Exploring a Stochastic Quasi-Newton Method

Ahmed Bou-Rabee

June 4, 2014

Abstract

In this project we present an open-source Python implementation of a stochastic quasi-Newton method [BHNY], test it, present the results of a modification, and discuss possible extensions.

1 Introduction

Stochastic optimization methods are useful when the problem under consideration is large. For example, in modern machine learning settings we often have a massive amount of data from which we would like to construct a model. Steepest descent is memory and computationally intensive as it requires the evaluation of the gradient on the entire set of data in each iteration. In batch stochastic gradient descent, we approximate the gradient by evaluating it on a small subset of the data, which is much less expensive.

However, both of these ignore curvature. In many cases, using the curvature as well as the gradient yields much faster convergence. However, finding the inverse of the Hessian in high dimensions is a costly operation. Many algorithms therefore approximate the inverse Hessian using iterative methods which analyze a succession of vectors. Quasi-Newton methods do not compute the Hessian explicitly and instead approximate it by analyzing a succession of gradient vectors. Each iteration is then

\[ x^{k+1} = x^k - \alpha^k H_k \nabla F(x^k), \]

where \( \alpha^k \) is the step size, \( H_k \) is an approximation of the inverse Hessian matrix, and \( x^k \) is the optimization variable. There are several flavors of quasi-Newton methods. One of the most well known is the BFGS algorithm [JS99].

The algorithm presented in [BHNY] is a stochastic quasi-Newton method. The optimization problem that we solve is the unconstrained stochastic convex optimization problem,

\[ \text{minimize } \mathbb{E}[f(x; \omega)], \]

where \( x \in \mathbb{R}^p \) is the optimization variable, \( \omega \) is a random variable, and \( f \) is a convex real-valued function that is continuously twice-differentiable.
2 Algorithm

The stochastic quasi-Newton method accepts parameters: \( x^0 \in \mathbb{R}^p \), an initial guess of the optimization variable; \( M \in \mathbb{N} \), the memory cap of L-BFGS; \( L \in \mathbb{N} \), the frequency with which we compute the curvature pairs; and \( K \in \mathbb{N} \), the number of iterations.

```plaintext
given initial parameters \( x_0, M, L, K \)
\( \bar{x}_j := x_0, \bar{x}_i := 0, s_t := 0, y_t := 0 \)
for \( k = 1, \ldots, K - 1 \) do
  Query the gradient oracle to compute the noisy gradient \( \hat{\nabla} F(x^k) \)
  \( \bar{x}_i := \bar{x}_i + x^k \)
  if \( k < 2L \), then update \( x^{k+1} := x^k - \frac{1}{k} \hat{\nabla} F(x^k) \)
  else \( [x^{k+1} H_t] = \) Limited Memory BFGS \((x^k, H_t)\)
  if \( \text{mod}(k, L) = 0 \) then update curvature pairs
  Calculate the noisy Hessian-vector product \( \bar{y}_t := \hat{\nabla}^2 F(\bar{x}_i)(\bar{x}_i - \bar{x}_j) \),
  \( \bar{w}_i := \bar{w}_i / L, \bar{s}_t := \bar{x}_i - \bar{x}_j, \bar{x}_j := \bar{x}_i, \bar{x}_i = 0 \)
return \( x^K \)
```

3 Example

An implementation of the algorithm has been completed in Python and is available at https://github.com/nitromannitol/stochastic-quasi-newton/. In this section we present the results of running stochastic quasi-Newton on logistic regression. This example was chosen in a desire to reproduce the results of the original paper. In this case, \( \omega \) is a random instance consisting of a data and label pair, reducing our minimization problem to

\[
\text{maximize } \frac{1}{n} \sum_{i=1}^{N} f(A_i, b_i, x),
\]

where \( f \) is the log-likelihood and \( A \in \mathbb{R}^{p \times N} \) is a matrix of data samples with labels \( b \in \{0, 1\}^N \).

3.1 Code Demonstration

First, we read in the data. Here, each row in the dataset represents a molecule with the first column describing a biological response. If the molecule exhibited the response, the output is 1, otherwise it is 0. The features of the data set are molecular descriptors, for instance, size or shape. In this dataset there are 3751 sample and 1776 features.

```python
data = numpy.matrix(numpy.genfromtxt(open('train.csv','r'), delimiter=','))[1:]
numFeatures = data.shape[1] - 1
numSamples = data.shape[0]
```
We must hard-code three functions: the objective, gradient oracle, and then the Hessian-vector oracle, to pass into the algorithm. These functions have a fixed standard input/output form. Please note that the corresponding functions in the source code differ slightly from below in an attempt to make them robust against numerical issues.

```python
def logLikelihood(w, dataSample):
    label = dataSample[0]
    x = dataSample[1:]
    z = w.T*x
    sigmoid = 1/(1 + math.exp(-z))
    return label*math.log(sigmoid) + (1.0-label)*math.log(1.0 - sigmoid)

def logisticGradient(w, dataSample):
    label = dataSample[0].item()
    x = dataSample[1:]
    z = w.T*x
    sigmoid = 1/(1 + math.exp(-z))
    return((label-sigmoid))*x

def logisticHessianVec(w, dataSample, s):
    x = dataSample[1:]
    z = w.T*x
    sigmoid = 1/(1 + math.exp(-z))
    return sigmoid*(1-sigmoid)*(x.T*s)*x
```

We form an expression object from these functions and the data and then form a problem from the expression.

```python
exp = Expression(logisticObjective, logisticGradient, logisticHessianVec,
                 data, numFeatures)
prob = Problem(exp)
```

Using the problem object we can minimize the given expression. We can run both stochastic gradient descent and stochastic quasi-Newton on the problem object. Here, we run stochastic quasi-Newton:

```python
[optval, x_final] = prob.sqnsolve(K=100, gradientBatchSize=50,
                                  hessianBatchSize=300, M=10, L=30, K=1000)
```

where \(x_{\text{final}}\) is the value of \(x\) at the optimal point and optval is the value of the objective at that point.

### 3.2 Results

Our implementation has the ability to produce extra outputs that will allow us to examine the performance of the at the algorithm. For instance, if we call
prob.sqnsolve( verbose = True)

We can examine the average iteration speed and look at the convergence plot of the algorithm. Figure 1 compares the convergence of stochastic gradient descent (SGD) with gradientBatchSize = 50 and stochastic quasi-Newton (SQN) with gradientBatchSize = 10, hessianBatchSize = 300, M = 10, and L = 20.

![Figure 1: Poor Performance of SQN](image)

According to Table 1, it seems that a larger value of $L$ and a smaller hessianBatchSize leads to better performance. However, larger values of $L$ means that the algorithm converges slower, even though each iteration is faster. This seems to imply that while the Hessian updates made by the algorithm are usually strong pushes in the right direction, they might cause the current point to overshoot the optimal value. Using this reasoning, we introduce a hyper-parameter $Z$, where in each iteration $L$ is incremented by 1 with frequency $Z$. From testing this, we find that this heuristic improves the performance of the algorithm in most cases. For instance, when running SQN with gradientBatchSize = 30, hessianBatchSize =
Table 1: Performance of SQN with various parameters

<table>
<thead>
<tr>
<th>Grad B.S.</th>
<th>Hess B.S.</th>
<th>M</th>
<th>L</th>
<th>Rel. Objective</th>
<th>NumIters</th>
<th>Rel. Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>300</td>
<td>10</td>
<td>20</td>
<td>1.03</td>
<td>120</td>
<td>0.2</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>10</td>
<td>20</td>
<td>1.02</td>
<td>169</td>
<td>0.62</td>
</tr>
<tr>
<td>500</td>
<td>300</td>
<td>10</td>
<td>20</td>
<td>1.03</td>
<td>130</td>
<td>5</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>10</td>
<td>20</td>
<td>1.0</td>
<td>169</td>
<td>0.5</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>10</td>
<td>20</td>
<td>0.9</td>
<td>193</td>
<td>0.6</td>
</tr>
<tr>
<td>30</td>
<td>1000</td>
<td>10</td>
<td>20</td>
<td>1.2</td>
<td>142</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>10</td>
<td>20</td>
<td>1.0</td>
<td>130</td>
<td>0.58</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>20</td>
<td>20</td>
<td>0.99</td>
<td>152</td>
<td>0.47</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>100</td>
<td>20</td>
<td>1.0</td>
<td>229</td>
<td>0.9</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>10</td>
<td>1</td>
<td>0.79</td>
<td>201</td>
<td>2.2</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>10</td>
<td>30</td>
<td>0.85</td>
<td>195</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>10</td>
<td>85</td>
<td>1.00</td>
<td>321</td>
<td>0.49</td>
</tr>
</tbody>
</table>

300, L = 30, M = 10, and Z = 10, against the SGD from before, we get the convergence graph in Figure 2. We see that in this run, SQN converges quite fast to the optimal value.

Figure 2: Performance of SQN with hyper-parameter Z=10

However, even with the addition of the hyper-parameter, the algorithm still does not always converge.
4 Final Remarks

4.1 Constrained Optimization

The method as it currently stands only solves unconstrained minimization problems. Once the algorithm itself has been improved, we can apply it to constrained minimization problems with the following natural extension: Pass in a set of constraints to the algorithm. During each iteration of stochastic quasi-Newton, project the current point onto a uniformly random selected subset of the constraints. Terminate the algorithm once we have performed a certain number of iterations and the point satisfies the constraints.

When this algorithm is performed according to [AS93], the expected number of iterations before the point is fully in the constraints is $O(d \log n)$ where $d$ is the number of variables and $n$ is the number of constraints. However, the effect of performing these projections during SQN is unknown. Also, the asymptotic result in [AS93] does not hold when the number of constraints is small. Therefore, in most cases, a non-deterministic projection method would perform better than the randomized projection method.

4.2 Limitations

The current version of the algorithm is not fit for use by non-experts. While the results of introducing hyper-parameter $Z$ are promising, there has not been enough testing done to validate its existence. Also, as SQN is very sensitive to parameter choice it is often very time-consuming to come up with the right choice of parameters for a particular problem. In some ways this defeats the purpose of having a software package, since most users will not be able to fully utilize SQN. One possible way to mitigate this is to automate parameter choices based on the input data. However, this is quite slow, and must be designed carefully.

Also, a weakness of the current implementation is that it is very cumbersome to hand-code the objective, gradient, and stochastic sub-gradient. Existing versions of stochastic gradient descent packages provide built in choices of loss functions, which makes the coding much easier, but lacks customizability. A compromise between the two would be to provide both a collection of built-in loss functions along with the ability to pass in custom functions. And, once the algorithm has converged to its final form, introducing regularization would make SQN much more robust.

References
