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 do some shit and it doesn't work.

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]

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

[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.

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

1. Review main results of the Wang, Ma, Goldfarb, Liu paper [4]
2. State the SdLBFGS algortihm.
3. Mention some details about how the algorithm translates to code.

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 $\mathrm{SQN}(x_1, \langle m_k \rangle_{k=1}^{\infty}, \langle \alpha_k \rangle_{k=1}^{\infty}, p)$
    data $\leftarrow$ *nil*()
    for $k = 1, 2, \ldots$ **do**
        $g_k \leftarrow \frac{1}{m_k} \sum_{i=1}^{m_k} g(x_k, \varepsilon_{k,i})$
        $\Delta x_k, s_{k-1}, \bar{y}_{k-1}, \rho_{k-1} \leftarrow \mathrm{SDLBFGS}(g_k, {}^*(\text{data}^\top))$ ▷ 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.

1: **function** $\mathrm{SDLBFGS}(g_{k-1}, \langle s_j \rangle_{j=k-p}^{k-2}, \langle \bar{y}_j \rangle_{j=k-p}^{k-2}, \langle \rho_j \rangle_{j=k-p}^{k-2})$
2:     mem $\leftarrow$ *min*$(p, k-1)$
3:     $s_{k-1} \leftarrow x_k - x_{k-1}$
4:     $y_{k-1} \leftarrow \frac{1}{m_{k-1}} \sum_{i=1}^{m_{k-1}} [g(x_k, \xi_{k-1,i}) - g(x_{k-1}, \xi_{k-1,i})]$
5:     $\gamma_k \leftarrow \mathit{max}\left(\frac{{y_{k-1}}^\top y_{k-1}}{s_{k-1}^\top y_{k-1}}, \delta\right) \geq \delta$
6:     $\theta \leftarrow \mathit{max}\left(\frac{3}{4} \frac{{\gamma_k}^{-1} {s_{k-1}}^\top s_{k-1}}{{\gamma_k}^{-1} {s_{k-1}}^\top s_{k-1} - {s_{k-1}}^\top y_{k-1}}, 1\right)$ ▷ θ preserves positive semidefiniteness.
7:     $\bar{y}_{k-1} \leftarrow \theta y_{k-1} + (1 - \theta {\gamma_k}^{-1} s_{k-1})$
8:     $\rho_{k-1} \leftarrow ({s_{k-1}}^\top \bar{y}_{k-1})^{-1}$
9:     **for** $i = 0, \ldots,$ mem $- 1$ **do**
10:         $\mu_i \leftarrow \rho_{k-i-1} {u_i}^\top s_{k-i-1}$
11:         $u_{i+1} \leftarrow u_i - \mu_i \bar{y}_{k-i-1}$
12:     **end for**
13:     $v_0 \leftarrow {\gamma_k}^{-1} u_{\text{mem}}$
14:     **for** $i = 0, \ldots,$ mem $- 1$ **do**
15:         $\nu_i \leftarrow \rho_{k-\text{mem}+i} {v_i}^\top \bar{y}_{k-\text{mem}+i}$
16:         $v_{i+1} \leftarrow v_i + (\mu_{\text{mem}-i-1} - \nu_i) s_{k-\text{mem}+i}$
17:     **end for**
18:     **return** $v_p, s_{k-1}, \bar{y}_{k-1}, \rho_{k-1}$
19: **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.

```
# ... setup
optimizer = ExternalOptimizer(objective_function, **kwargs)
# ... setup
with tf.Session() as session:
    # ... training setup
    optimizer.minimize(session, **training_kwargs)
```

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.

5 Discussion

1. Explaining why the basic project — implementing a useful practical TensorFlow optimzier — failed.
2. Takeaways from the results.
3. Future work.

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).