Fast large-scale optimization by unifying stochastic gradient and quasi-Newton methods

Jascha Sohl-Dickstein, Ben Poole, Surya Ganguli
Stanford University
{jascha,benpoole,sganguli}@stanford.edu

Abstract

We present an algorithm for minimizing a sum of functions that combines the computational efficiency of stochastic gradient descent (SGD) with the second order curvature information leveraged by quasi-Newton methods. We unify these approaches by maintaining an independent Hessian approximation for each contributing function in the sum. We maintain computational tractability and limit memory requirements even for high dimensional optimization problems by storing and manipulating these quadratic approximations in a shared, time evolving, low dimensional subspace. Each update step requires only a single contributing function or minibatch evaluation (as in SGD), and each step is scaled using an approximate inverse Hessian and little to no adjustment of hyperparameters is required (as is typical for quasi-Newton methods). This algorithm contrasts with earlier stochastic second order techniques that treat the Hessian of each contributing function as a noisy approximation to the full Hessian, rather than as a target for direct estimation. We experimentally demonstrate improved convergence on seven diverse optimization problems. The algorithm is released as open source Python and MATLAB packages.

This paper is a condensed version of a paper published at ICML 2014 [33].

1 Introduction

A common problem in optimization is to find a vector $\mathbf{x}^* \in \mathcal{R}^M$ which minimizes a function $F(\mathbf{x})$, where $F(\mathbf{x})$ is a sum of N computationally cheaper differentiable subfunctions $f_i(\mathbf{x})$,

$$F(\mathbf{x}) = \sum_{i=1}^{N} f_i(\mathbf{x}), \tag{1}$$

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}} F(\mathbf{x}). \tag{2}$$

There are two general approaches to efficiently optimizing a function of this form. The first is to use a quasi-Newton method [10], of which BFGS [7, 12, 13, 30] or LBFGS [20] are the most common choices. The second approach is to use Stochastic Gradient Descent (SGD) [26, 6] or a more recent variant [4, 27, 21, 22, 1]. Combining quasi-Newton and stochastic gradient methods could improve optimization time, and reduce the need to tweak optimization hyperparameters. This problem has been approached from a number of directions [28, 34, 23, 9, 35, 19, 16, 8, 29, 11, 5]. All of these approaches treat the Hessian on a subset of the data as a noisy approximation to the full Hessian. To reduce noise in the Hessian approximation, they rely on regularization and very large minibatches to descend $F(\mathbf{x})$. Thus, each update step requires the evaluation of many subfunctions and/or yields a highly regularized (i.e. diagonal) approximation to the full Hessian.

We develop a novel second-order quasi-Newton technique that treats the full Hessian of each subfunction as a direct target for estimation, and maintains a separate quadratic approximation of each

subfunction. This approach differs from all previous work, which in contrast treats the Hessian of each subfunction as a noisy approximation to the full Hessian. Our approach allows us to combine Hessian information from multiple subfunctions in a more natural and efficient way. Moreover, we develop a novel method to maintain computational tractability and limit the memory requirements of this quasi-Newton method in the face of high dimensional optimization problems (large M). We do this by storing and manipulating the subfunctions in a shared, adaptive low dimensional subspace, determined by the recent history of the gradients and iterates.

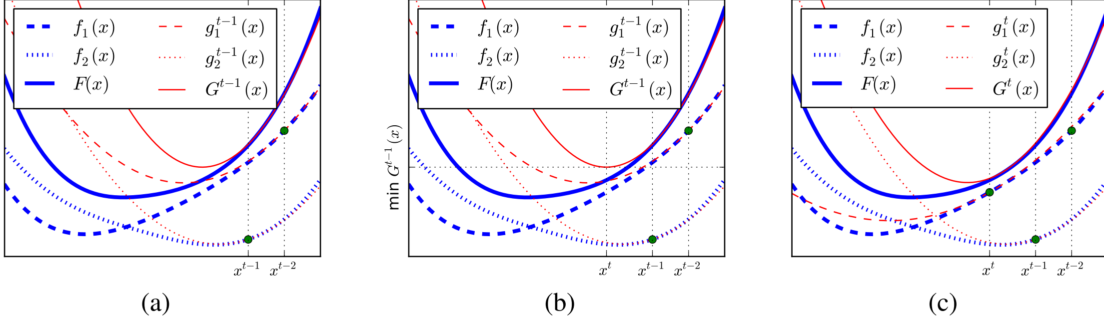


Figure 1: A cartoon illustrating the proposed optimization technique. *(a)* The objective function $F(x)$ (solid blue line) consists of a sum of two subfunctions (dashed blue lines), $F(x) = f_1(x) + f_2(x)$. At learning step $t-1$, $f_1(x)$ and $f_2(x)$ are approximated by quadratic functions $g_1^{t-1}(x)$ and $g_2^{t-1}(x)$ (red dashed lines). The sum of the approximating functions $G^{t-1}(x)$ (solid red line) approximates the full objective $F(x)$. The green dots indicate the parameter values at which each subfunction has been evaluated *(b)* The next parameter setting x^t is chosen by minimizing the approximating function $G^{t-1}(x)$ from the prior update step. *(c)* After each parameter update, the quadratic approximation for one of the subfunctions is updated using a second order expansion around the new parameter vector x^t. The constant and first order term in the expansion are evaluated exactly, and the second order term is estimated by performing BFGS on the subfunction's history. In this case the approximating subfunction $g_1^t(x)$ is updated (long-dashed red line). This update is also reflected by a change in the full approximating function $G^t(x)$ (solid red line). Optimization proceeds by repeating these two illustrated update steps. In order to remain tractable in memory and computational overhead, optimization is performed in an adaptive low dimensional subspace determined by the history of gradients and iterates.

2 Algorithm

The algorithm is described in detail in Appendix A. In brief, it consists of the following:

We define a series of functions $G^t(\mathbf{x})$ intended to approximate $F(\mathbf{x})$,

$$G^t(\mathbf{x}) = \sum_{i=1}^{N} g_i^t(\mathbf{x}), \tag{3}$$

where the superscript t indicates the learning iteration. Each $g_i^t(\mathbf{x})$ serves as a quadratic approximation to the corresponding $f_i(\mathbf{x})$,

$$g_i^t(\mathbf{x}) = f_i(\mathbf{x}^t) + (\mathbf{x} - \mathbf{x}^t)^T f_i'(\mathbf{x}^t) + \frac{1}{2}(\mathbf{x} - \mathbf{x}^t)^T \mathbf{H}_i^t (\mathbf{x} - \mathbf{x}^t) \approx f_i(\mathbf{x}). \tag{4}$$

The functions $g_i^t(\mathbf{x})$ are stored, and one of them is updated per learning step.

Optimization is performed by alternating between: minimizing the quadratic approximate objective function $G^{t-1}(\mathbf{x})$ in order to select the next iterate $\mathbf{x}^t$; and evaluating a single subfunction j at $\mathbf{x}^t$ and updating the corresponding quaratic approximation $g_j^t(\mathbf{x})$. This is illustrated in Figure 1. The approximate Hessian term $\mathbf{H}_i^t$ is set by running BFGS on the stored history for subfunction i.

The dimensionality M of $\mathbf{x} \in \mathcal{R}^M$ is typically large. As a result, the memory and computational cost of working directly with the matrices $\mathbf{H}_i^t \in \mathcal{R}^{M \times M}$ is typically prohibitive, as is the cost of

storing the history terms $\Delta f'$ and $\Delta \mathbf{x}$ required by BFGS. To reduce the dimensionality from M to a tractable value, all history is instead stored and all updates computed in a lower dimensional subspace, with dimensionality between K_{min} and K_{max}. This subspace is constructed such that it includes the most recent gradient and position for every subfunction, and thus $K_{min} \geq 2N$. This guarantees that the subspace includes both the steepest gradient descent direction over the full batch, and the update directions from the most recent parameter update. The subspace is represented by the orthonormal columns of a matrix $\mathbf{P}^t \in \mathcal{R}^{M \times K^t}$, $\left(\mathbf{P}^t\right)^T \mathbf{P}^t = \mathbf{I}$. K^t is the subspace dimensionality at optimization step t.

At each optimization step, an additional column is added to the subspace, expanding it to include the most recent gradient direction. This is done by first finding the component in the gradient vector which lies outside the existing subspace, and then appending that component to the current subspace. In order to prevent the subspace from growing too large, whenever $K^t > K_{max}$ the subspace is collapsed to only include the most recent gradient and position measurements from each subfunction. This collapse is performed by way of a QR decomposition on the most recent gradients and iterates.

The only portion of this algorithm with non-negligible computational cost is the projection of the gradients and iterates in to and out of the low dimensional subspace. This overhead is discussed in detail in the appendix.

3 Experimental Results

Open source Python and MATLAB code which implements the proposed technique, and which directly generates the plots in this paper, is provided at `https://github.com/Sohl-Dickstein/Sum-of-Functions-Optimizer`.

We compared our optimization technique to several competing optimization techniques for seven objective functions. The results are illustrated in Figures 2 and 3. *SFO* refers to Sum of Functions Optimizer, and is the new algorithm presented in this paper. *SAG* refers to Stochastic Average Gradient method, with the trailing number providing the Lipschitz constant. *SGD* refers to Stochastic Gradient Descent, with the trailing number indicating the step size. *ADAGrad* indicates the AdaGrad algorithm, with the trailing number indicating the initial step size. *LBFGS* refers to the limited memory BFGS algorithm. *LBFGS minibatch* repeatedly chooses one tenth of the subfunctions, and runs LBFGS for ten iterations on them. *Hessian-free* refers to Hessian-free optimization.

For *SAG*, *SGD*, and *ADAGrad* the hyperparameter was chosen by a grid search. The best hyperparameter value, and the hyperparameter values immediately larger and smaller in the grid search, are shown in the plots and legends for each model in Figure 2. In *SGD+momentum* the two hyperparameters for both step size and momentum coefficient were chosen by a grid search, but only the best parameter values are shown. The grid-searched momenta were 0.5, 0.9, 0.95, and 0.99, and the grid-searched step lengths were all integer powers of ten between 10^{-5} and 10^{2}. For *Hessian-free*, the hyperparameters, source code, and objective function are identical to those used in [23], and the training data was divided into four "chunks." For all other experiments and optimizers the training data was divided into $N = 100$ minibatches (or subfunctions).

A detailed description of all target objective functions is included in Appendix D. The logistic regression and Ising model / Hopfield objectives are convex, and are plotted relative to their global minimum. The global minimum was taken to be the smallest value achieved on the objective by any optimizer. In Figure 3, a twelve layer neural network was trained on cross entropy reconstruction error for the CURVES dataset. This objective, and the parameter initialization, was chosen to be identical to an experiment in [23].

References

[1] F Bach and E Moulines. Non-strongly-convex smooth stochastic approximation with convergence rate O (1/n). *Neural Information Processing Systems*, 2013.

[2] AJ Bell and TJ Sejnowski. An information-maximization approach to blind separation and blind deconvolution. *Neural computation*, 1995.

[3] J Bergstra and O Breuleux. Theano: a CPU and GPU math expression compiler. *Proceedings of the Python for Scientific Computing Conference (SciPy)*, 2010.

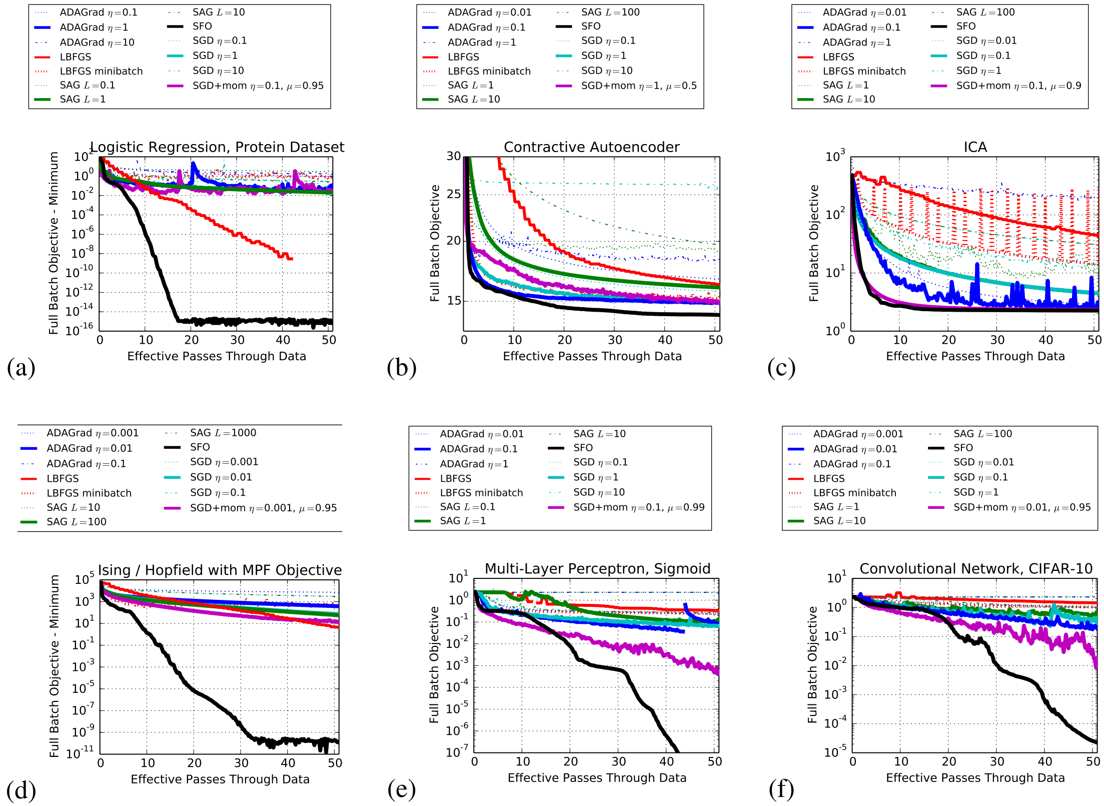


Figure 2: A comparison of SFO to competing optimization techniques for six objective functions. The bold lines indicate the best performing hyperparameter for each optimizer. Note that unlike all other techniques besides LBFGS, SFO does not require tuning hyperparameters (for instance, the displayed SGD+momentum traces are the best out of 32 hyperparameter configurations). The objective functions shown are *(a)* a logistic regression problem, *(b)* a contractive autoencoder trained on MNIST digits, *(c)* an Independent Component Analysis (ICA) model trained on MNIST digits, *(d)* an Ising model / Hopfield associative memory trained using Minimum Probability Flow, *(e)* a multi-layer perceptron with sigmoidal units trained on MNIST digits, and *(f)* a multilayer convolutional network with rectified linear units trained on CIFAR-10. The logistic regression and MPF Ising objectives are convex, and their objective values are plotted relative to the global minimum.

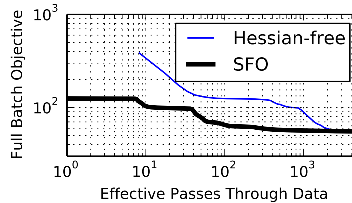


Figure 3: A comparison of SFO to Hessian-free optimization for a twelve layer neural network trained on the CURVES dataset. This problem is identical to an experiment in [23], and the Hessian-free convergence trace was generated using source code from that same paper. SFO converges in approximately one tenth the number of effective passes through the data as Hessian-free optimization.

[4] Doron Blatt, Alfred O Hero, and Hillel Gauchman. A convergent incremental gradient method with a constant step size. *SIAM Journal on Optimization*, 18(1):29–51, 2007.

[5] Antoine Bordes, Léon Bottou, and Patrick Gallinari. SGD-QN: Careful quasi-Newton stochastic gradient descent. *The Journal of Machine Learning Research*, 10:1737–1754, 2009.

[6] Léon Bottou. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nimes*, 91:8, 1991.

[7] CG Broyden. The convergence of a class of double-rank minimization algorithms 2. The new algorithm. *IMA Journal of Applied Mathematics*, 1970.

[8] RH Byrd, SL Hansen, J Nocedal, and Y Singer. A Stochastic Quasi-Newton Method for Large-Scale Optimization. *arXiv preprint arXiv:1401.7020*, 2014.

[9] RH Richard H Byrd, GM Gillian M Chin, Will Neveitt, and Jorge Nocedal. On the use of stochastic hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21(3):977–995, 2011.

[10] John E Dennis Jr and Jorge J Moré. Quasi-Newton methods, motivation and theory. *SIAM review*, 19(1):46–89, 1977.

[11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2010.

[12] R Fletcher. A new approach to variable metric algorithms. *The computer journal*, 1970.

[13] D Goldfarb. A family of variable-metric methods derived by variational means. *Mathematics of computation*, 1970.

[14] IJ Goodfellow and D Warde-Farley. Maxout networks. *arXiv:1302.4389*, 2013.

[15] IJ Goodfellow and D Warde-Farley. Pylearn2: a machine learning research library. *arXiv:1308.4214*, 2013.

[16] P Hennig. Fast probabilistic optimization from noisy gradients. *International Conference on Machine Learning*, 2013.

[17] Christopher Hillar, Jascha Sohl-Dickstein, and Kilian Koepsell. Efficient and optimal binary Hopfield associative memory storage using minimum probability flow. *Redwood Center Technical Report*, (arXiv 1204.2916), April 2012.

[18] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv:1207.0580*, 2012.

[19] Chih-Jen Lin, Ruby C Weng, and S Sathiya Keerthi. Trust region newton method for logistic regression. *The Journal of Machine Learning Research*, 9:627–650, 2008.

[20] Dong C DC Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[21] J Mairal. Optimization with First-Order Surrogate Functions. *International Conference on Machine Learning*, 2013.

[22] Julien Mairal. Incremental Majorization-Minimization Optimization with Application to Large-Scale Machine Learning. *arXiv:1402.4419*, February 2014.

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

[24] JM Papakonstantinou. *Historical Development of the BFGS Secant Method and Its Characterization Properties*. 2009.

[25] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 833–840, 2011.

[26] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, pages 400–407, 1951.

[27] N Le Roux, M Schmidt, and F Bach. A Stochastic Gradient Method with an Exponential Convergence Rate for Finite Training Sets. *NIPS*, 2012.

[28] Nicol Schraudolph, Jin Yu, and Simon Günter. A stochastic quasi-Newton method for online convex optimization. *AIstats*, 2007.

[29] Nicol N Schraudolph. Local gain adaptation in stochastic gradient descent. In *Artificial Neural Networks, 1999. ICANN 99. Ninth International Conference on (Conf. Publ. No. 470)*, volume 2, pages 569–574. IET, 1999.

[30] DF Shanno. Conditioning of quasi-Newton methods for function minimization. *Mathematics of computation*, 1970.

[31] Jascha Sohl-Dickstein, Peter Battaglino, and Michael DeWeese. New Method for Parameter Estimation in Probabilistic Models: Minimum Probability Flow. *Physical Review Letters*, 107(22):11–14, November 2011.

[32] Jascha Sohl-Dickstein, Peter B. Battaglino, and Michael R. DeWeese. Minimum Probability Flow Learning. *International Conference on Machine Learning*, 107(22):11–14, November 2011.

[33] Jascha Sohl-Dickstein, Ben Poole, and Surya Ganguli. Fast large-scale optimization by unifying stochastic gradient and quasi-Newton methods. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 604–612, 2014.

[34] Peter Sunehag, Jochen Trumpf, S V N Vishwanathan, and Nicol Schraudolph. Variable metric stochastic approximation theory. *arXiv preprint arXiv:0908.3529*, August 2009.

[35] Oriol Vinyals and Daniel Povey. Krylov subspace descent for deep learning. *arXiv preprint arXiv:1111.4259*, 2011.

Appendix

A Algorithm Full Description

Our goal is to combine the benefits of stochastic and quasi-Newton optimization techniques. We first describe the general procedure by which we optimize the parameters $\mathbf{x}$. We then describe the construction of the shared low dimensional subspace which makes the algorithm tractable in terms of computational overhead and memory for large problems. This is followed by a description of the BFGS method by which an online Hessian approximation is maintained for each subfunction. Finally, we end this section with a review of implementation details.

A.1 Approximating Functions

We define a series of functions $G^t(\mathbf{x})$ intended to approximate $F(\mathbf{x})$,

$$G^t(\mathbf{x}) = \sum_{i=1}^{N} g_i^t(\mathbf{x}), \tag{5}$$

where the superscript t indicates the learning iteration. Each $g_i^t(\mathbf{x})$ serves as a quadratic approximation to the corresponding $f_i(\mathbf{x})$. The functions $g_i^t(\mathbf{x})$ will be stored, and one of them will be updated per learning step.

A.2 Update Steps

As is illustrated in Figure 1, optimization is performed by repeating the steps:

1. Choose a vector $\mathbf{x}^t$ by minimizing the approximating objective function $G^{t-1}(\mathbf{x})$,

   $$\mathbf{x}^t = \underset{\mathbf{x}}{\operatorname{argmin}}\, G^{t-1}(\mathbf{x}). \tag{6}$$

   Since $G^{t-1}(\mathbf{x})$ is a sum of quadratic functions $g_i^{t-1}(\mathbf{x})$, it can be exactly minimized by a Newton step,

   $$\mathbf{x}^t = \mathbf{x}^{t-1} - \eta^t \left(\mathbf{H}^{t-1}\right)^{-1} \frac{\partial G^{t-1}\left(\mathbf{x}^{t-1}\right)}{\partial \mathbf{x}}, \tag{7}$$

   where $\mathbf{H}^{t-1}$ is the Hessian of $G^{t-1}(\mathbf{x})$. The step length η^t is typically unity, and will be discussed in Section **??**.

2. Choose an index $j \in \{1...N\}$, and update the corresponding approximating subfunction $g_i^t(\mathbf{x})$ using a second order power series around $\mathbf{x}^t$, while leaving all other subfunctions unchanged,

$$g_i^t(\mathbf{x}) = \begin{cases} g_i^{t-1}(\mathbf{x}) & i \neq j \\ \begin{bmatrix} f_i(\mathbf{x}^t) \\ \quad + \left(\mathbf{x} - \mathbf{x}^t\right)^T f_i'(\mathbf{x}^t) \\ \quad + \frac{1}{2}\left(\mathbf{x} - \mathbf{x}^t\right)^T \mathbf{H}_i^t \left(\mathbf{x} - \mathbf{x}^t\right) \end{bmatrix} & i = j \end{cases}. \tag{8}$$

The constant and first order term in Equation 8 are set by evaluating the subfunction and gradient, $f_j(\mathbf{x}^t)$ and $f_j'(\mathbf{x}^t)$. The quadratic term $\mathbf{H}_j^t$ is set by using the BFGS algorithm to generate an online approximation to the true Hessian of subfunction j based on its history of gradient evaluations (see Section A.4). The Hessian of the summed approximating function $G^t(\mathbf{x})$ in Equation 7 is the sum of the Hessians for each $g_j^t(\mathbf{x})$, $\mathbf{H}^t = \sum_j \mathbf{H}_j^t$.

A.3 A Shared, Adaptive, Low-Dimensional Representation

The dimensionality M of $\mathbf{x} \in \mathcal{R}^M$ is typically large. As a result, the memory and computational cost of working directly with the matrices $\mathbf{H}_i^t \in \mathcal{R}^{M \times M}$ is typically prohibitive, as is the cost of storing the history terms $\Delta f'$ and $\Delta \mathbf{x}$ required by BFGS (see Section A.4). To reduce the dimensionality from M to a tractable value, all history is instead stored and all updates computed in a lower dimensional subspace, with dimensionality between K_{min} and K_{max}. This subspace is constructed such that it includes the most recent gradient and position for every subfunction, and thus $K_{min} \geq 2N$. This guarantees that the subspace includes both the steepest gradient descent direction over the full batch, and the update directions from the most recent Newton steps (Equation 7).

For the results in this paper, $K_{min} = 2N$ and $K_{max} = 3N$. The subspace is represented by the orthonormal columns of a matrix $\mathbf{P}^t \in \mathcal{R}^{M \times K^t}$, $\left(\mathbf{P}^t\right)^T \mathbf{P}^t = \mathbf{I}$. K^t is the subspace dimensionality at optimization step t.

A.3.1 Expanding the Subspace with a New Observation

At each optimization step, an additional column is added to the subspace, expanding it to include the most recent gradient direction. This is done by first finding the component in the gradient vector which lies outside the existing subspace, and then appending that component to the current subspace,

$$\mathbf{q}_{\text{orth}} = f_j'\left(\mathbf{x}^t\right) - \mathbf{P}^{t-1}\left(\mathbf{P}^{t-1}\right)^T f_j'\left(\mathbf{x}^t\right), \tag{9}$$

$$\mathbf{P}^t = \begin{bmatrix} \mathbf{P}^{t-1} & \frac{\mathbf{q}_{\text{orth}}}{||\mathbf{q}_{\text{orth}}||} \end{bmatrix}, \tag{10}$$

where j is the subfunction updated at time t. The new position $\mathbf{x}^t$ is included automatically, since the position update was computed within the subspace $\mathbf{P}^{t-1}$. Vectors embedded in the subspace $\mathbf{P}^{t-1}$ can be updated to lie in $\mathbf{P}^t$ simply by appending a 0, since the first K^{t-1} dimensions of $\mathbf{P}^t$ consist of $\mathbf{P}^{t-1}$.

A.3.2 Restricting the Size of the Subspace

In order to prevent the dimensionality K^t of the subspace from growing too large, whenever $K^t > K_{max}$, the subspace is collapsed to only include the most recent gradient and position measurements from each subfunction. The orthonormal matrix representing this collapsed subspace is computed by a QR decomposition on the most recent gradients and positions. A new collapsed subspace is thus computed as,

$$\mathbf{P}' = \text{orth}\left(\left[f_1'\left(\mathbf{x}^{\tau_1^t}\right) \cdots f_N'\left(\mathbf{x}^{\tau_N^t}\right) \quad \mathbf{x}^{\tau_1^t} \cdots \mathbf{x}^{\tau_N^t}\right]\right), \tag{11}$$

where τ_i^t indicates the learning step at which the ith subfunction was most recently evaluated, prior to the current learning step t. Vectors embedded in the prior subspace $\mathbf{P}$ are projected into the new subspace $\mathbf{P}'$ by multiplication with a projection matrix $\mathbf{T} = \left(\mathbf{P}'\right)^T \mathbf{P}$. Vector components which point outside the subspace defined by the most recent positions and gradients are lost in this projection.

Note that the subspace $\mathbf{P}'$ lies within the subspace $\mathbf{P}$. The QR decomposition and the projection matrix $\mathbf{T}$ are thus both computed within $\mathbf{P}$, reducing the computational and memory cost (see Section B.1).

A.4 Online Hessian Approximation

An independent online Hessian approximation $\mathbf{H}_j^t$ is maintained for each subfunction j. It is computed by performing BFGS on the history of gradient evaluations and positions for that single subfunction[1].

[1] We additionally experimented with Symmetric Rank 1 [10] updates to the approximate Hessian, but found they performed worse than BFGS. See Figure D.1.

Optimizer	*Computation per pass*	*Memory use*
SFO	$\mathcal{O}\left(QN + MN^2\right)$	$\mathcal{O}\left(MN\right)$
SFO, 'sweet spot'	$\mathcal{O}\left(QN\right)$	$\mathcal{O}\left(MN\right)$
LBFGS	$\mathcal{O}\left(QN + ML\right)$	$\mathcal{O}\left(ML\right)$
SGD	$\mathcal{O}\left(QN\right)$	$\mathcal{O}\left(M\right)$
AdaGrad	$\mathcal{O}\left(QN\right)$	$\mathcal{O}\left(M\right)$
SAG	$\mathcal{O}\left(QN\right)$	$\mathcal{O}\left(MN\right)$

Table A.1: Leading terms in the computational cost and memory requirements for SFO and several competing algorithms. Q is the cost of computing the value and gradient for a single subfunction, M is the number of data dimensions, N is the number of subfunctions, and L is the number of history terms retained. "SFO, 'sweet spot"' refers to the case discussed in Section B.1.1 where the minibatch size is adjusted to match computational overhead to subfunction evaluation cost. For this table, it is assumed that $M \gg N \gg L$.

A.4.1 History Matrices

For each subfunction j, we construct two matrices, $\Delta f'$ and $\Delta \mathbf{x}$. Each column of $\Delta f'$ holds the change in the gradient of subfunction j between successive evaluations of that subfunction, including all evaluations up until the present time. Each column of $\Delta \mathbf{x}$ holds the corresponding change in the position $\mathbf{x}$ between successive evaluations. Both matrices are truncated after a number of columns L, meaning that they include information from only the prior $L + 1$ gradient evaluations for each subfunction. For all results in this paper, $L = 10$ (identical to the default history length for the LBFGS implementation used in Section 3).

A.4.2 BFGS Updates

The BFGS algorithm functions by iterating through the columns in $\Delta f'$ and $\Delta \mathbf{x}$, from oldest to most recent. Let s be a column index, and $\mathbf{B}_s$ be the approximate Hessian for subfunction j after processing column s. For each s, the approximate Hessian matrix $\mathbf{B}_s$ is set so that it obeys the secant equation $\Delta {f'}_s = \mathbf{B}_s \Delta \mathbf{x}_s$, where $\Delta {f'}_s$ and $\Delta \mathbf{x}_s$ are taken to refer to the sth columns of the gradient difference and position difference matrix respectively.

In addition to satisfying the secant equation, $\mathbf{B}_s$ is chosen such that the difference between it and the prior estimate $\mathbf{B}_{s-1}$ has the smallest weighted Frobenius norm[2]. This produces the standard BFGS update equation

$$\mathbf{B}_s = \mathbf{B}_{s-1} + \frac{\Delta {f'}_s \Delta {f'}_s^T}{\Delta {f'}_s^T \Delta \mathbf{x}_s} - \frac{\mathbf{B}_{s-1} \Delta \mathbf{x}_s \Delta \mathbf{x}_s^T \mathbf{B}_{s-1}}{\Delta \mathbf{x}_s^T \mathbf{B}_{s-1} \Delta \mathbf{x}_s}. \tag{12}$$

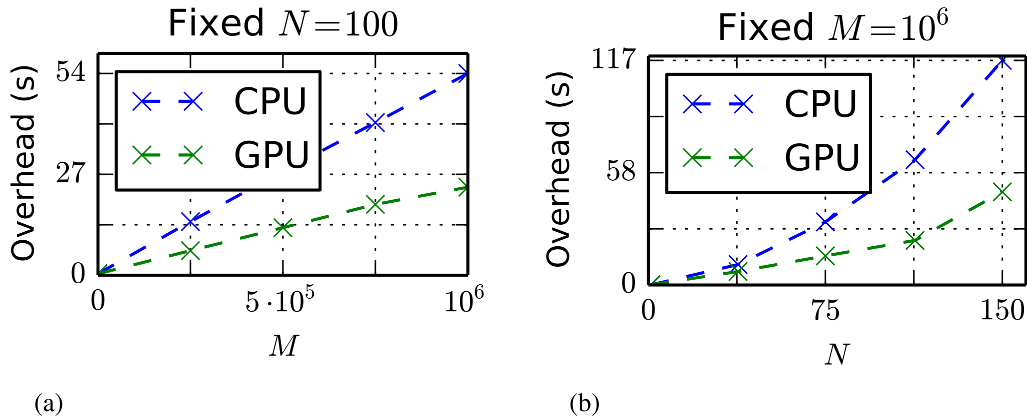
The final update is used as the approximate Hessian for subfunction j, $\mathbf{H}_j^t = \mathbf{B}_{\max(s)}$.

B Properties

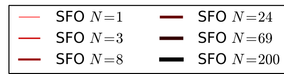
B.1 Computational Overhead and Storage Cost

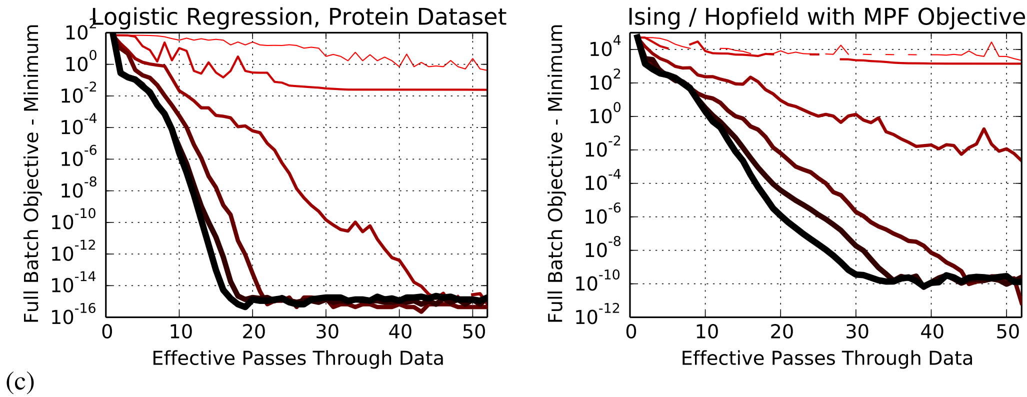
Table A.1 compares the cost of SFO to competing algorithms. The dominant computational costs are the $\mathcal{O}\left(MN\right)$ cost of projecting the M dimensional gradient and current parameter values into and out of the $\mathcal{O}\left(N\right)$ dimensional active subspace for each learning iteration, and the $\mathcal{O}\left(Q\right)$ cost of evaluating a single subfunction. The dominant memory cost is $\mathcal{O}\left(MN\right)$, and stems from storing the active subspace $\mathbf{P}^t$. Table B.1 provides the contribution to the computational cost of each component of SFO. Figure A.1 plots the computational overhead per a full pass through all the subfunctions associated with SFO as a function of M and N. If each of the N subfunctions corresponds to a minibatch, then the computational overhead can be shrunk as described in Section B.1.1.

[2]The weighted Frobenius norm is defined as $||\mathbf{E}||_{F,\mathbf{W}} = ||\mathbf{W}\mathbf{E}\mathbf{W}||_F$. For BFGS, $\mathbf{W} = \mathbf{B}_s^{-\frac{1}{2}}$ [24]. Equivalently, in BFGS the unweighted Frobenius norm is minimized after performing a linear change of variables to map the new approximate Hessian to the identity matrix.



(a) (b)





(c)

Figure A.1: An exploration of computational overhead and optimizer performance, especially as the number of minibatches or subfunctions N is adjusted. *(a)* Computational overhead required by SFO for a full pass through all the subfunctions as a function of dimensionality M for fixed $N = 100$. *(b)* Computational overhead of SFO as a function of N for fixed $M = 10^6$. Both plots show the computational time required for a full pass of the optimizer, excluding time spent computing the target objective and gradient. This time is dominated by the $\mathcal{O}\left(MN^2\right)$ cost per pass of N iterations of subspace projection. *CPU* indicates that all computations were performed on a 2012 Intel i7-3970X CPU (6 cores, 3.5 GHz). *GPU* indicates that subspace projection was performed on a GeForce GTX 660 Ti GPU. *(c)* Optimization performance on the two convex problems in Section 3 as a function of the number of minibatches N. Note that near maximal performance is achieved after breaking the target problem into only a small number of minibatches.

Without the low dimensional subspace, the leading term in the computational cost of SFO would be the far larger $\mathcal{O}\left(M^{2.4}\right)$ cost per iteration of inverting the approximate Hessian matrix in the full M dimensional parameter space, and the leading memory cost would be the far larger $\mathcal{O}\left(M^2 N\right)$ from storing an $M \times M$ dimensional Hessian for all N subfunctions.

B.1.1 Ideal Minibatch Size

Many objective functions consist of a sum over a number of data points D, where D is often larger than M. For example, D could be the number of training samples in a supervised learning problem, or data points in maximum likelihood estimation. To control the computational overhead of SFO in such a regime, it is useful to choose each subfunction in Equation 5 to itself be a sum over a minibatch of data points of size S, yielding $N = \frac{D}{S}$. This leads to a computational cost of evaluating a single subfunction and gradient of $\mathcal{O}(Q) = \mathcal{O}(MS)$. The computational cost of projecting this gradient from the full space to the shared N dimensional adaptive subspace, on the other hand, is $\mathcal{O}(MN) = \mathcal{O}\left(M\frac{D}{S}\right)$. Therefore, in order for the costs of function evaluation and projection to be the same order, the minibatch size S should be proportional to $\sqrt{D}$, yielding

$$N \propto \sqrt{D}. \tag{13}$$

The constant of proportionality should be chosen small enough that the majority of time is spent evaluating the subfunction. In most problems of interest, $\sqrt{D} \ll M$, justifying the relevance of the regime in which the number of subfunctions N is much less than the number of parameters M. Finally, the computational and memory costs of SFO are the same for sparse and non-sparse objective functions, while Q is often much smaller for a sparse objective. Thus the ideal S (N) is likely to be larger (smaller) for sparse objective functions.

Note that as illustrated in Figure A.1c and Figure 2 performance is very good even for small N.

B.2 Convergence

Concurrent work by [21] considers a similar algorithm to that described in Section A.2, but with $\mathbf{H}_i^t$ a scalar constant rather than a matrix. Proposition 6.1 in [21] shows that in the case that each g_i majorizes its respective f_i, and subject to some additional smoothness constraints, $G^t(\mathbf{x})$ monotonically decreases, and $\mathbf{x}^*$ is an asymptotic stationary point. Proposition 6.2 in [21] further shows that for strongly convex f_i, the algorithm exhibits a linear convergence rate to $\mathbf{x}^*$. A near identical proof should hold for a simplified version of SFO, with random subfunction update order, and with $\mathbf{H}_i^t$ regularized in order to guarantee satisfaction of the majorization condition.

C Computational Complexity

Here we provide a description of the computational cost of each component of the SFO algorithm. See Table B.1. The computational cost of matrix multiplication for $N \times N$ matrices is taken to be $\mathcal{O}\left(N^{2.4}\right)$.

C.1 Function and Gradient Computation

By definition, the cost of computing the function value and gradient for each subfunction is $\mathcal{O}(Q)$, and this must be done N times to complete a full effective pass through all the subfunctions, yielding a total cost per pass of $\mathcal{O}(QN)$.

C.2 Subspace Projection

Once per iteration, the updated parameter values $\mathbf{x}^t$ must be projected from the N dimensional adaptive low dimensional subspace into the full M dimensional parameter space. Similarly, once per iteration the gradient must be projected from the full M dimensional parameter space into the N dimensional subspace. See Section A.3. Additionally the residual of the gradient projection must be appended to the subspace as described in Equation 10. Each of these operations has cost $\mathcal{O}(MN)$, stemming from multiplication of a parameter or gradient vector by the subspace matrix $\mathbf{P}^t$. They are performed N times per effective pass through the data, yielding a total cost per pass of $\mathcal{O}\left(MN^2\right)$.

C.3 Subspace Collapse

In order to constrain the dimensionality K^t of the subspace to remain order $\mathcal{O}(N)$, the subspace must be collapsed every $\mathcal{O}(N)$ steps, or $\mathcal{O}(1)$ times per pass. This is described in Section A.3.2.

The collapsed subspace is computed using a QR decomposition on the history terms (see Equation 11) within the current subspace, with computational complexity $\mathcal{O}\left(N^3\right)$. The old subspace matrix $\mathbf{P}$ is then projected into the new subspace matrix $\mathbf{P}'$, involving the multiplication of a $\mathcal{O}\left(M \times N\right)$ matrix with a $\mathcal{O}\left(N \times N\right)$ projection matrix, with corresponding complexity $\mathcal{O}\left(MN^{1.4}\right)$. The total complexity per pass is thus $O\left(MN^{1.4} + N^3\right)$.

C.4 Minimize $G^t\left(\mathbf{x}\right)$

$G^t\left(\mathbf{x}\right)$ is minimized by an explicit matrix inverse in Equation 7. Computing this inverse has cost $\mathcal{O}\left(N^{2.4}\right)$, and must be performed N times per effective pass through the data.

With only small code changes, this inverse could instead be updated each iteration using the Woodbury identity and the inverse from the prior step, with cost $\mathcal{O}\left(N^2\right)$. However, minimization of $G^t\left(\mathbf{x}\right)$ is not a leading contributor to the overall computational cost, and so increasing its efficiency would have little effect.

C.5 BFGS

The BFGS iterations are performed in the $\mathcal{O}\left(L\right)$ dimensional subspace defined by the columns of $\Delta f'$ and $\Delta \mathbf{x}$. Since BFGS consists of L rank two updates of an $\mathcal{O}\left(L \times L\right)$ matrix, the cost of performing BFGS iterations is $\mathcal{O}\left(L^3\right)$. See Section A.4.2. The cost of using a QR decomposition to compute the L dimensional subspace defined by the columns of $\Delta f'$ and $\Delta \mathbf{x}$ is $\mathcal{O}\left(NL^2\right)$, and the cost of projecting the L history terms of length N into the subspace is $\mathcal{O}\left(NL^{1.4}\right)$. BFGS is performed N times per effective pass through the data. The total cost of BFGS is therefore $\mathcal{O}\left(N^2L^2 + NL^3\right)$.

In the current implementation, the full BFGS chain for a subfunction is recomputed every iteration. However, BFGS could be modified to only compute the rank two update to the prior Hessian approximation at each iteration. This would sacrifice the history-dependent initialization described in Section **??**. The resulting complexity per iteration would instead be $\mathcal{O}\left(N^2\right)$, and the computational complexity per pass would instead be $\mathcal{O}\left(N^3\right)$. BFGS is also not a leading contributor to the overall computational cost, and so increasing its efficiency would have little effect.

D Objective Functions

A more detailed description of the objective functions used for experiments in the main text follows.

D.1 Logistic Regression

We chose the logistic regression objective, L2 regularization penalty, and training dataset to be identical to the protein homology test case in the recent Stochastic Average Gradient paper [27], to allow for direct comparison of techniques. The one difference is that our total objective function is divided by the number of samples per minibatch, but unlike in [27] is not also divided by the number of minibatches. This different scaling places the hyperparameters for all optimizers in the same range as for our other experiments.

D.2 Autoencoder

We trained a contractive autoencoder, which penalizes the Frobenius norm of the Jacobian of the encoder function, on MNIST digits. Autoencoders of this form have been successfully used for learning deep representations in neural networks [25]. Sigmoid nonlinearities were used for both encoder and decoder. The regularization penalty was set to 1, and did not depend on the number of hidden units. The reconstruction error was divided by the number of training examples per minibatch. There were 784 visible units, and 256 hidden units.

D.3 Independent Components Analysis

We trained an Independent Components Analysis (ICA) [2] model with Student's t-distribution prior on MNIST digits by minimizing the negative log likelihood of the ICA model under the digit images. Both the receptive fields and the Student's t shape parameter were estimated. Digit images were preprocessed by performing PCA whitening and discarding components with variance less than 10^{-4} times the maximum variance. The objective function was divided by the number of training examples per minibatch.

D.4 Ising Model / Hopfield Network via MPF

We trained an Ising/Hopfield model on MNIST digits, using code from [17]. Optimal Hopfield network storage capacity can be achieved by training the corresponding Ising model via MPF [17, 32, 31]. The MPF objective was divided by the number of training examples per minibatch. An L2 regularization penalty with coefficient 0.01 was added to the objective for each minibatch.

D.5 Multilayer Perceptron

We trained a deep neural network to classify digits on the MNIST digit recognition benchmark. We used a similar architecture to [18]. Our network consisted of: 784 input units, one hidden layer of 1200 units, a second hidden layer of 1200 units, and 10 output units. We ran the experiment using both rectified linear and sigmoidal units. The objective used was the standard softmax regression on the output units. Theano [3] was used to implement the model architecture and compute the gradient.

D.6 Deep Convolutional Network

We trained a deep convolutional network on CIFAR-10 using max pooling and rectified linear units. The architecture we used contains two convolutional layers with 48 and 128 units respectively, followed by one fully connected layer of 240 units. This architecture was loosely based on [14]. Pylearn2 [15] and Theano were used to implement the model.

E Implementation Details

Here we expand on the implementation details for SFO. Figure D.1 provides empirical motivation for several of the design choices made in this paper, by showing the change in convergence traces when those design choices are changed. Note that even when these design choices are changed, convergence is still more rapid than for the competing techniques in Figure 2.

E.1 BFGS Initialization

No History An approximate Hessian can only be computed as described in Section A.4 after multiple gradient evaluations. If a subfunction j only has one gradient evaluation, then its approximate Hessian $\mathbf{H}_j^t$ is set to the identity times the median eigenvalue of the average Hessian of the other active subfunctions. If j is the very first subfunction to be evaluated, $\mathbf{H}_j^t$ is initialized as the identity matrix times a large positive constant (10^6).

The First BFGS Step The initial approximate Hessian matrix used in BFGS is set to a scaled identity matrix, so that $\mathbf{B}_0 = \beta\mathbf{I}$. This initialization will be overwritten by Equation 12 for all explored directions. It's primary function, then, is to set the estimated Hessian for unexplored directions. Gradient descent routines tend to progress from directions with large slopes and curvatures, and correspondingly large eigenvalues, to directions with shallow slopes and curvatures, and smaller eigenvalues. The typical eigenvalue in an unexplored direction is thus expected to be smaller than in previously explored directions. We therefore set β using a measure of the smallest eigenvalue in an explored direction

The scaling factor β is set to the smallest non-zero eigenvalue of a matrix $\mathbf{Q}$, $\beta = \min_{\lambda_Q>0} \lambda_Q$, where λ_Q indicates the eigenvalues of $\mathbf{Q}$. $\mathbf{Q}$ is the symmetric matrix with the smallest Frobenius

norm which is consistent with the squared secant equations for all columns in $\Delta f'$ and $\Delta \mathbf{x}$. That is,

$$\mathbf{Q} = \left[(\Delta \mathbf{x})^{+T} (\Delta f')^T \Delta f' (\Delta \mathbf{x})^{+} \right]^{\frac{1}{2}}, \tag{14}$$

where $^+$ indicates the pseudoinverse, and $\frac{1}{2}$ indicates the matrix square root. All of the eigenvalues of $\mathbf{Q}$ are non-negative. $\mathbf{Q}$ and λ_Q are computed in the subspace defined by $\Delta f'$ and $\Delta \mathbf{x}$, reducing computational cost (see Section B.1).

E.2 Enforcing Positive Definiteness

It is typical in quasi-Newton techniques to enforce that the Hessian approximation remain positive definite. In SFO, at the end of the BFGS procedure, each $\mathbf{H}_i^t$ is constrained to be positive definite by performing an eigendecomposition, and setting any eigenvalues which are too small to the median positive eigenvalue. The median is used because it provides a measure of "typical" curvature. When an eigenvalue is negative (or *extremely* close to 0), it provides a poor estimate of curvature over the interval required to reach a minimum in the direction of the corresponding eigenvector. Replacing it with the median eigenvalue therefore provides a more reasonable estimate. If λ_{max} is the maximum eigenvalue of $\mathbf{H}_i^t$, then any eigenvalues smaller than $\gamma \lambda_{max}$ are set to be equal to $\text{median}_{\lambda > 0} \lambda$. For all experiments shown here, $\gamma = 10^{-8}$. As described in Section A.3, a shared low dimensional representation makes this eigenvalue computation tractable.

E.3 Choosing a Target Subfunction

The subfunction j to update in Equation 8 is chosen as,

$$j = \operatorname*{argmax}_i \left[\mathbf{x}^t - \mathbf{x}^{\tau_i} \right]^T \mathbf{H}^t \left[\mathbf{x}^t - \mathbf{x}^{\tau_i} \right], \tag{15}$$

where τ_i indicates the time at which subfunction i was last evaluated.

That is, the updated subfunction is the one which was last evaluated farthest from the current location, using the approximate Hessian as a metric. This is motivated by the observation that the approximating functions which were computed farthest from the current location tend to be the functions which are least accurate at the current location, and therefore the most useful to update. This contrasts with the cyclic choice of subfunction in [4], and the random choice of subfunction in [27]. See Figure D.1 for a comparison of the optimization performance corresponding to each update ordering scheme.

E.4 Growing the Number of Active Subfunctions

For many problems of the form in Equation 1, the gradient information is nearly identical between the different subfunctions early in learning. We therefore begin with only a small number of active subfunctions, and expand the active set as learning progresses. We expand the active set by one subfunction every time the average gradient shrinks to within a factor α of the standard error in the average gradient. This comparison is performed using the inverse approximate Hessian as the metric. That is, we increment the active subset whenever

$$\left(\bar{f}'^t \right)^T \mathbf{H}^{t^{-1}} \bar{f}'^t < \alpha \frac{\sum_i \left(f_i'^t \right)^T \mathbf{H}^{t^{-1}} f_i'^t}{\left(N^t - 1 \right) N^t}, \tag{16}$$

where N^t is the size of the active subset at time t, $\mathbf{H}^t$ is the full Hessian, and $\bar{f}'^t$ is the average gradient,

$$\bar{f}'^t = \frac{1}{N^t} \sum_i f_i{}' \left(\mathbf{x}_i^t \right). \tag{17}$$

For all the experiments shown here, $\alpha = 1$, and the initial active subset size is two. We additionally increased the active active subset size by 1 when a bad update is detected (Section **??**) or when a full pass through the active batch occurs without a batch size increase. See Figure D.1 for a comparison to the case where all subfunctions are initially active.

E.5 Detecting Bad Updates

For some ill-conditioned problems, such as ICA with a Student's t-prior (see Section 3), we additionally found it necessary to identify bad proposed parameter updates. In BFGS and LBFGS, bad update detection is also performed, but it is achieved via a line search on the full objective. Since we only evaluate a single subfunction per update step, a line search on the full objective is impossible. Updates are instead labeled bad when the value of a subfunction has increased since its previous evaluation, and also exceeds its approximating function by more than the corresponding reduction in the summed approximating function (ie $f_j\left(\mathbf{x}^t\right) - g_j^{t-1}\left(\mathbf{x}^t\right) > G^{t-1}\left(\mathbf{x}^{t-1}\right) - G^{t-1}\left(\mathbf{x}^t\right)$).

When a bad update proposal is detected, $\mathbf{x}^t$ is reset to its previous value $\mathbf{x}^{t-1}$. The BFGS history matrices $\Delta f'$ and $\Delta \mathbf{x}$ are also updated to include the change in gradient in the failed update direction. Additionally, after a failed update, the update step length in Equation 7 is temporarily shortened. It then decays back towards 1 with a time constant of one data pass. That is, the step length is updated as,

$$\eta^{t+1} = \begin{cases} \frac{1}{N} + \frac{N-1}{N}\eta^t & \text{successful update} \\ \frac{1}{2}\eta^t & \text{failed update} \end{cases}. \tag{18}$$

This temporary shortening of the update step length is motivated by the observation that when the approximate Hessian for a single subfunction becomes inaccurate it has often become inaccurate for the remaining subfunctions as well, and failed update steps thus tend to co-occur.

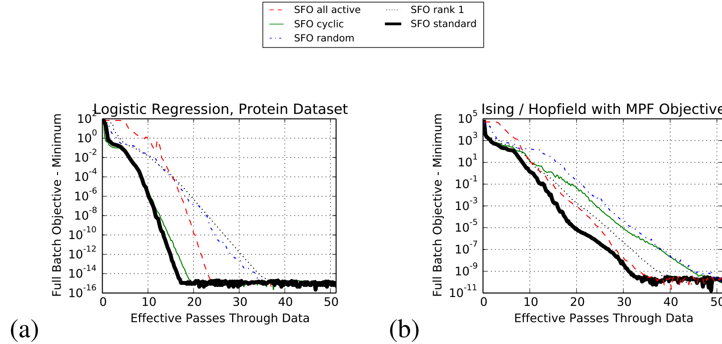


Figure D.1: SFO is insensitive to several specific design decisions. Plots showing the consequences of changing several of the design choices made in SFO for *(a)* the logistic regression objective, and *(b)* the Ising / Hopfield objective. SFO as described in this paper corresponds to the *SFO standard* line. All other lines correspond to changing a single design choice. *SFO rank 1* corresponds to using a rank 1 rather than BFGS update (Section A.4). *SFO all active* corresponds to starting optimization with all the subfunctions active. *SFO random* and *SFO cyclic* correspond to random and cyclic update ordering, rather than maximum distance ordering. For all design choices, SFO outperforms all other techniques in Figure 2.

Operation	*One time cost*	*Repeats per pass*	*Cost per pass*
Function and gradient computation	$\mathcal{O}(Q)$	$\mathcal{O}(N)$	$\mathcal{O}(QN)$
Subspace projection	$\mathcal{O}(MN)$	$\mathcal{O}(N)$	$\mathcal{O}\left(MN^2\right)$
Subspace collapse	$\mathcal{O}\left(MN^{1.4}+N^3\right)$	$\mathcal{O}(1)$	$\mathcal{O}\left(MN^{1.4}+N^3\right)$
Minimize $G^t(\mathbf{x})$	$\leq \mathcal{O}\left(N^{2.4}\right)$	$\mathcal{O}(N)$	$\leq \mathcal{O}\left(N^{3.4}\right)$
BFGS	$\leq \mathcal{O}\left(NL^2+L^3\right)$	$\mathcal{O}(N)$	$\leq \mathcal{O}\left(N^2L^2+NL^3\right)$
Total			$\mathcal{O}\left(QN+MN^2+N^{3.4}+N^2L^2+NL^3\right)$

Table B.1: Computational cost for components of SFO. Q is the cost of evaluating the objective function and gradient for a single subfunction, M is the number of parameter dimensions, N is the number of subfunctions, L is the number of history terms kept per subfunction. Typically, $M \gg N \gg L$. In this case all contributions besides $\mathcal{O}\left(QN+MN^2\right)$ become small. Additionally the number of subfunctions can be chosen such that $\mathcal{O}(QN) = \mathcal{O}\left(MN^2\right)$ (see Section B.1.1). Contributions labeled with '$\leq$' could be reduced with small changes in implementation, as described in Section C. However as also discussed, and as illustrated by Figure A.1, they are not typically the leading terms in the cost of the algorithm.