A Stochastic Quasi-Newton Optimizer for TensorFlow

Jason Chen
Department of Computer Science
Stony Brook University
Stony Brook, NY 11790
hungrchen@cs.stonybrook.edu

David Kraemer
Department of Applied Mathematics
Stony Brook University
Stony Brook, NY 11790
davidkraemer@stonybrook.edu

Abstract

We implement a stochastic damped limited memory version of the classical BFGS quasi-Newton minimization algorithm for use in the `TensorFlow` graphical computation and neural network framework. The implementation is linked to `TensorFlow` as an `ExternalOptimizerInterface` object, which circumvents architectural challenges in implementing second-order methods. We provide empirical data to show that the implementation can solve minimization problems in line with its peers. We do not report results on typical problems that are solved using `TensorFlow` because we find problems of this scale are presently computationally intractible for our implementation.

1 Introduction

Limited memory variations of classical second-order minimization algorithms can prove highly attractive in applications with high-dimensional problems or when the cost of each iteration step is prohibitive under these classical formulations. Specifically, the canonical second-order technique, Newton's method, involves explicit computation of the Hessian matrix of the objective function, whose time complexity grows with the square of the parameter dimension. Even with BFGS, which seeks to reduce the direct computational difficulty of exactly constructing the Hessian itself, encounters similar difficulties of scale. The limited memory approach is to bound this overall cost and ensure that approximate Hessians can be computed on existing hardware.

We implement a variant of stochastic second-order algorithm as described by Wang, Ma, Goldfarb, and Liu [4] for application in the `TensorFlow` library [3]. `TensorFlow` is a framework for graph-based computation models. Its primary use within the statistical learning community is in neural networks, which can be readily implemented by the library. After the model is constructed, the actual training is performed by an `Optimizer` object. In this setting, the `Optimizer` is presented with a partially observable objective function and gradient from which to perform optimization. Some typical `Optimizer` subclasses which are available in the Python API include an `GradientDescentOptimizer`[1]and an `AdagradOptimizer`[2]

The presently available optimization algorithms implemented in the Python library are either first or zeroeth-order methods, requiring access to either the objective function or its gradient. The conspicuous absence of second-order methods, such as Newton or quasi-Newton algorithms, has been a subject of much discussion inside the `TensorFlow` community[3]

[1]See source.

[2]See source.

[3]The primary discussion has occurred surrounding Issue #446, in which contributors have noted that the current infrastructure for developing second-order methods is lacking and any implementation would require considerable re-inventing the wheel. .

Preprint. Work in progress.

Following the suggestions of the community discussion, we present an implementation of Stochastic damped limited memory BFGS (SdLBFGS) for `TensorFlow` through the use of an `ExternalOptimizerInterface`. Presently used to wrap existing minimization routines implemented by outside libraries such as `SciPy`, it serves as a simple way to engage with the `TensorFlow` internals while avoiding a deep dive into its architecture.

Our work proceeds as follows. Section 2 reviews the problem proposed in Wang et al and gives an explicit formulation of SdLBFGS according to their paper. We identify small modifications made to the core pseudocode and explain their necessity. Section 3 describes our SdLBFGS implementation and how it interfaces with `TensorFlow`. Section 4 shows preliminary experimental results on the correctness of our implementation on simple problems. Importantly, we report that our present implementation is incapable of giving meaningful progress on training practical models on common datasets. In Section 5, we discuss the implications of this negative result on future development and specify areas in which our project may be continued and improved.

Source code for this project can be found at `https://github.com/DavidNKraemer/cse592-project`. Documentation, cruft removal, and user interface improvements are currently evolving.

2 Review of theory

The SdLBFGS algorithm is proposed by Wang, Ma, Goldfarb, and Liu [4] for solving nonconvex stochastic optimization problems. In particular, it solves

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x) = \mathbb{E}[F(x, \xi)] \tag{1}$$

for $F \in C^1(\mathbb{R}^n \times \mathbb{R}^d)$ and $\xi \in \mathbb{R}^d$ is a random variable with cumulative distribution function P. Here $\mathbb{E}[\cdot]$ denotes the mathematical expectation. In general F need not be convex.

The function F is readily interpreted in the context of supervised statistical learning. Given a corpus Ξ of data, we put a distribution on Ξ which reflects a sampling regime for a training set. The object of the learning process is, then, to the model parameter $x \in \mathbb{R}^n$ which minimizes F *in expectation* with respect to the training set. Provided that the sampling maintains some resemblance to the overall corpus, this gives a useful approximate parameter for the whole corpus.

The SdLBFGS algorithm emerges from two concurrent optimization techniques. First, it follows in the tradition of approximate second-order algorithms which seek to perform a version of Newton's method while avoiding the computation of the Hessian $\nabla^2 f(x)$ explicitly. The classical BFGS approach is the canoncal quasi-Newton method. Second, it follows the development of stochastic algorithms which seek to randomly sample the components of the gradient $\nabla f(x)$ so that the essential convergence properties hold in expectation. In this sense it succeeds Stochastic Gradient Descent, among other canonical methods of this class. The limited memory constraint has antecedents in both streams of optimization techniques.

Wang et al. [4] assume that Problem (1) satisfies the following assumptions:

1. There is a real lower bound for f. That is, $\inf_{x \in \mathbb{R}^n} f(x) > -\infty$.
2. The gradient ∇f is Lipschitz continuous. That is, there exists $L > 0$ such that
$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x - y\|$$
for all $x, y \in \mathbb{R}^n$. Moreover, $\nabla_{xx} F(x, \xi)$ exists and is continuous, and there exists $\kappa > 0$ with $\|\nabla^2_{xx} F(x, \varepsilon)\| \leq \kappa$ for any x, ξ.
3. The sampling regime is unbiased. That is, for the kth iterate the following hold:
$$\mathbb{E}[g(x_k, \xi_k)]_{\xi_k} = \nabla f(x_k),$$
$$\mathbb{E}[\|g(x_k, \xi_k) - \nabla f(x_k)\|^2]_{\xi_k} \leq \sigma^2,$$
where $g(x, \xi_k) = \nabla F(x, \xi_k)$ and σ^2 is the noise level of estimating $g(x, \xi_k)$.

For a convex function, $\nabla^2 f(x)$ is always positive semidefinite (see Boyd and Vandenberghe §3.1.4 [1]), but this need not be the case in for nonconvex functions. The work in Wang et al. [4] shows how

the BFGS approximate Hessian may be tweaked to ensure that positive semidefiniteness is preserved even in nonconvex cases. The overall stochastic quasi-Newton method is described in Algorithm 2, while the detailed implementation of SdLBFGS is given in Algorithm 2.

We note that the formulation of Algorithm 2 differs slightly from Wang et al. In the original paper, the use of mem in the section corresponding to lines 15 and 16 of Algorithm 2 is replaced by p. This leads to the difficulty that if p indicates memory *capacity* rather than simply the current amount of data in memory, the indices can become negative early in the iteration. Specifically, when $k < p$ in the original paper the algorithm has undefined behavior. We clarify this ambiguity in the following way: in our implementation, p indicates the total memory capacity of the procedure, regardless of the total amount of memory presently in use. The quantity mem replaces p in the SdLBFGS update step so as to prevent array index out of bounds errors when k is small (i.e., $k < p$), but both our implementation and the original paper's exhibit the same behavior whenever $k > p$.

Algorithm 1 The high level stochastic quasi-Newton minimization algorithm. Given an initial point $x_1 \in \mathbb{R}^n$, batch sizes $\langle m_k \rangle$, step sizes $\langle \alpha_k \rangle$, and a maximum memory capacity p, perform BFGS with the stochastic dampened direction update.

function SQN($x_1, \langle m_k \rangle_{k=1}^{\infty}, \langle \alpha_k \rangle_{k=1}^{\infty}, p$)
  data $\leftarrow$ *nil*()
  for $k = 1, 2, \ldots$ **do**
    $g_k \leftarrow \frac{1}{m_k} \sum_{i=1}^{m_k} g(x_k, \varepsilon_{k,i})$
    $\Delta x_k, s_{k-1}, \bar{y}_{k-1}, \rho_{k-1} \leftarrow$ SdLBFGS($g_k, {}^*(\text{data}^\top)$) ▷ Tuple unpacking
    $x_{k+1} \leftarrow x_k - \alpha_k \Delta x_k$
    append(data, $(s_{k-1}, \bar{y}_{k-1}, \rho_{k-1})$)
    if $k > p$ **then**
      pop(data)
    end if
  end for
end function

Algorithm 2 The SdLBFGS update step. The resulting output is $H_k g_k = v_p$, where H_k is the approximation of the kth iterate Hessian and g_k is the approximation of the kth iterate gradient. Note that the computation of H_k is implicit, preventing additional storage requirements.

function SdLBFGS($g_{k-1}, \langle s_j \rangle_{j=k-p}^{k-2}, \langle \bar{y}_j \rangle_{j=k-p}^{k-2}, \langle \rho_j \rangle_{j=k-p}^{k-2}$)
  mem $\leftarrow$ *min*$(p, k-1)$
  $s_{k-1} \leftarrow x_k - x_{k-1}$
  $y_{k-1} \leftarrow \frac{1}{m_{k-1}} \sum_{i=1}^{m_{k-1}} [g(x_k, \xi_{k-1,i}) - g(x_{k-1}, \xi_{k-1,i})]$
  $\gamma_k \leftarrow max\left(\frac{{y_{k-1}}^\top y_{k-1}}{{s_{k-1}}^\top y_{k-1}}, \delta\right) \geq \delta$
  $\theta \leftarrow max\left(\frac{3}{4} \frac{{\gamma_k}^{-1} {s_{k-1}}^\top s_{k-1}}{{\gamma_k}^{-1} {s_{k-1}}^\top s_{k-1} - {s_{k-1}}^\top y_{k-1}}, 1\right)$ ▷ θ preserves positive semidefiniteness.
  $\bar{y}_{k-1} \leftarrow \theta y_{k-1} + (1 - \theta {\gamma_k}^{-1} s_{k-1})$
  $\rho_{k-1} \leftarrow ({s_{k-1}}^\top \bar{y}_{k-1})^{-1}$
  for $i = 0, \ldots,$ mem $- 1$ **do**
    $\mu_i \leftarrow \rho_{k-i-1} {u_i}^\top s_{k-i-1}$
    $u_{i+1} \leftarrow u_i - \mu_i \bar{y}_{k-i-1}$
  end for
  $v_0 \leftarrow {\gamma_k}^{-1} u_{\text{mem}}$
  for $i = 0, \ldots,$ mem $- 1$ **do**
    $\nu_i \leftarrow \rho_{k-\text{mem}+i} {v_i}^\top \bar{y}_{k-\text{mem}+i}$
    $v_{i+1} \leftarrow v_i + (\mu_{\text{mem}-i-1} - \nu_i) s_{k-\text{mem}+i}$
  end for
  return $v_p, s_{k-1}, \bar{y}_{k-1}, \rho_{k-1}$
end function

3 Implementation

Our optimization object, `SQNOptimizer`, is implemented using `NumPy` [2] with `ndarray` objects serving as the primary data structures for the numerical computations in Algorithm 2. It inherits the features of the `ExternalOptimizerInterface`, which packages the objective function and its gradients from the network model in use and presents it in a `NumPy`-interoperable format.

The general framework for using built-in `Optimizer` objects follows the structure presented in Figure 1. The initialization of line 2 indicates that `optimizer` is a minimization operation, rather than the

```
1 # ... setup
2 optimizer = Optimizer(**kwargs).minimize(objective_function)
3 # ... setup
4 with tf.Session() as session:
5     # ... training setup
6     optimizer.run(**training_kwargs)
7
```

Figure 1: The typical workflow for using built-in `Optimizer` objects in a statistical learning setting. Here the `optimizer` is an operation which is configured to minimize the objective function. The actual minimization occurs inside of the session block.

result of a minimization. By contrast, for subclasses of an `ExternalOptimizerInterface`, this structure is replaced by the workflow in Figure 2.

```
1 # ... setup
2 optimizer = ExternalOptimizer(objective_function, **kwargs)
3 # ... setup
4 with tf.Session() as session:
5     # ... training setup
6     optimizer.minimize(session, **training_kwargs)
7
```

Figure 2: The workflow for using objects of `ExternalOptimizerInterface` subclasses, such as the implementation of `SQNOptimizer`. Here the `optimizer` itself is the minimizer while the `minimize` method actually performs estimation of the minimal objective function value.

Now, the `minimize` method returns the *result* of a minimization rather than a minimization operation. As `SQNOptimizer` inherits this functionality, the second workflow is required.

4 Empirical results

1. Results on simple optimization problems.
2. Results on potentially more difficult optimization problems.
3. Lack of results on MNIST with TensorFlow.

We evaluate our implementation of SdLBFGS on a collection of test functions. Figure 4 shows the effect of changing the memory capacity on a relatively small fully-observable convex differentiable problem. Figure 3. Additionally, we compare the performance of our implementation of SdLBFGS with other common minimizers on a dataset[4] for a Higgs boson classification problem. Empirical results in Figure 5 are interesting simply by virtue of lacking any theoretical justification for convergence, as the loss function is not C^1 with Lipschitz gradient. The results when the loss function is in C^1 with Lipschitz gradient, as in Figure 6, however, do conform to the theoretical guarantees.

We do not report on results of using the `SQNOptimizer` to solve standard neural network problems in the `TensorFlow` framework. Preliminary results suggest that the present implementation scales insufficiently well for use in large scale machine learning problems.

[4]The data was provided by Professor Orabona.

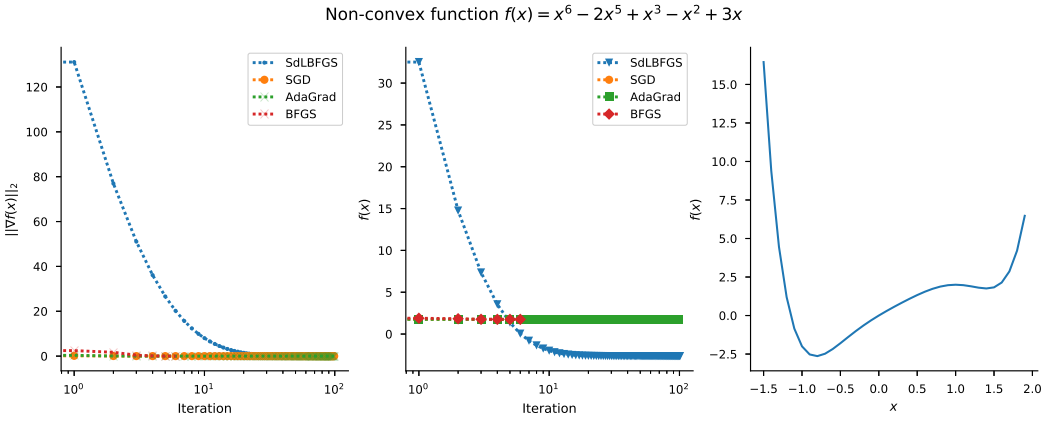

Figure 3: SdLBFGS compared with traditional BFGS together with AdaGrad and Stochastic Gradient Descent on the problem $f(x) = x^6 - 2x^5 + x^3 - x^2 + 3x$, shown on the right. Each algorithm is initialized near the local minimum at around $x = 1.6$. We see that SGD, AdaGrad, and BFGS quickly converge to the local minimum, whereas SdLBFGS converges to the global minimum at around $x = -0.8$ more slowly.

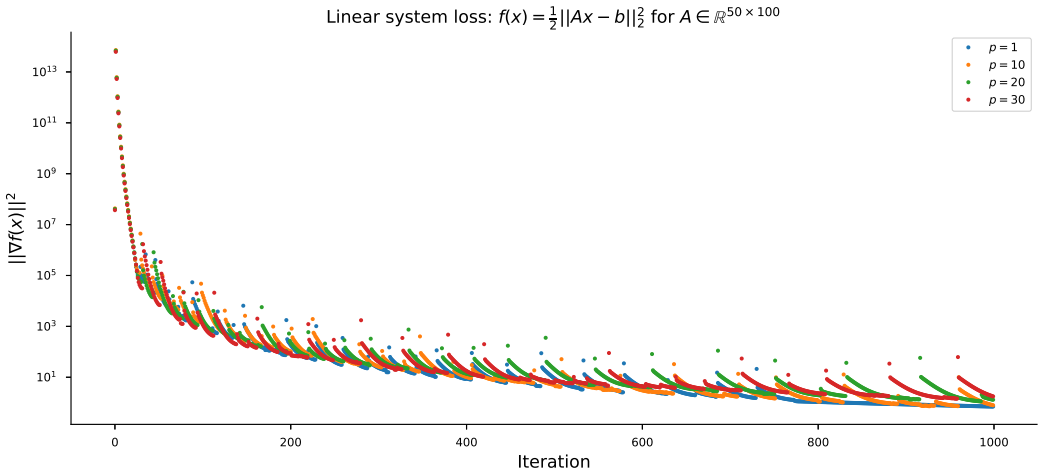

Figure 4: SdLBFGS performance on the solving the overdetermined linear system $Ax = b$ with $A \in \mathbb{R}^{50 \times 100}$ with convex smooth loss function $\|Ax - b\|_2^2$. We sweep different memory capacity values $p \in \{1, 10, 20, 30\}$ and observe that convergence is approximately equivalent between configurations.

5 Discussion

In this work we sought to develop an implementation of SdLBFGS which can be used to solve large scale machine learning problems using `TensorFlow`. By this measure, we have reported a negative result. Our preliminary results suggest that our implementation, broadly speaking, works satisfactorily on small problems, but our attempts at using the `SQNOptimizer` on problems of the size of a typical neural network have been proven infeasible.

Might we say with certainty, then, that the external optimizer approach taken in this paper is simply insufficient for solving problems that are typical for the `TensorFlow` framework? Much analysis of our implementation remains before we may reject with certainty our approach. The fundamental obstacles we encountered were sensitivities to algorithm parameter choices and overall speed in

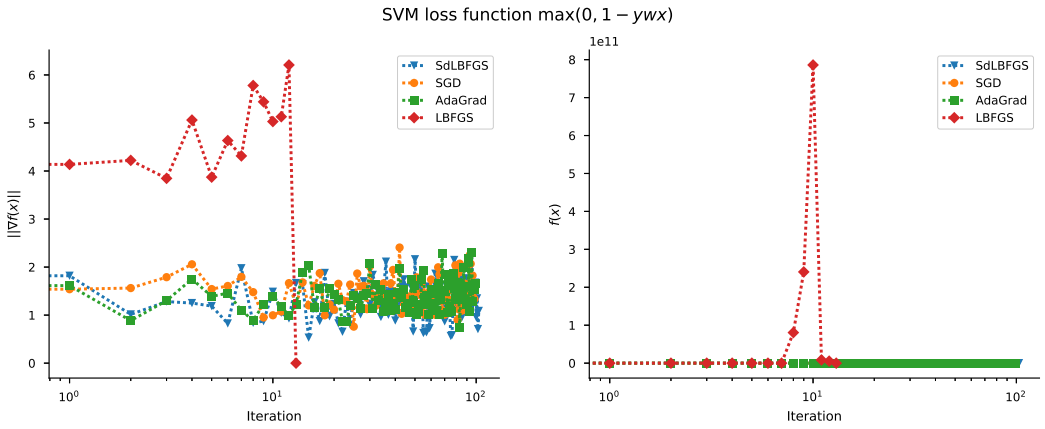

Figure 5: SdLBFGS performance on the Higgs boson dataset using an SVM classifier with a hinge loss function, compared with other minimization algorithms. Though no theoretical guarantees are made about the performance of SdBFGS, we observe that it behaves quite well with respect to its peers. The outlier performance is the standard L-BFGS algorithm, which seems to behave poorly on this problem.

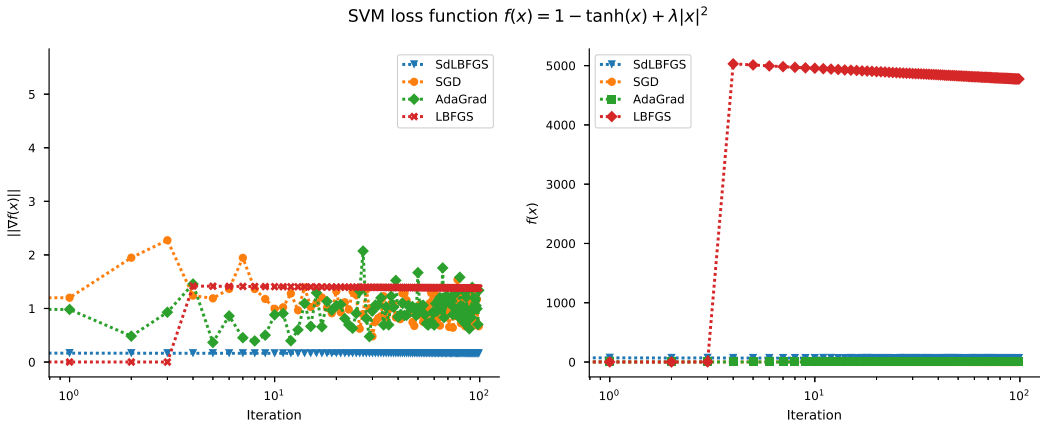

Figure 6: SdLBFGS performance on the Higgs boson dataset using an SVM classifier with a smooth regularized function, compared with other minimization algorithms. Unlike in Figure 5, we do have theoretical convergence guarantees for this $f(x)$. We also observe that it behaves quite well with respect to its peers. The outlier performance is again the standard L-BFGS algorithm, which seems to behave poorly on this problem.

computation, both of which yield many possibilities for future work. In the former, there may exist choices of step size α_k which perform sufficiently admirably for this application, or the trade off frontier between memory capacity and computational feasibility may yet become apparent.

In the latter, many directions exist through which the performance of our native Python and `NumPy` implementation might improve significantly. Having identified and dealt with potential bottlenecks around the main body of the SdLBFGS update, the judicious introduction of performance optimizing tools could bring the runtime cost of the implementation into relative feasibility. Such tools exist inside the Python ecosystem, from JIT-enhancing through Numba to static typing optimizations with Cython. In many cases significant speedups may be found by merely modifying the underlying data structures or by adopting lazier evaluation techniques.

The central point is that we cannot rule out the `ExternalOptimizerInterface` approach until the exploration of these potential improvements has sufficiently resolved the questions raised above. Further effort is needed to resolve this partial result.

References

1. Boyd, S. & Vandenberghe, L. *Convex Optimization* ISBN: 0521833787 (Cambridge University Press, New York, NY, USA, 2004).
2. Oliphant, T. E. *A guide to NumPy* (2006).
3. Martín Abadi *et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems* Software available from tensorflow.org. 2015. `https://www.tensorflow.org/`.
4. Wang, X., Ma, S., Goldfarb, D. & Liu, W. Stochastic Quasi-Newton Methods for Nonconvex Stochastic Optimization. *SIAM Journal on Optimization* **27,** 927–956 (2017).