Krylov Subspace Descent for Deep Learning

Oriol Vinyals
University of California, Berkeley

Daniel Povey
Microsoft Research

Abstract

In this paper, we propose a second order optimization method to learn models where both the dimensionality of the parameter space and the number of training samples is high. In our method, we construct on each iteration a Krylov subspace formed by the gradient and an approximation to the Hessian matrix, and then use a subset of the training data samples to optimize over this subspace. As with the Hessian Free (HF) method of Martens (2010), the Hessian matrix is never explicitly constructed, and is computed using a subset of data. In practice, as in HF, we typically use a positive definite substitute for the Hessian matrix such as the Gauss-Newton matrix. We investigate the effectiveness of our proposed method on deep neural networks, and compare its performance to widely used methods such as stochastic gradient descent, conjugate gradient descent and L-BFGS, and also to HF. Our method leads to faster convergence than either L-BFGS or HF, and generally performs better than either of them in cross-validation accuracy. It is also simpler and more general than HF, as it does not require a positive semidefinite approximation of the Hessian matrix to work well nor the setting of a damping parameter. The chief drawback versus HF is the need for memory to store a basis for the Krylov subspace.

1 Introduction

Many algorithms in machine learning and other scientific computing fields rely on optimizing a function with respect to a parameter space. In many cases, the objective function being optimized takes the form of a sum over a large number of terms that can be treated as identically distributed: for instance, labeled training samples. In deep learning, the problem that we are trying to solve often consists of minimizing the negated log-likelihood:

$$f(\boldsymbol{\theta}) = -\log(p(\mathbf{Y}|\mathbf{X};\boldsymbol{\theta})) = -\sum_{i=1}^{N} \log(p(\mathbf{y}_i|\mathbf{x}_i;\boldsymbol{\theta})) \quad (1)$$

where $(\mathbf{X}, \mathbf{Y})$ are our observations and labels respectively, and p is the posterior probability of our labels which is modeled by a deep neural network with parameters $\boldsymbol{\theta}$. In this case it is possible to use subsets of the training data to obtain noisy estimates of quantities such as gradients; the canonical example of this is Stochastic Gradient Descent (SGD).

The simplest reference point to start from when explaining our method is Newton's method with line search, where on iteration m we do an update of the form:

$$\theta_{m+1} = \theta_m - \alpha \mathbf{H}_m^{-1} \mathbf{g}_m, \quad (2)$$

where $\mathbf{H}_m$ and $\mathbf{g}_m$ are, respectively, the Hessian and the gradient on iteration m of the objective function (1); here, α would be chosen to minimize (1) at θ_{m+1}. For high dimensional problems it is not practical to invert the Hessian; however, we can efficiently approximate (2) using only multiplication by $\mathbf{H}_m$, by using the Conjugate Gradients (CG) method with a truncated number of iterations. In addition, it is possible to multiply by $\mathbf{H}_m$ without explicitly forming it, using what is known as the "Pearlmutter trick" (Pearlmutter, 1994) for multiplying an arbitrary vector by the Hessian; this is described for neural networks but is applicable to quite general types of functions[1]. This type of optimization method is known as "truncated Newton" or "Hessian-free inexact Newton" (Morales

Appearing in Proceedings of the 15th International Conference on Artificial Intelligence and Statistics (AISTATS) 2012, La Palma, Canary Islands. Volume XX of JMLR: W&CP XX. Copyright 2012 by the authors.

[1] This was actually known to the optimization community prior to (Pearlmutter, 1994); see Nocedal and Wright (2006, Chapter 8).

and Nocedal, 2000). In Byrd et al. (2010), this method is applied but using only a subset of data to approximate the Hessian $\mathbf{H}_m$. A more sophisticated version of the same idea was described in the earlier paper (Martens, 2010), in which preconditioning is applied, the Hessian is damped with the unit matrix in a Levenberg-Marquardt fashion, and the method is extended to non-convex problems by using the Gauss-Newton matrix (described below) as a substitute the Hessian. These changes made it possible to use HF to effectively train deep networks from random initializations, which would not have been possible with any previously described versions of HF.

Our method is quite similar to the one described in Martens (2010), which we will refer to as Hessian Free (HF). We also multiply by the Hessian (or Gauss-Newton matrix) using the Pearlmutter trick on a subset of data, but on each iteration, instead of approximately computing $(\mathbf{H}_m + \lambda\mathbf{I})^{-1}\mathbf{g}_m$ using truncated CG, we compute a basis for the Krylov subspace spanned by $\mathbf{g}_m, \mathbf{H}_m\mathbf{g}_m, \ldots \mathbf{H}_m^{K-1}\mathbf{g}_m$ for some K fixed in advance (e.g. $K = 20$), and numerically optimize the parameter change within this subspace, using BFGS to minimize the original nonlinear objective function measured on a subset of the training data. It is easy to show that, for any λ, the approximate solution to $\mathbf{H}_m + \lambda\mathbf{I}$ found by K iterations of CG will lie in this subspace, so we are in effect automatically choosing the optimal λ in the Levenburg-Marquardt smoothing method of HF – although our algorithm is free to choose a solution more general than this. This is clear from the CG algorithm itself, and from the fact that the order-K Krylov subspaces generated by $\mathbf{g}$ and $\mathbf{H}+\lambda\mathbf{I}$ are all the same irrespective of λ. We note that both our method and HF use preconditioning, which we have glossed over in the discussion above. Compared with HF, the advantages of our method are:

- Greater simplicity and robustness: there is no need for heuristics to initialize and update the smoothing value λ.
- Generality: unlike HF, our method can be applied even if $\mathbf{H}$ (or whatever approximation or substitute we use) is not positive semidefinite.
- Empirical advantages: our method generally seems to work better than HF in both optimization speed and classification performance.

The chief disadvantages versus HF are:

- Memory requirement: we require storage of K times the parameter dimension to store the subspace (HF does not require memory proportional to the number of CG iterations).
- Convergence properties: the use of a subset of data to optimize over the subspace will prevent convergence to an optimum.

Regarding the convergence properties: for deep neural networks, we view this as more of a theoretical than a practical problem, since for typical setups in training deep networks the residual parameter noise due to the use of data subsets would be far less than that due to overtraining. We hope that not-too-restrictive conditions could be found under which our algorithm (or a modified version of it, with increasing subset sizes) could be shown to converge; however, we do not have either the time or the skills needed to perform this type of analysis ourselves. We also believe that application of the normal types of convergence proof would fail to capture the reasons why our algorithm is better than gradient descent, and it would be very hard to obtain convergence results that were strong enough to be interesting.

Our motivation for the work presented here is twofold: firstly, we are interested in large-scale non-convex optimization problems where the parameter dimension and the number of training samples is large and the Hessian has large condition number. We had previously investigated quite different approaches based on preconditioned SGD to solve an instance of this type of optimization problem (our method could be viewed as an extension to Le Roux et al. (2007)), but after reading Martens (2010) our interest switched to methods of the HF type. Secondly, we have an interest in deep neural nets, particularly to solve problems in speech recognition, and we were intrigued by the suggestion in Martens (2010) that the use of optimization methods of this type might remove the necessity for pretraining, which would result in a welcome simplification. Other recent work on the usefulness of second order methods for deep neural networks includes Bengio and Glorot (2010) and Le et al. (2011).

2 The Hessian matrix and the Gauss-Newton matrix

The Hessian matrix $\mathbf{H}$ (that is, the matrix of second derivatives w.r.t. the parameters) can be used in HF optimization whenever it is guaranteed positive semidefinite, i.e. when minimizing functions that are convex in the parameters. For non-convex problems, it is possible to substitute a positive definite approximation to the Hessian. One option is the Fisher information matrix,

$$\mathbf{F} = \sum_i \mathbf{g}_i \mathbf{g}_i^T, \quad (3)$$

where indices i correspond to samples and the $\mathbf{g}_i$ quantities are the gradients for each sample. This is a suitable stand-in for the Hessian because it is in a certain sense dimensionally the same, i.e. it changes the same way under transformations of the parameter space. If the model can be interpreted as producing a probability or likelihood, it is possible under certain assumptions (including model correctness) to show that close to convergence, the Fisher and Hessian matrices have the same expected value. The use of the Fisher matrix in this way is known as Natural Gradient Descent (Amari, 1998); in Le Roux et al. (2007), a low-rank approximation of the Fisher matrix was used instead. Another alternative that has less theoretical justification but which seems to work better in practice in the case of neural networks is the Gauss-Newton matrix, or rather a slight generalization of the Gauss-Newton matrix that we will now describe.

2.1 The Gauss-Newton matrix

The Gauss-Newton matrix is defined when we have a function (typically nonlinear) from a vector to a vector, $f : \mathbb{R}^n \to \mathbb{R}^m$. Let the Jacobian of this function be $\mathbf{J} \in \mathbb{R}^{m \times n}$, then the Gauss-Newton matrix is $\mathbf{G} = \mathbf{J}^T\mathbf{J}$, with $\mathbf{G} \in \mathbb{R}^{n \times n}$. If the problem is least-squares on the output of f, then $\mathbf{G}$ can be thought of as one term in the Hessian on the input to f. In its application to neural-network training, for each training example we consider the network as a nonlinear function from the neural-network parameters $\boldsymbol{\theta}$ to the output of the network, with the neural-network input treated as a constant. As in Schraudolph (2002), we generalize this from least squares to general convex error functions by using the expression $\mathbf{J}^T\mathbf{H}\mathbf{J}$, where $\mathbf{H}$ is the (positive semidefinite) second derivative of the error function w.r.t. the neural network output. This can be thought of as the part of the Hessian that remains after ignoring the nonlinearity of the neural-network in the parameters. In the rest of this document, following Martens (2010) we will refer to this matrix $\mathbf{J}^T\mathbf{H}\mathbf{J}$ simply as the Gauss-Newton matrix, or $\mathbf{G}$, and depending on the context, we may actually be referring to the summation of this expression over a number of neural-network training samples.

2.2 Efficiently multiplying by the Gauss-Newton matrix

As described in Schraudolph (2002), it is possible to efficiently multiply a vector by $\mathbf{G}$ using a version of the "Pearlmutter trick"; the algorithm is similar in spirit to backprop and for completeness we give it here as Algorithm 1; however, the reader should feel free to skip over this section if this level of detail is not required.

Our notation and our derivation for this algorithm differ from Pearlmutter (1994), Schraudolph (2002), and we will explain this briefly; we find our approach easier to follow. The basic idea is to write down an algorithm that efficiently computes the inner product of the Gauss-Newton matrix with two given vectors (i.e. $s = \boldsymbol{\theta}_2^T\mathbf{G}\boldsymbol{\theta}_1$), and then use reverse-mode automatic differentiation (similar to neural-net backprop) to compute the derivative of this scalar w.r.t. $\boldsymbol{\theta}_2$, which will equal the desired product $\mathbf{G}\boldsymbol{\theta}_1$.

First we will explain how we compute the inner product. Imagine that we are given a parameter vector $\boldsymbol{\theta}$, and two vectors $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ which we interpret as directions in parameter space; we want to write down an algorithm that computes the scalar $s = \boldsymbol{\theta}_2^T\mathbf{G}\boldsymbol{\theta}_1$. Assume the neural-network input is given and fixed; let $\mathbf{v}$ be the network output, and write it as $\mathbf{v}(\boldsymbol{\theta})$ to emphasize the dependence on the parameters, and then let $\mathbf{v}_1$ be defined as

$$\mathbf{v}_1 = \lim_{\alpha \to 0} \frac{1}{\alpha} \left(\mathbf{v}(\boldsymbol{\theta} + \alpha\boldsymbol{\theta}_1) - \mathbf{v}(\boldsymbol{\theta}) \right), \tag{4}$$

so that $\mathbf{v}_1 = \mathbf{J}\boldsymbol{\theta}_1$. We define $\mathbf{v}_2$ similarly. These can both be computed in a modified forward pass through the network, using forward-mode automatic differentiation. Then, if $\mathbf{H}$ is the Hessian of the error function in the output of the network (taken at parameter value $\boldsymbol{\theta}$), s is given by

$$s = \mathbf{v}_2^T\mathbf{H}\mathbf{v}_1, \tag{5}$$

since $\mathbf{v}_2^T\mathbf{H}\mathbf{v}_1 = \boldsymbol{\theta}_2^T\mathbf{J}^T\mathbf{H}\mathbf{J}\boldsymbol{\theta}_1 = \boldsymbol{\theta}_2^T\mathbf{G}\boldsymbol{\theta}_1$. The Hessian $\mathbf{H}$ of the error function would typically not be constructed as a matrix, but we would compute (5) given some analytic expression for $\mathbf{H}$. This Hessian $\mathbf{H}$ w.r.t. the output activations for a particular sample should not be confused with the Hessian w.r.t. the parameter vector, even though we use the same letter. Suppose we have written down the algorithm for computing s (we have not done so here because of space constraints). Then we treat $\boldsymbol{\theta}_1$ as a fixed quantity, but compute the derivative of s w.r.t. $\boldsymbol{\theta}_2$, taking $\boldsymbol{\theta}_2$ around zero for convenience. This derivative equals the desired product $\mathbf{G}\boldsymbol{\theta}_1$. This is how we obtained Algorithm 1. In the algorithm we denote the derivative of s w.r.t. a quantity x by $\hat{x}$, i.e. by adding a hat. We note that in this algorithm, we have a "backward pass" for quantities with subscript 2, which did not appear in the forward pass, because they were zero (since we take $\boldsymbol{\theta}_2 = 0$) and we optimized them out.

Something to note here is that when the linearity of the last layer is softmax and the error is negated cross-entropy (equivalently negated log-likelihood, if the label is known), we actually view the softmax nonlinearity as part of the error function. This is a closer approximation to the Hessian, and the error function remains positive semidefinite.

To explain the notation of Algorithm 1: $\mathbf{h}^{(i)}$ is the input to the nonlinearity of the i'th layer and $\mathbf{v}^{(i)}$ is the output; $\odot$ means elementwise multiplication; $\phi^{(i)}$ is the nonlinear function of the i'th layer, and when we apply it to vectors it acts elementwise; $\mathbf{W}^{(1)}$ is the neural-network weights for the first layer (so $\mathbf{h}^{(1)} = \mathbf{W}^{(1)}\mathbf{v}^{(0)}$, and so on); we use the subscript 1 for quantities that represent how quantities change when we move the parameters in direction $\boldsymbol{\theta}_1$ (as in Eq. (4)). The error function is written as $\mathcal{E}(\mathbf{v}^{(L)}, y)$ (where L is the last layer), and y, which may be a discrete value, a scalar or a vector, represents the supervision information which the network is trained with. Typically $\mathcal{E}$ would represent a squared loss or negated cross-entropy. In the squared-loss case, the quantity $\frac{\partial^2}{\partial \mathbf{v}^2}\mathcal{E}(\mathbf{v}^{(L)}, y)$ in Line 10 of Algorithm 1 is just the unit matrix. The other case we deal with here is negated cross entropy. We include the soft-max nonlinearity in the error function, treating the elements of the output layer $\mathbf{v}^{(L)}$ as unnormalized log probabilities. If the elements of $\mathbf{v}^{(L)}$ are written as v_j and we let $\mathbf{p}$ be the vector of probabilities, with $p_j = \exp(v_j)/\sum_i \exp(v_i)$, then the matrix $\mathbf{H}$ of second derivatives is given by

$$\frac{\partial^2}{\partial \mathbf{v}^2}\mathcal{E}(\mathbf{v}^{(L)}, y) = \mathrm{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^T. \qquad (6)$$

Algorithm 1 Compute product $\hat{\boldsymbol{\theta}}_2 = \mathbf{G}\boldsymbol{\theta}_1$: MultiplyG$(\boldsymbol{\theta}, \boldsymbol{\theta}_1, \mathbf{x}, y)$

1: // Note, $\boldsymbol{\theta} = (\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots)$ and $\boldsymbol{\theta}_1 = (\mathbf{W}_1^{(1)}, \mathbf{W}_2^{(2)}, \ldots)$.
2: $\mathbf{v}^{(0)} \leftarrow \mathbf{x}$
3: $\mathbf{v}_1^{(0)} \leftarrow \mathbf{0}$
4: **for** $l = 1 \ldots L$ **do**
5: $\quad\mathbf{h}^{(l)} \leftarrow \mathbf{W}^{(l)}\mathbf{v}^{(l-1)}$
6: $\quad\mathbf{h}_1^{(l)} \leftarrow \mathbf{W}^{(l)}\mathbf{v}_1^{(l-1)} + \mathbf{W}_1^{(l)}\mathbf{v}^{(l-1)}$
7: $\quad\mathbf{v}^{(l)} \leftarrow \phi^{(l)}(\mathbf{h}^{(l)})$
8: $\quad\mathbf{v}_1^{(l)} \leftarrow \phi'^{(l)}(\mathbf{h}^{(l)}) \odot \mathbf{h}_1^{(l)}$
9: **end for**
10: $\hat{\mathbf{v}}_2^{(L)} \leftarrow \frac{\partial^2}{\partial \mathbf{v}^2}\mathcal{E}(\mathbf{v}^{(L)}, y)\mathbf{v}_1^{(L)}$
11: **for** $l = L \ldots 1$ **do**
12: $\quad\hat{\mathbf{h}}_2^{(l)} \leftarrow \hat{\mathbf{v}}_2^{(l)} \odot \phi'^{(l)}(\mathbf{h}^{(l)})$
13: $\quad\hat{\mathbf{v}}_2^{(l-1)} \leftarrow {\mathbf{W}^{(l)}}^T \hat{\mathbf{h}}_2^{(l)}$
14: $\quad\hat{\mathbf{W}}_2^{(l)} \leftarrow \hat{\mathbf{h}}_2^{(l)} \, {\mathbf{v}^{(l-1)}}^T$
15: **end for**
16: **return** $\hat{\boldsymbol{\theta}}_2 \equiv \left(\hat{\mathbf{W}}_2^{(1)}, \ldots, \hat{\mathbf{W}}_2^{(L)}\right)$

3 Krylov Subspace Descent: overview

Now we describe our method, and how it relates to Hessian Free (HF) optimization. The discussion in the previous section (on the Hessian versus Gauss-Newton matrix) is orthogonal to the distinction between KSD and HF, because either method can use any Hessian substitute, with the proviso that our method can use the Hessian even when it is not positive definite.

In the rest of this section we will use $\mathbf{H}$ to refer to either the Hessian or a substitute such as $\mathbf{G}$ or $\mathbf{F}$. In Martens (2010) and in the work we describe here, these matrices are approximated using a subset of data samples. In both HF and KSD, the whole computation is preconditioned using the diagonal of the Fisher matrix $\mathbf{F}$ (since this is easy to compute); however, in this overview we will gloss over this preconditioning. In HF, on each iteration the CG algorithm is used to approximately compute

$$\mathbf{d} = -(\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{g}, \qquad (7)$$

where $\mathbf{d}$ is the step direction, and $\mathbf{g}$ is the gradient. As described in Martens (2010), CG aims to minimize the function $\frac{1}{2}\mathbf{x}^T(\mathbf{H}+\lambda\mathbf{I})\mathbf{x} - \mathbf{x}^T\mathbf{g}$ which is a quadratic approximation of our objective function. The approximate solution $\mathbf{d}_{CG}$ reached after K iterations of CG will lie in the Krylov subspace of dimension K spanned by $\{\mathbf{g}, (\mathbf{H}+\lambda\mathbf{I})\mathbf{g}, \ldots, (\mathbf{H}+\lambda\mathbf{I})^{K-1}\mathbf{g}\}$. This is easy to see by looking at the CG algorithm.

In HF, the step size to take in the direction $\mathbf{d}_{CG}$ is determined by a backtracking line search. The value of λ is kept updated by Levenburg-Marquardt style heuristics. Other heuristics are used to control the stopping of the CG iterations. In addition, the CG iterations for optimizing $\mathbf{d}$ are not initialized from zero (which would be the natural choice) but from the previous value of $\mathbf{d}$; this loses some convergence guarantees but seems to improve performance, perhaps by adding a kind of momentum to the updates.

In our method, we compute an orthogonal basis $\mathbf{P}$ for the subspace spanned by $\{\mathbf{g}, \mathbf{H}\mathbf{g}, \ldots, \mathbf{H}^{K-1}\mathbf{g}, \mathbf{d}_{\text{prev}}\}$, which is the Krylov subspace of dimension K generated by $\mathbf{g}$ and $\mathbf{H}$, augmented with the previous search direction. Note that the Krylov subspace of dimension $\mathbf{K}$ generated by $\mathbf{g}$ and $\mathbf{H}+\lambda\mathbf{I}$ is the same as that generated by $\mathbf{g}$ and $\mathbf{H}$, which is easy to verify. Our method optimizes the objective function f over this subspace using BFGS, approximating the objective function using a subset of samples. Our BFGS phase may be viewed as a modification of the line search phase of HF, but done in a higher dimension and using a subset of the data. The BFGS phase uses a different subset of data from that used to compute the Hessian; if we used the same subset we would get a very biased estimate of the optimal step to take within the subspace.

4 Krylov Subspace Descent in detail

In this section we describe the details of the KSD algorithm, including the preconditioning.

For notation purposes: on iteration n of the overall optimization we will write the training data set used to obtain the gradient as $\mathcal{A}_n$ (which is always the entire dataset in our experiments); the set used to compute the Hessian or Hessian substitute as $\mathcal{B}_n$; and the set used for BFGS optimization over the subspace, as $\mathcal{C}_n$. For clarity when dealing with multiple subset sizes, we will typically normalize all quantities by the number of samples: that is, objective function values, gradients, Hessians and the like will always be divided by the number of samples in the set over which they were computed.

On each iteration we will compute a diagonal preconditioning matrix $\mathbf{D}$ (we omit the subscript n). $\mathbf{D}$ is expected to be a rough approximation to the Hessian. In our experiments, following Martens (2010), we set $\mathbf{D}$ to the diagonal of the Fisher matrix computed over $\mathcal{A}_n$. To precondition, we define a normalized parameter vector $\tilde{\boldsymbol{\theta}} = \mathbf{D}^{1/2}\boldsymbol{\theta}$, compute the Krylov subspace in terms of $\tilde{\boldsymbol{\theta}}$, and convert back to the "canonical" coordinates. The result is the subspace spanned by the vectors

$$\left\{(\mathbf{D}^{-1}\mathbf{H})^k\mathbf{D}^{-1}\mathbf{g}, 0 \leq k < K\right\} \tag{8}$$

We add into this subspace the previous search direction $\mathbf{d}_{\text{prev}}$, and we optimize over the resulting subspace with BFGS. Including the previous search direction in the subspace is inspired by one of the features of the HF implementation of Martens (2010), and modestly improves optimization performance. The algorithm to compute an orthogonal basis for the subspace, and the Hessian (or Hessian substitute) within it, is given as Algorithm 2, which starts with the gradient direction $\mathbf{g}$ and successively multiplies by $\mathbf{D}^{-1}\mathbf{H}$, while applying Gram-Schmidt orthogonalization to the resulting vectors to obtain an orthonormal basis.

The complete algorithm is given as Algorithm 3. The most important parameter is K, the dimension of the Krylov subspace (e.g. 20). The flooring constant ϵ is (we believe) an unimportant parameter; we used 10^{-4}. The subset sizes may be important; we recommend that $\mathcal{A}_n$ should be all of the training data, and $\mathcal{B}_n$ and $\mathcal{C}_n$ should each be about $1/K$ of the training data, and disjoint from each other but not from $\mathcal{A}_n$. This is the subset size that keeps the computation approximately balanced between the gradient computation, subspace construction and subspace optimization. Implementations of the BFGS algorithm would typically also have parameters: for instance, parameters of the line-search algorithm and stopping criteria; however, we expect that in practice these would not have too much effect on performance because the algorithm is likely to converge almost exactly (since the subspace dimension and the number of iterations are about the same).

Algorithm 2 Construct basis $\mathbf{P} = [\mathbf{p}_1, \ldots, \mathbf{p}_{K+1}]$ for the subspace, and the Hessian (or substitute) $\bar{\mathbf{H}}$ in the co-ordinates of the subspace.

1: $\mathbf{p}_1 \leftarrow \mathbf{D}^{-1}\mathbf{g}$
2: $\mathbf{p}_1 \leftarrow \frac{1}{||\mathbf{p}_1||_2}\mathbf{p}_1$
3: **for** $k = 1 \ldots K+1$ **do**
4:   $\mathbf{w} \leftarrow \mathbf{H}\mathbf{p}_k$ // If Gauss-Newton matrix, computed with Algorithm 1; if the Hessian, see Pearlmutter (1994).
5:   **if** $k < K$ **then**
6:     $\mathbf{u} \leftarrow \mathbf{D}^{-1}\mathbf{w}$ // $\mathbf{u}$ will be $\mathbf{p}_{m+1}$
7:   **else if** $k = K$ **then**
8:     $\mathbf{u} \leftarrow \mathbf{d}_{\text{prev}}$ // Previous search direction; use arbitrary nonzero vector if 1st iteration
9:   **end if**
10:   **for** $j = 1 \ldots k$ **do**
11:     $\bar{h}_{k,j} \leftarrow \mathbf{w}^T\mathbf{p}_j$ // Compute element of reduced-dimension Hessian
12:     $\mathbf{u} \leftarrow \mathbf{u} - (\mathbf{u}^T\mathbf{p}_j)\mathbf{p}_j$ // Orthogonalize $\mathbf{u}$
13:   **end for**
14:   **if** $k \leq K$ **then**
15:     $\mathbf{p}_{k+1} \leftarrow \frac{1}{||\mathbf{u}||_2}\mathbf{u}$ // Normalize length and set next direction.
16:   **end if**
17: **end for**
18: // Now set upper triangle of $\bar{\mathbf{H}}$ to lower triangle.

Each iteration of Algorithm 3 computes a Krylov subspace of dimension K from the gradient and the Hessian or Hessian substitute, and optimizes over this subspace using BFGS with the objective function approximated using a data subset $\mathcal{C}_n$. Lines 7 to 9 are an additional preconditioning step to help the BFGS to converge faster, in which we try to find new coordinates in which $\bar{\mathbf{H}}$ is the unit matrix. Line 7 is needed to handle cases where $\bar{\mathbf{H}}$ has zero or negative eigenvalues. The flooring described in Line 7 may be done as follows: do the Singular Value Decomposition $\bar{\mathbf{H}} = \mathbf{U}\mathbf{D}\mathbf{V}^T$, then let $\hat{\mathbf{D}}$ be a floored version of $\mathbf{D}$, with diagonal elements $\hat{d}_i = \max(d_i, \epsilon \max_i d_i)$; then let $\hat{\mathbf{H}} = \mathbf{U}\hat{\mathbf{D}}\mathbf{U}^T$ (note: the use of $\mathbf{U}$ on both sides is not a typo). This has the effect of flipping the sign of negative eigenvalues, and then imposing a floor of ϵ times the largest eigenvalue.

5 Experiments

To evaluate KSD, we performed several experiments to compare it with SGD and with other second order optimization methods[2], namely L-BFGS and HF.

[2]Note: we may properly speak of HF and KGD as second-order methods only when $\mathbf{H}$ is the actual Hessian matrix

Dataset	Train smp.	Test smp.	Input	Output	Model	Task
CURVES	20K	10K	784 (bin.)	784 (bin.)	400-200-100-50-25-5	AE
MNIST_{AE}	60K	10K	784 (bin.)	784 (bin.)	1000-500-250-30	AE
MNIST_{CL}	60K	10K	784 (bin.)	10 (class)	500-500-2000	Class
$\text{MNIST}_{CL,PT}$[1]	60K	10K	784 (bin.)	10 (class)	500-500-2000	Class
Aurora	1.2M	100K[2]	352 (real)	56 (class)	512-1024-1536	Class
Starcraft	900	100	5077 (mix)	8 (class)	10	Class

Table 1: Datasets and models used in our setup.

Algorithm 3 Krylov Subspace Descent

1: $\mathbf{d}_{\text{prev}} \leftarrow \mathbf{e}_1$ // or any arbitrary nonzero vector
2: **for** $n = 1, 2 \ldots$ **do**
3: // Sample three sets from training data, $\mathcal{A}_n$, $\mathcal{B}_n$ and $\mathcal{C}_n$.
4: $\mathbf{g} \leftarrow \frac{1}{|\mathcal{A}_n|} \sum_{i \in \mathcal{A}_n} \mathbf{g}_i(\boldsymbol{\theta})$ // Get average function gradient over this batch.
5: Set $\mathbf{D}$ to diagonal of Fisher matrix on $\mathcal{A}_n$, floored to ϵ times its maximum.
6: Run Algorithm 2 to find $\mathbf{P}$ and $\bar{\mathbf{H}}$ on subset $\mathcal{B}_n$
7: Let $\hat{\mathbf{H}}$ be the result of flooring the eigenvalues of $\bar{\mathbf{H}}$ to ϵ times the maximum.
8: Do the Cholesky decomposition $\hat{\mathbf{H}} = \mathbf{C}\mathbf{C}^T$
9: Let $\bar{\mathbf{P}} = \mathbf{P}\mathbf{C}^{-T}$ (do this in-place; $\mathbf{C}^{-T}$ is upper triangular)
10: $\mathbf{a} \leftarrow 0 \in \mathbb{R}^{K+1}$
11: Find the value $\mathbf{a}^*$ that minimizes the objective function measured on $\mathcal{C}_n$, using about K iterations of BFGS, with objective function measured at $\boldsymbol{\theta} + \bar{\mathbf{P}}\mathbf{a}$ and gradient $\bar{\mathbf{P}}^T\mathbf{g}$ (where $\mathbf{g}$ is the gradient w.r.t. the parameters, measured at parameter-value $\boldsymbol{\theta} + \bar{\mathbf{P}}\mathbf{a}$).
12: $\mathbf{d}_{\text{prev}} \leftarrow \bar{\mathbf{P}}\mathbf{a}^*$
13: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{d}_{\text{prev}}$
14: **end for**

We report both training and cross validation errors, and running time (we terminated the algorithms with an early stopping rule using held-out validation data). Our implementations of both KSD and HF are based on Matlab using Jacket[3] to perform the expensive matrix operations on a Geforce GTX580 GPU with 1.5GB of memory.

5.1 Datasets and models

Here we describe the datasets that we used to compare KSD to other methods.

- CURVES: Artificial dataset consisting of curves at 28×28 resolution. The dataset consists of 20K training samples, and 10K testing samples. We considered an autoencoder network, as in Hinton and Salakhutdinov (2006).

- MNIST: Single digit vision classification task. The digits are 28×28 pixels, with a 60K training, and 10K testing samples. We considered both an autoencoder network, and classification (Hinton and Salakhutdinov, 2006).

- Aurora: Spoken digits dataset, with different levels of real noise (airport, train station, ...). We used Perceptual Linear Prediction features and performed classification of 56 English phones. These frame level phone error rates are the ones reported in Table 2. Also reported in the text are Word Error Rates, which were produced by using the phone posteriors in a Tandem system, concatenated with standard MFCC to train a Hidden Markov Model with Gaussian Mixture Model emissions. Further details on the setup can be found in Vinyals and Ravuri (2011).

- Starcraft: The dataset consists of a real time strategy video game sequences from 1000 games. The goal is to predict the strategy the opponent chose based on a fully observed game sequence after five minutes, and features contain orderings between buildings, presence/absence features, or times that certain buildings were built.

The models (i.e. network architectures) for each dataset are summarized in Table 1. We tried to explore a wide variety of models covering different sizes, input and output characteristics, and tasks. Note that the error reported for the autoencoder (AE) task is the L2 norm squared between input and output, and for the classification (Class) task is the classification error (i.e. 100 - accuracy). The non linearities considered were logistic functions for all the hidden layers except for the "coding" layer (i.e. middle layer) in the autencoders, which was linear, and the visible layer for

[3] www.accelereyes.com

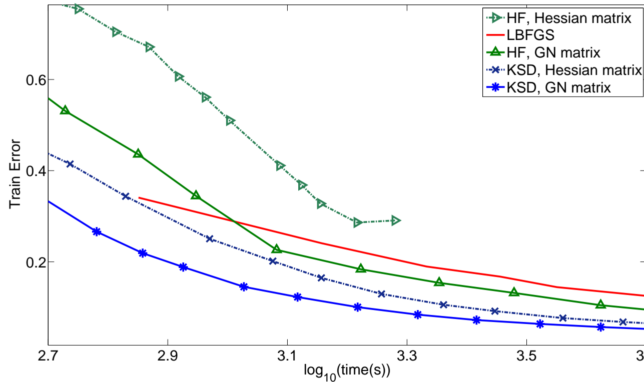

Figure 1: Aurora convergence curves for various algorithms.

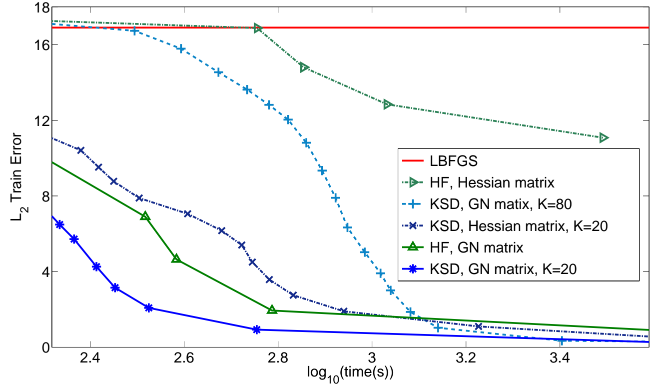

Figure 2: CURVES convergence curves for various algorithms.

	HF			KSD		
Dataset	Tr. err.	CV err.	Time	Tr. err.	CV err.	Time
CURVES	0.13	**0.19**	1	0.17	0.25	0.2
MNIST_{AE}	1.7	2.7	1	1.8	**2.5**	0.2
MNIST_{CL}	0%	2.01%	1	0%	**1.70%**	0.6
$\text{MNIST}_{CL,PT}$	0%	1.40%	1	0%	**1.29%**	0.6
Aurora	5.1%	8.7%	1	4.5%	**8.1%**	0.3
Starcraft	0%	11%	1	0%	**5%**	0.7

Table 2: Results comparing two second order methods: Hessian Free and Krylov Subspace Descent. Time reported is relative to the running time of HF (lower than 1 means faster).

classification, which was softmax.

5.2 Results and discussion

Table 2 summarizes our results. We observe that KSD converges faster than HF, and tends to lead to lower generalization error. Our implementation for the two methods is almost identical; the steps that dominate the computation (computing objective functions, gradients and Hessian or Gauss-Newton products) are shared between both and are computed on a GPU.

For all the experiments we used the Gauss-Newton matrix unless otherwise specified. The dimensionality of the Krylov subspace was set to 20, the number of BFGS iterations was set to 30 (although in many cases the optimization on the projected gradients converged before reaching 30), and an L2 regularization term was added to the objective function. However, motivated by the observation that on CURVES, HF tends to use a large number of iterations, we experimented with a larger subspace dimension of $K = 80$ and these are the numbers we report in Table 2.

For comparability in memory usage with KSD, we used a moving window of size 10 for the L-BFGS methods. We do not show SGD performance in Figures 1 and 2 as it was worse than L-BFGS.

When using HF or KSD, pre-training helped significantly in the MNIST classification task, but not for the other tasks (we do not show the results with pre-training in the other cases; there was no substantial difference in training or testing errors). However, when using SGD or CG for optimization (results not shown), pre-training helped on all tasks except Starcraft (which is not a deep network). This is consistent with the notion put forward in Martens (2010) that it might be possible to do away with the need for pre-training if we use powerful second-order optimization methods. The one case in which pre-training helped even when using HF or KSD, is MNIST; this dataset had zero training errors, which in consistent with the regularization interpretation of pre-training which is put forward in Erhan et al. (2010). Our experiments support the notion that when using advanced second-order optimization methods and when overfitting is not a major issue, pre-training is not necessary.

In Figures 1 and 2, we show the convergence of KSD

[1]For $\text{MNIST}_{CL,PT}$ we initialize the weights using pre-training RBMs as in Hinton and Salakhutdinov (2006). In the other experiments, we did not find a significant difference between pretraining and random initialization as in Martens (2010).

[2]We report both classification error rate on a 100K CV set, and word error rate on a 5M testing set with different levels of noise

and HF with both the Hessian and Gauss-Newton matrices. HF eventually "gets stuck" when using the Hessian; the algorithm was not designed to be used for non-positive definite matrices, and the CG routine terminates when it detects a non-descent direction. Even before getting stuck, it is clear that it does not work well with the actual Hessian. Our method also works better with the Gauss-Newton matrix than with the Hessian, although the difference is smaller. Our method is always faster than HF and L-BFGS.

6 Conclusion and future work

In this paper, we proposed a new second order optimization method. Our approach relies on efficiently computing the matrix-vector product between the Hessian (or a PSD approximation to it), and a vector. Unlike Hessian Free (HF) optimization, we do not require the approximation of the Hessian to be PSD, and our method requires fewer heuristics; however, it requires more memory.

Our planned future work in this direction includes investigating the circumstances under which pre-training is necessary: that is, we would like to confirm our statement that pre-training is not necessary when using sufficiently advanced optimization methods, as long as overfitting is not the main issue. Current work shows that the presented method is also able to efficiently train recursive neural networks, with no need to use the structural damping of the Gauss-Newton matrix proposed in Martens and Sutskever (2011).

Acknowledgments

We would like to thank James Martens and Ilya Sutskever for useful discussions. We would also like to thank the anonymous reviewers for helping improve the paper. Oriol Vinyals would like to acknowledge the Microsoft Research Fellowship.

References

Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural Computation*, 10:251–276, 1998.

Yoshua Bengio and Xavier Glorot. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS 2010*, volume 9, pages 249–256, May 2010.

Richard H. Byrd, Gillian M. Chiny, Will Neveitt, and Jorge Nocedal. On the use of stochastic hessian information in optimization methods for machine learning. *(submitted for publication)*, 2010.

Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *The Journal of Machine Learning Research*, 11:625–660, 2010.

Geoffrey Hinton and Ruslan Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Science*, 313(5786):504 – 507, 2006.

Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y. Ng. On optimization methods for deep learning. In *ICML*, 2011.

Nicolas Le Roux, Yoshua Bengio, and Pierre antoine Manzagol. Topmoumoute online natural gradient algorithm. In *NIPS*, 2007.

James Martens. Deep learning via Hessian-free optimization. In *ICML*, 2010.

James Martens and Ilya Sutskever. Learning Recurrent Neural Networks with Hessian-Free Optimization. In *ICML*, 2011.

José Luis Morales and Jorge Nocedal. Enriched Methods for Large-Scale Unconstrained Optimization. *Computational Optimization and Applications*, 21: 143–154, 2000.

Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

Barak A. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 6:147–160, 1994.

Nicol N. Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. In *Neural Computation*, 2002.

Oriol Vinyals and Suman Ravuri. Comparing Multilayer Perceptron to Deep Belief Network Tandem Features for Robust ASR. In *ICASSP*, 2011.