Krylov Subspace Descent for Deep Learning

Oriol Vinyals  
EECS Department, U.C. Berkeley, CA  
vinyals@eecs.berkeley.edu

Daniel Povey  
Microsoft Research, Redmond, WA  
dpovey@microsoft.com

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 [6], 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 learning the parameters of 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 semi-definite 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. Commonly, the problem that we are trying to solve consists of minimizing the negated log-likelihood:

$$f(\theta) = - \log(p(Y|X; \theta)) = - \sum_{i=1}^{N} \log(p(y_i|x_n; \theta))$$

(1)

where \((X, Y)\) 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 \(\theta\). 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 method we describe is intended to minimize (1) as quickly as possible, and like SGD, it makes use of the fact that (1) is a sum over many terms.

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:

$$\theta_{m+1} = \theta_m - \alpha H_m^{-1} g_m,$$

(2)

where \(H_m\) and \(g_m\) are, respectively, the Hessian and the gradient on iteration \(m\) of the objective function (1); here, \(\alpha\) would be chosen to minimize (1) at \(\theta_{m+1}\). For high dimensional problems it is not practical to invert the Hessian; however, we can efficiently approximate (2) using only
multiplication by $H_m$, by using the Conjugate Gradients (CG) method with a truncated number of iterations. In addition, it is possible to multiply by $H_m$ without explicitly forming it, using what is known as the “Pearlmutter trick” [10] (although it was known to the optimization community prior to that; see [9, Chapter 8]) for multiplying an arbitrary vector by the Hessian; this is described for neural networks but is applicable to quite general types of functions. This type of optimization method is known as “truncated Newton” or “Hessian-free inexact Newton” [8]. In [2], this method is applied but using only a subset of data to approximate the Hessian $H_m$. A more sophisticated version of the same idea was described in the earlier paper [6], 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 substituting the Gauss-Newton matrix for the Hessian.

In this paper we propose Krylov Subspace Descent (KSD), which is quite similar to the method described in [6], referred to as Hessian Free (HF) in the rest of the paper. 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 $(H_m + \lambda I)^{-1}g_m$ using truncated CG, we compute a basis for the Krylov subspace spanned by $g_m, H_m g_m, \ldots, H_m^{K-1} g_m$ 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 $\lambda$, the approximate solution to $H_m + \lambda I$ found by $K$ iterations of CG will lie in this subspace, so we are in effect automatically choosing the optimal $\lambda$ in the Levenburg-Marquardt smoothing method of HF (although our algorithm is free to choose a solution more general than this). 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 $\lambda$.
- Generality: unlike HF, our method can be applied even if $H$ (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.
- Convergence properties: the use of a subset of data to optimize over the subspace will prevent convergence to an optimum.

Our immediate motivation for investigating these types of methods was that we needed optimization methods that could rapidly train large neural networks (e.g. for speech recognition) on very large numbers of training samples. Our method shares many features of its implementation with HF (e.g. the same code may be used for multiplying by the Hessian or its substitutes), but it is fundamentally different in that on each iteration the gradient and Hessian matrix do not provide a single search direction to search in, but rather a whole subspace, over which we efficiently optimize the original nonlinear objective function using a subset of samples.

Note that our method can only be considered a true second order method when using the actual Hessian.

2 Krylov Subspace Descent: overview

Now we describe our method, and how it relates to Hessian Free (HF) optimization. The discussion 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.

It is also possible to substitute in the Fisher information matrix, another positive definite approximate to the Hessian, defined as,

$$ F = \sum_i g_i g_i^T, \quad (3) $$

2
where indices \(i\) correspond to samples and the \(g_i\) quantities are the gradients for each sample. In the rest of this section we will use \(H\) to refer to either the Hessian or a substitute such as the Gauss-Newton matrix \(G\) or \(F\). In [6] and 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 \(F\) (since this is easy to compute); however, in the discussion below we will gloss over this preconditioning. In HF, on each iteration the CG algorithm is used to approximate compute

\[
d = -(H + \lambda I)^{-1}g,
\]

where \(d\) is the step direction, and \(g\) is the gradient. The step size is determined by a backtracking line search. The value of \(\lambda\) 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 \(d\) are not initialized from zero (which would be the natural choice) but from the previous value of \(d\); this loses some convergence guarantees but seems to improve performance, perhaps by adding a kind of momentum to the updates.

In our method (again glossing over preconditioning), we compute a basis for the subspace spanned by \(\{g, Hg, \ldots, H^{K-1}g, d_{\text{prev}}\}\), which is the Krylov subspace of dimension \(K\), augmented with the previous search direction. We then optimize the objective function over this subspace using BFGS, approximating the objective function using a subset of samples.

\section{Krylov Subspace Descent in detail}

In this section we describe the details of the KSD algorithm\footnote{For a complete description of the algorithm, the reader can find a Technical Report version of this article under the same title in http://www.arxiv.org}. For notation purposes: on iteration \(n\) of the overall optimization we will define three random subsets of the data as follows: the training data set used to obtain the gradient as \(A_n\) (which is always the entire dataset in our experiments); the set used to compute the Hessian or Hessian substitute as \(B_n\); and the set used for BFGS optimization over the subspace, as \(C_n\). 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 \(D\) (we omit the subscript \(n\)). \(D\) is expected to be a rough approximation to the Hessian. In our experiments, following [6], we set \(D\) to the diagonal of the Fisher matrix computed over \(A_n\). To precondition, we define a new variable \(\theta = D^{1/2}\theta\), compute the Krylov subspace in terms of this variable, and convert back to the “canonical” co-ordinates. The result is the subspace spanned by the vectors

\[
\{ (D^{-1}H)^k D^{-1}g, 0 \leq k < K \}
\]

We adjoin the previous step direction \(d_{\text{prev}}\) to this, and it becomes the subspace we optimize over with BFGS.

On each iteration of optimization, after computing an orthogonal basis \(V\) for (5), we do a further preconditioning step within the subspace using the reduced-dimension Hessian \(H\), which gives us a new, non-orthogonal basis \(V\) for the subspace. This step is done to help the BFGS converge faster.

The complete algorithm is given as Algorithm 1. We observed that the most important parameter was \(K\), the dimension of the Krylov subspace (20 was used for most experiments). The flooring constant \(\epsilon\) was set to 10\(^{-5}\). The subset sizes are important; we recommend that \(A_n\) should be all of the training data, and \(B_n\) and \(C_n\) should each be about \(1/K\) of the training data, and disjoint from each other but not from \(A_n\). 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 1 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: // Obtain three sets from training data, \( \mathcal{A}_n, \mathcal{B}_n \) and \( \mathcal{C}_n \).
4: \( \mathbf{g} \leftarrow \frac{1}{|\mathcal{A}_n|} \sum_{i \in \mathcal{A}_n} \mathbf{g}_i(\theta) \) // Get average function gradient over this batch.
5: Set \( \mathbf{D} \) to diagonal of Fisher matrix on \( \mathcal{A}_n \), floored to \( \epsilon \) times its maximum.
6: Find \( \mathbf{V} \) and \( \bar{\mathbf{H}} \) on subset \( \mathcal{B}_n \) // Orthogonal basis for subspace defined in Eq. 5.
7: Let \( \hat{\mathbf{H}} \) be the result of flooring the eigenvalues of \( \bar{\mathbf{H}} \) to \( \epsilon \) times the maximum.
8: Do the Cholesky decomposition \( \hat{\mathbf{H}} = \mathbf{C} \mathbf{C}^T \).
9: Let \( \bar{\mathbf{V}} = \mathbf{V} \mathbf{C} \) − \( \mathbf{C}^T \) (do this in-place; \( \mathbf{C}^T \) is upper triangular).
10: \( a \leftarrow 0 \in \mathbb{R}^{K+1} \)
11: Find the optimum \( a^* \) with BFGS for about \( K \) iterations using the subset \( \mathcal{C}_n \), with objective function measured at \( \theta + \bar{V}a \) and gradient \( \hat{\mathbf{V}}^T \mathbf{g} \) (where \( \mathbf{g} \) is the gradient w.r.t. \( \theta \)).
12: \( \mathbf{d}_{\text{prev}} \leftarrow \bar{V}a^* \)
13: \( \theta \leftarrow \theta + \mathbf{d}_{\text{prev}} \)
14: end for

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
MNIST,CL,PT | 60K | 10K | 784 (bin.) | 10 (class) | 500-500-2000 | Class
Aurora | 1.2M | 100K² | 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.

4 Experiments

To evaluate KSD, we performed several experiments to compare it with SGD and with other second order optimization methods, namely L-BFGS and HF. 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\(^2\) to perform the expensive matrix operations on a Geforce GTX580 GPU with 1.5GB of memory.

4.1 Datasets and models

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

- CURVES: Artificial dataset consisting of curves at \( 28 \times 28 \) resolution. The dataset consists of 20K training samples, and 10K testing samples. We considered an autoencoder network, as in [4].
- MNIST: Single digit vision classification task. The digits are \( 28 \times 28 \) pixels, with a 60K training, and 10K testing samples. We considered both an autoencoder network, and classification [4].
- Aurora: Spoken digits dataset, with different levels of real noise (airport, train station, ...). We used PLP 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 [12].
- 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.

\(^2\)www.accelereyes.com
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).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>HF Tr. err.</th>
<th>CV err.</th>
<th>Time</th>
<th>KSD Tr. err.</th>
<th>CV err.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURVES</td>
<td>0.13</td>
<td>0.19</td>
<td>1</td>
<td>0.17</td>
<td>0.25</td>
<td>0.2</td>
</tr>
<tr>
<td>MNIST_{AE}</td>
<td>1.7</td>
<td>2.7</td>
<td>1</td>
<td>1.8</td>
<td>2.5</td>
<td>0.2</td>
</tr>
<tr>
<td>MNIST_{CL}</td>
<td>0%</td>
<td>2.01%</td>
<td>1</td>
<td>0%</td>
<td>1.70%</td>
<td>0.6</td>
</tr>
<tr>
<td>MNIST_{CL, PT}</td>
<td>0%</td>
<td>1.40%</td>
<td>1</td>
<td>0%</td>
<td>1.29%</td>
<td>0.6</td>
</tr>
<tr>
<td>Aurora</td>
<td>5.1%</td>
<td>8.7%</td>
<td>1</td>
<td>4.5%</td>
<td>8.1%</td>
<td>0.3</td>
</tr>
<tr>
<td>Starcraft</td>
<td>0%</td>
<td>11%</td>
<td>1</td>
<td>0%</td>
<td>5%</td>
<td>0.7</td>
</tr>
</tbody>
</table>

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 autoencoders, which was linear, and the visible layer for classification, which was softmax.

4.2 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 compatibility in memory usage with KSD, we used a moving window of size 10 for the L-BFGS methods. We do not report 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 significant difference). 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 [6] that it might be possible to do away with the need for pre-training if we use powerful second-order optimization methods. The one exception to this, MNIST, has zero training error when using HF and KSD, which is consistent with a regularization interpretation of pre-training. This is opposite to the conclusions reached in [3] (their conclusion was that pre-training helps by finding a better “basin of attraction”), but that paper was not using these types of optimization methods. 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. We are not giving this issue the attention it deserves, since the primary focus of this paper is on our optimization method; we may try to support these conclusions more convincingly in future work.

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

---

1For MNIST_{CL, PT} we initialize the weights using pretraining RBMs as in [4]. In the other experiments, we did not find a significant difference between pretraining and random initialization as in [6].

2We report both classification error rate on a 100K CV set, and word error rate on a 5M testing set with different levels of noise.
than with the Hessian, although the difference is smaller. Our method is always faster than HF and L-BFGS.

Figure 1: Aurora convergence curves for various algorithms.

Figure 2: CURVES convergence curves for various algorithms.

5 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 [7].

References