcal{L}\circ\mathcal{M}} = A\vec{z} + \vec{b}$, for some A and $\vec{b}$ not dependent on $\vec{w}$.[1] 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 $\vec{b} = -\vec{z}^*$, so that $J'_{\mathcal{L}\circ\mathcal{M}} = \vec{z} - \vec{z}^*$ (Bishop, 1995, chapter 6). This will simplify some of the calculations described in Section 4 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 $\mathcal{F}(\vec{w})$ with respect to its inputs $\vec{w}$:

$$H_{\mathcal{F}} \;\equiv\; \frac{\partial J_{\mathcal{F}}}{\partial \vec{w}'}\,, \quad \textit{i.e.,} \quad (H_{\mathcal{F}})_{ij} \;=\; \frac{\partial^2 \mathcal{F}(\vec{w})}{\partial w_i \, \partial w_j}\,. \tag{3}$$

[1] For supervised learning, a very similar if somewhat more restrictive definition of matching loss functions is given by Helmbold et al. (1996); Auer et al. (1996).

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: $\bar{H} \equiv \langle H \rangle_{\vec{x}}$. For matching loss functions, $H_{\mathcal{L} \circ \mathcal{M}} = A J_{\mathcal{M}} = J'_{\mathcal{M}} A'$.

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:

$$F_{\mathcal{F}} \;\equiv\; J'_{\mathcal{F}} J_{\mathcal{F}}\,, \quad \textit{i.e.,} \quad (F_{\mathcal{F}})_{ij} \;=\; \frac{\partial \mathcal{F}(\vec{w})}{\partial w_i} \, \frac{\partial \mathcal{F}(\vec{w})}{\partial w_j}\,. \tag{4}$$

Note that $F_{\mathcal{F}}$ always has rank one. Again we abbreviate $F \equiv F_{\mathcal{L} \circ \mathcal{M} \circ \mathcal{N}} = \vec{g}\vec{g}'$. The Fisher information matrix proper, $\bar{F} \equiv \langle F \rangle_{\vec{x}}$, describes the geometric structure of weight space (Amari, 1985) and is used in the *natural gradient* descent approach (Amari, 1998).

3 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 $\bar{F}$ (Amari, 1998), which — being a quadratic form — is positive semi-definite by definition. On the other hand, $\bar{F}$ ignores *all* second-order interactions between system parameters, thus throwing away potentially useful curvature information. By contrast, we shall derive an approximation of the Hessian that is provably positive semi-definite even though it does make use of second derivatives to better model Hessian curvature.

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 model can be written as

$$H \;=\; \frac{\partial}{\partial \vec{w}'} (J_{\mathcal{L} \circ \mathcal{M}} \, J_{\mathcal{N}}) \;=\; J'_{\mathcal{N}} \, H_{\mathcal{L} \circ \mathcal{M}} \, J_{\mathcal{N}} \;+\; \sum_{i=1}^{o} (J_{\mathcal{L} \circ \mathcal{M}})_i \, H_{\mathcal{N}_i}\,, \tag{5}$$

where i ranges over the o outputs of $\mathcal{N}$, 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

$$G \equiv J'_{\mathcal{N}} H_{\mathcal{L} \circ \mathcal{M}} J_{\mathcal{N}} . \tag{6}$$

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

G models the second-order interactions among $\mathcal{N}$'s outputs (via $H_{\mathcal{L} \circ \mathcal{M}}$) while ignoring those arising within $\mathcal{N}$ 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; for multiple outputs it provides a richer ($\text{rank}(G) \leq o$ *vs.* $\text{rank}(F) = 1$) model of Hessian curvature.

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

Firstly, the residual $J'_{\mathcal{L} \circ \mathcal{M}} = \vec{z} - \vec{z}^*$ vanishes at the optimum for realizable problems, so that the Gauss-Newton approximation (6) of the Hessian (5) 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 (5) 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 $\text{diag}[\text{diag}(\vec{z})(1 - \vec{z})]$, positive semi-definite because $(\forall i)\ 0 < z_i < 1$. For softmax output with cross-entropy loss we have $H_{\mathcal{L} \circ \mathcal{M}} = \text{diag}(\vec{z}) - \vec{z}\vec{z}'$, which is also positive semi-definite since $(\forall i)\ z_i > 0$ and $\sum_i z_i = 1$, and thus

$$\begin{aligned} (\forall \vec{v} \in \mathbb{R}^o) \quad \vec{v}'[\text{diag}(\vec{z}) - \vec{z}\vec{z}']\,\vec{v} \;&= \sum_i z_i v_i^2 \;-\; (\sum_i z_i v_i)^2 \\ &= \sum_i z_i v_i^2 \;-\; 2\,(\sum_i z_i v_i)(\sum_j z_j v_j) \;+\; (\sum_j z_j v_j)^2 \\ &= \sum_i z_i (v_i - \sum_j z_j v_j)^2 \;\geq\; 0\,. \end{aligned} \tag{7}$$

Model-trust region. As long as G is positive semi-definite — as we have proven above for standard loss functions — the extended Gauss-Newton algorithm will not take steps in uphill directions. However, it may still take very large (even infinite) steps. These may take us outside the *model-trust region*, the area in which our quadratic model of the error surface is reasonable. Model-trust region methods restrict the gradient step to a suitable neighborhood around the current point.

One popular way to enforce a model-trust region is the addition of a small, diagonal term to the curvature matrix. Levenberg (1944) has suggested adding

λI to the Gauss-Newton matrix $\bar{G}$; Marquardt (1963) has elaborated the additive term to $\lambda \operatorname{diag}(\bar{G})$. The Levenberg-Marquardt algorithm directly inverts the resulting curvature matrix; where affordable (*i.e.*, for relatively small systems) it has become today's workhorse of nonlinear least-squares optimization.

4 Fast Curvature Matrix-Vector Products

We now describe algorithms that compute the product of F, G, or H with an arbitrary n-dimensional vector $\vec{v}$ in $O(n)$. They can be used in conjunction with (1) to implement rapid and (if so desired) stochastic versions of various second-order gradient descent methods, including Newton's method, Gauss-Newton, Levenberg-Marquardt, and natural gradient descent.

4.1 The Passes

The fast matrix-vector products are all constructed from the same set of *passes* in which certain quantities are propagated through all or part of our neural network model (comprising $\mathcal{N}$, $\mathcal{M}$, and $\mathcal{L}$) in either forward or reverse direction. For implementation purposes it should be noted that *automatic differentiation* software tools[2] 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 $\mathcal{F}(\vec{w})$ it implements by propagating activity (*i.e.,* intermediate results) forward through $\mathcal{F}$.

$\boldsymbol{r_1}$. The ordinary backward pass of a neural network, calculating $J'_{\mathcal{F}}\vec{u}$ by propagating the vector $\vec{u}$ backwards through $\mathcal{F}$. This pass uses intermediate results computed in the f_0 pass.

$\boldsymbol{f_1}$. Following Pearlmutter (1994), we define the *Gateaux* derivative

$$\mathcal{R}_{\vec{v}}(\mathcal{F}(\vec{w})) \equiv \left.\frac{\partial \mathcal{F}(\vec{w}+r\vec{v})}{\partial r}\right|_{r=0} = J_{\mathcal{F}}\,\vec{v} \tag{8}$$

which describes the effect on a function $\mathcal{F}(\vec{w})$ of a weight perturbation in the direction of $\vec{v}$. By pushing $\mathcal{R}_{\vec{v}}$ — 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 $J_{\mathcal{F}}\vec{v}$ from $\vec{v}$; see Pearlmutter (1994) for details and examples. This f_1 pass uses intermediate results from the f_0 pass.

$\boldsymbol{r_2}$. When the $\mathcal{R}_{\vec{v}}$ 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 $H_{\mathcal{F}}\,\vec{v} = \mathcal{R}_{\vec{v}}(J'_{\mathcal{F}})$. Again, see Pearlmutter (1994) for details and examples. This r_2 pass uses intermediate results from the f_0, f_1, and r_1 passes.

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

4.2 The Algorithms

The first step in all three matrix-vector products is the computation of the gradient $\vec{g}$ of our neural network model by standard backpropagation:

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

Fisher information. To compute $F\vec{v} = \vec{g}\vec{g}'\vec{v}$, simply multiply the gradient $\vec{g}$ by the inner product between $\vec{g}$ and $\vec{v}$. If there is no random access to $\vec{g}$ or $\vec{v}$ — *i.e.,* its elements can be accessed only through passes like the above — the scalar $\vec{g}'\vec{v}$ can instead be calculated by f_1-propagating $\vec{v}$ forward through the model ($\mathcal{N}$, $\mathcal{M}$, and $\mathcal{L}$). This step is also necessary for the other two matrix-vector products.

Hessian. After f_1-propagating $\vec{v}$ forward through $\mathcal{N}$, $\mathcal{M}$, and $\mathcal{L}$, r_2-propagate $\mathcal{R}_{\vec{v}}(1) = 0$ back through the entire model ($\mathcal{L}$, $\mathcal{M}$, then $\mathcal{N}$) to obtain $H\vec{v} = \mathcal{R}_{\vec{v}}(\vec{g})$ (Pearlmutter, 1994). For matching loss functions, the shortcut is to f_1-propagate $\vec{v}$ through just $\mathcal{N}$ and $\mathcal{M}$ to obtain $\mathcal{R}_{\vec{v}}(\vec{z})$, then r_2-propagate $\mathcal{R}_{\vec{v}}(J'_{\mathcal{L}\circ\mathcal{M}}) = A\,\mathcal{R}_{\vec{v}}(\vec{z})$ back through just $\mathcal{N}$.

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

$$G \;=\; J'_{\mathcal{N}}H_{\mathcal{L}\circ\mathcal{M}}J_{\mathcal{N}} \;=\; J'_{\mathcal{N}}J'_{\mathcal{M}}A'J_{\mathcal{N}}\,, \tag{9}$$

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 $G\vec{v} = J'_{\mathcal{N}}AJ_{\mathcal{M}}J_{\mathcal{N}}\vec{v}$ by continuing the f_1 pass through $\mathcal{M}$, multiplying with A, then r_1-propagating back through $\mathcal{N}$ only.

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

4.3 Computational Cost

Table 1 summarizes, for a number of gradient descent methods, their choice of curvature matrix C, the passes needed (for a matching loss function) to calculate both the gradient $\vec{g}$ and the fast matrix-vector product $\bar{C}\vec{v}$, 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.

Computing both gradient and curvature matrix-vector product is typically about 2–3 times as expensive as calculating the gradient alone. In combination with iteration (1), however, one can use the $O(n)$ matrix-vector product to implement second-order gradient methods whose rapid convergence more than compensates for the additional cost. We shall describe two such algorithms in the following section.

5 Rapid Second-Order Gradient Descent

We know of two neural network learning algorithms that combine the $O(n)$ curvature matrix-vector product with iteration (1) in some form: *matrix momentum* (Orr, 1995; Orr and Leen, 1997), and our own *stochastic meta-descent* (Schraudolph, 1999b,c; Schraudolph and Giannakopoulos, 2000). Since both of these were derived by entirely different routes, we gain fresh insight into their operation by examining how they implement (1).

5.1 Stochastic Meta-Descent

Algorithm. *Stochastic meta-descent* (SMD – Schraudolph, 1999b,c) is a new online algorithm for local learning rate adaptation. It updates the weights $\vec{w}$ by the simple gradient descent

$$\vec{w}_{t+1} \quad = \quad \vec{w}_t \; - \; \mathrm{diag}(\vec{p}_t)\, \vec{g}\,. \tag{10}$$

The vector $\vec{p}$ of local learning rates is adapted multiplicatively:

$$\vec{p}_t \quad = \quad \mathrm{diag}(\vec{p}_{t-1})\, \max(\tfrac{1}{2}\,,\, 1 \,+\, \mu\, \mathrm{diag}(\vec{v}_t)\, \vec{g}\,) \tag{11}$$

using a scalar meta-learning rate μ. Finally, the auxiliary vector $\vec{v}$ used in (11) is itself updated iteratively via

$$\vec{v}_{t+1} \quad = \quad \lambda\, \vec{v}_t \;+\; \mathrm{diag}(\vec{p}_t)\, (\,\vec{g} \,-\, \lambda\, C \vec{v}_t)\,, \tag{12}$$

		Pass	f_0	r_1	f_1	r_2	**Cost**
Method		result:	$\mathcal{F}$	$J'_{\mathcal{F}}\,\vec{u}$	$J_{\mathcal{F}}\,\vec{v}$	$H_{\mathcal{F}}\,\vec{v}$	(for $\vec{g}$
$C =$	name	cost:	2	3	4	7	& $\bar{C}\vec{v}$)
I	simple gradient		$\surd$	$\surd$			6
F	natural gradient		$\surd$	$\surd$	$(\surd)$		10
G	Gauss-Newton		$\surd$	$\surd\!\surd$	$\surd$		14
H	Newton's method		$\surd$	$\surd$	$\surd$	$\surd$	18

Table 1: Choice of curvature matrix C for various gradient descent methods, passes needed to compute gradient $\vec{g}$ and fast matrix-vector product $\bar{C}\vec{v}$, and associated cost (for a multi-layer perceptron) in flops per weight and pattern.

where $0 \leq \lambda \leq 1$ is a forgetting factor for nonstationary tasks. Although derived as part of a dual gradient descent procedure (minimizing loss with respect to both $\vec{w}$ and $\vec{p}$), (12) implements an interesting variation of (1). SMD thus employs rapid second-order techniques indirectly, to help adapt local learning rates for the gradient descent in weight space.

Linearization. The learning rate update (11) minimizes the system's loss with respect to $\vec{p}$ by *exponentiated* gradient descent (Kivinen and Warmuth, 1995), but has been re-linearized in order to avoid the computationally expensive exponentiation operation (Schraudolph, 1999a). The particular linearization used, $e^u \approx \max(\varrho, 1+u)$, is based on a first-order Taylor expansion about $u = 0$, bounded below by $0 < \varrho < 1$ so as to safeguard against unreasonably small (and worse: negative) multipliers for $\vec{p}$. The value of ϱ determines the maximum permissible learning rate reduction; we follow many other step size control methods in setting this to $\varrho = \frac{1}{2}$, the ratio between optimal and maximum stable step size in a symmetric bowl. Compared to direct exponentiated gradient descent, our linearized version (11) thus dampens radical changes (in both directions) to $\vec{p}$ that may occasionally arise due to the stochastic nature of the data.

Diagonal, adaptive conditioner. For $\lambda = 1$, SMD's update (12) of $\vec{v}$ implements (1) with the diagonal conditioner $D = \mathrm{diag}(\vec{p})$. Note that the learning rates $\vec{p}$ are being adapted so as to make the gradient step $\mathrm{diag}(\vec{p})\vec{g}$ as effective as possible. A well-adapted $\vec{p}$ will typically make this step similar to the second-order gradient $\bar{C}^{-1}\vec{g}$. In this restricted sense, we can regard $\mathrm{diag}(\vec{p})$ as an empirical diagonal approximation of $\bar{C}^{-1}$ — making it a good choice for the conditioner D in iteration (1).

Initial learning rates. Although SMD is very effective at adapting local learning rates to changing requirements, it is nonetheless sensitive to their initial values. All three of its update rules rely on $\vec{p}$ for their conditioning, so initial values that are very far from optimal are bound to cause problems: divergence if they are too high, lack of progress is they are too low. A simple architecture-dependent technique such as *tempering* (Schraudolph and Sejnowski, 1996) should usually suffice to adequately initialize $\vec{p}$; the fine tuning can be left to the SMD algorithm.

Model-trust region. For $\lambda < 1$, the stochastic fixpoint of (12) is no longer $\vec{v} \to C^{-1}\vec{g}$, but rather

$$\vec{v} \;\to\; \left[\lambda C \;+\; (1-\lambda)\,\mathrm{diag}(\vec{p})^{-1}\right]^{-1}\vec{g}\,. \tag{13}$$

This clearly implements a model-trust region approach, in that a diagonal matrix is being added (in small proportion) to C before inverting it. Moreover, the elements along the diagonal are not all identical as in the Levenberg (1944) method, but scale individually as suggested by Marquardt (1963). The scaling factors are determined by $1/\vec{p}$, rather than $\mathrm{diag}(\bar{C})$ as the Levenberg-Marquardt method would have it, but these two vectors are related by our above argument that $\vec{p}$ is a diagonal approximation of $\bar{C}^{-1}$. For $\lambda < 1$, SMD's iteration

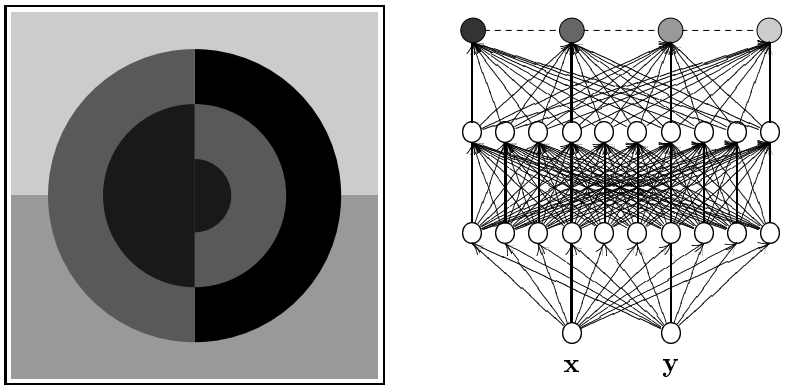

Figure 1: The four regions task (left), and the network we trained on it (right).

(12) can thus be regarded as implementing an efficient stochastic variant of the Levenberg-Marquardt model-trust region approach.

Benchmark setup. We will illustrate the behavior of SMD with empirical data obtained on the "four regions" benchmark (Singhal and Wu, 1989): a fully connected feedforward network $\mathcal{N}$ with two hidden layers of 10 units each (Figure 1, right) is to classify two continuous inputs in the range [-1,1] into four disjoint, non-convex regions (Figure 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 $\vec{p}_0 = 0.1$, and a hyperbolic tangent nonlinearity on the hidden units. For each run the 184 weights (including bias weights for all units) are initialized to uniformly random values in the range [-0.3,0.3]. Training patterns are generated online by drawing independent, uniformly random input samples; since each pattern is seen only once, the empirical loss provides an unbiased estimate of generalization ability. Patterns are presented in mini-batches of ten each so as to reduce the computational overhead associated with SMD's parameter updates (10), (11), and (12).[3]

Curvature matrix. Figure 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 (12). 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.

[3]In exploratory experiments, comparative results when training fully online (*i.e.*, pattern by pattern) were noisier but not substantially different.

(For comparison, we cannot get matrix momentum to converge *at all* on anything as difficult as this benchmark.)

Stability. In contrast to matrix momentum, the high stochasticity of $\vec{v}$ affects the weights in SMD only indirectly, being buffered — and largely averaged out — by the incremental update (11) of learning rates $\vec{p}$. This makes SMD far more stable, especially when G is used as the curvature matrix. Its residual tendency to occasionally misbehave can be suppressed further by slightly lowering λ so as to create a model-trust region. By curtailing the memory of iteration (12), however, this approach can compromise the rapid convergence of SMD. Figure 3 illustrates the resulting stability/performance tradeoff on the four regions benchmark:

When using 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, plotted up to the earliest point of divergence). 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 (Harmon and Baird, 1996), the to our knowledge best online method for local learning rate adaptation preceding SMD.

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 (12) under control, aiming to ensure the stability of SMD while maintaining its excellent performance near $\lambda = 1$.

5.2 Matrix Momentum

Algorithm. The investigation of asymptotically optimal adaptive momentum for first-order stochastic gradient descent (Leen and Orr, 1994) led Orr (1995)

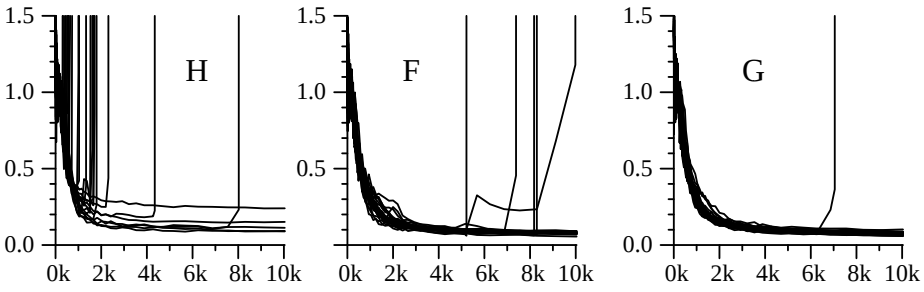

Figure 2: Loss curves for 25 runs of SMD with $\lambda = 1$, when using the Hessian (left), the Fisher information (center), or the extended Gauss-Newton matrix (right) for C in Equation (12). Vertical spikes indicate divergence.

to propose the following *matrix momentum* weight update:

$$\vec{w}_{t+1} \;=\; \vec{w}_t \;+\; \vec{v}_{t+1}\,, \qquad \vec{v}_{t+1} \;=\; \vec{v}_t \;-\; \mu\,(\varrho_t \vec{g} \;+\; C\vec{v}_t)\,, \tag{14}$$

where μ is a scalar constant less than the inverse of $\bar{C}$'s largest eigenvalue, and ϱ_t a rate parameter that is annealed from one to zero. We recognize (1) with scalar conditioner $D = \mu$ and stochastic fixpoint $\vec{v} \to -\varrho_t\, C^{-1}\vec{g}$; thus matrix momentum attempts to directly approximate partial second-order gradient steps via this fast, stochastic iteration.

Rapid convergence. Orr (1995) found that in the late, *annealing* phase of learning, matrix momentum converges at optimal (second-order) asymptotic rates; this has been confirmed by subsequent analysis in a statistical mechanics framework (Rattray and Saad, 1999; Scarpetta et al., 1999). Moreover, compared to SMD's slow, incremental adaptation of learning rates, matrix momentum's direct second-order update of the weights promises a far shorter initial transient before rapid convergence sets in. Matrix momentum thus looks like the ideal candidate for a fast $O(n)$ stochastic gradient descent method.

Instability. Unfortunately matrix momentum has a strong tendency to diverge for nonlinear systems when far from an optimum, as is the case during the *search* phase of learning. Current implementations therefore rely on simple

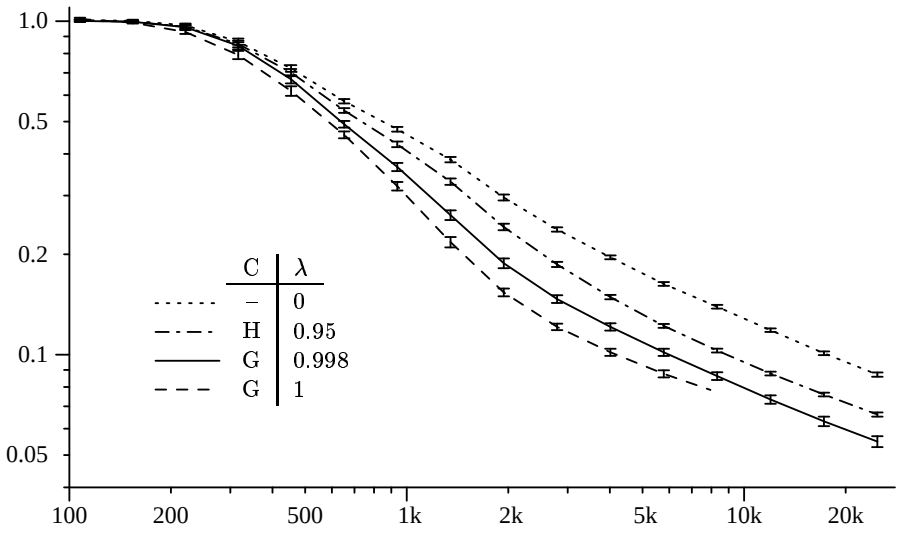

Figure 3: Average loss over 25 runs of SMD for various combinations of curvature matrix C and forgetting factor λ. Memory ($\lambda \to 1$) accelerates convergence over the conventional, memory-less case $\lambda = 0$ (dotted) but can lead to instability. With the Hessian H all 25 runs remain stable up to $\lambda = 0.95$ (dot-dashed); using the extended Gauss-Newton matrix G pushes this limit up to $\lambda = 0.998$ (solid). The curve for $\lambda = 1$ (dashed) is plotted up to the earliest point of divergence.

(first-order) stochastic gradient descent initially, turning on matrix momentum only once the vicinity of an optimum has been reached (Orr, 1995; Orr and Leen, 1997). The instability of matrix momentum is not caused by lack of semi-definiteness on behalf of the curvature matrix: Orr (1995) used the Gauss-Newton approximation, and Scarpetta et al. (1999) reached similar conclusions for the Fisher information matrix. Instead it is thought to be a consequence of the noise inherent in the stochastic approximation of the curvature matrix (Rattray and Saad, 1999; Scarpetta et al., 1999).

Recognizing matrix momentum as implementing the same iteration (1) as SMD suggests that its stability might be improved in similar ways — specifically, by incorporating a model-trust region parameter λ and an adaptive diagonal conditioner. However, whereas in SMD such a conditioner was trivially available in the vector $\vec{p}$ of local learning rates, here it is by no means easy to construct, given our restriction to $O(n)$ algorithms which are affordable for very large systems. We are at present investigating several routes towards a stable, adaptively conditioned form of matrix momentum.

6 Summary

We have extended the notion of Gauss-Newton approximation of the Hessian from nonlinear least squares problems to arbitrary loss functions, and shown that it is positive semi-definite for the standard loss functions used in neural network regression and classification. We have given algorithms that compute the product of either the Fisher information or our extended Gauss-Newton matrix with an arbitrary vector in $O(n)$, similar to but even cheaper than the fast Hessian-vector product described by Pearlmutter (1994).

We have shown how these fast matrix-vector products may be used to construct $O(n)$ iterative approximations to a variety of common second-order gradient algorithms, including the Newton, natural gradient, Gauss-Newton, and Levenberg-Marquardt steps. Applying these insights to our recent *stochastic meta-descent* (SMD) algorithm (Schraudolph, 1999b) — specifically, replacing the Hessian with our extended Gauss-Newton approximation — resulted in improved stability and performance. We are now investigating whether *matrix momentum* (Orr, 1995) can similarly be stabilized though the incorporation of adaptive diagonal conditioner and model-trust region parameter.

Acknowledgments

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

S. Amari. *Differential-Geometrical Methods in Statistics*, volume 28 of *Lecture Notes in Statistics*. Springer Verlag, New York, 1985.

S. Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10(2):251–276, 1998.

P. Auer, M. Herbster, and M. K. Warmuth. Exponentially many local minima for single neurons. In Touretzky et al. (1996), pages 316–322.

C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

M. E. Harmon and L. C. Baird, III. Multi-player residual advantage learning with general function approximation. Technical Report WL-TR-1065, Wright Laboratory, WL/AACF, Wright-Patterson Air Force Base, OH 45433-7308, 1996. `http://www.leemon.com/papers/sim_tech/sim_tech.pdf`.

D. P. Helmbold, J. Kivinen, and M. K. Warmuth. Worst-case loss bounds for single neurons. In Touretzky et al. (1996), pages 309–315.

J. Kivinen and M. K. Warmuth. Additive versus exponentiated gradient updates for linear prediction. In *Proc. 27th Annual ACM Symposium on Theory of Computing*, pages 209–218, New York, NY, May 1995. The Association for Computing Machinery.

T. K. Leen and G. B. Orr. Optimal stochastic search and adaptive momentum. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 477–484. Morgan Kaufmann, San Francisco, CA, 1994.

K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly Journal of Applied Mathematics*, II(2):164–168, 1944.

D. Marquardt. An algorithm for least-squares estimation of non-linear parameters. *Journal of the Society of Industrial and Applied Mathematics*, 11(2): 431–441, 1963.

M. F. Møller. *Efficient Training of Feed-Forward Neural Networks*. PhD thesis, Computer Science Department, Århus University, Århus, Denmark, 1993a. `http://www.daimi.au.dk/PB/464/PB464.ps.gz`.

M. F. Møller. Exact calculation of the product of the Hessian matrix of feed-forward network error functions and a vector in $O(n)$ time. Technical Report DAIMI PB-432, Computer Science Department, Århus University, 1993b. `http://www.daimi.au.dk/PB/432/PB432.ps.gz`. Part of (Møller, 1993a).

G. B. Orr. *Dynamics and Algorithms for Stochastic Learning.* PhD thesis, Department of Computer Science and Engineering, Oregon Graduate Institute, Beaverton, OR 97006, 1995. `ftp://neural.cse.ogi.edu/pub/neural/papers/orrPhDch1-5.ps.Z`, `orrPhDch6-9.ps.Z`.

G. B. Orr and T. K. Leen. Using curvature information for fast stochastic search. In M. C. Mozer, M. I. Jordan, and T. Petsche, editors, *Advances in Neural Information Processing Systems*, volume 9. The MIT Press, Cambridge, MA, 1997.

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

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, second edition, 1992.

M. Rattray and D. Saad. Incorporating curvature information into on-line learning. In D. Saad, editor, *On-Line Learning in Neural Networks*, Publications of the Newton Institute, chapter 9, pages 183–207. Cambridge University Press, 1999.

S. Scarpetta, M. Rattray, and D. Saad. Matrix momentum for practical natural gradient learning. *Journal of Physics A*, 32:4047–4059, 1999.

N. N. Schraudolph. A fast, compact approximation of the exponential function. *Neural Computation*, 11(4):853–862, 1999a. `http://www.inf.ethz.ch/~schraudo/pubs/exp.ps.gz`.

N. N. Schraudolph. Local gain adaptation in stochastic gradient descent. In *Proceedings of the 9th International Conference on Artificial Neural Networks*, pages 569–574, Edinburgh, Scotland, 1999b. IEE, London. `http://www.inf.ethz.ch/~schraudo/pubs/smd.ps.gz`.

N. N. Schraudolph. Online learning with adaptive local step sizes. In M. Marinaro and R. Tagliaferri, editors, *Neural Nets — WIRN Vietri-99: Proceedings of the 11th Italian Workshop on Neural Nets*, Perspectives in Neural Computing, pages 151–156, Vietri sul Mare, Salerno, Italy, 1999c. Springer Verlag, Berlin.

N. N. Schraudolph and X. Giannakopoulos. Online independent component analysis with local learning rate adaptation. In S. A. Solla, T. K. Leen, and K.-R. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12, pages 789–795. The MIT Press, Cambridge, MA, 2000. `http://www.inf.ethz.ch/~schraudo/pubs/smdica.ps.gz`.

N. N. Schraudolph and T. J. Sejnowski. Tempering backpropagation networks: Not all weights are created equal. In Touretzky et al. (1996), pages 563–569. `http://www.inf.ethz.ch/~schraudo/pubs/nips95.ps.gz`.

S. Singhal and L. Wu. Training multilayer perceptrons with the extended Kalman filter. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems. Proceedings of the 1988 Conference*, pages 133–140, San Mateo, CA, 1989. Morgan Kaufmann.

D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors. *Advances in Neural Information Processing Systems*, volume 8, 1996. The MIT Press, Cambridge, MA.

P. J. Werbos. Backpropagation: past and future. In *Proceedings of the IEEE International Conference on Neural Networks, San Diego, 1988*, volume I, pages 343–353, Long Beach, CA, 1988. IEEE Press.

H. H. Yang and S. Amari. Complexity issues in natural gradient descent method for training multilayer perceptrons. *Neural Computation*, 10(8):2137–2157, 1998.