Appendix: Supplementary Material

1 Background

In the following, we provide a brief primer on line search and trust region methods, as well as on quasi-Newton methods and their limited memory variants. Further details may be found in [1].

1.1 Line Search and Trust Region Methods

In any optimization algorithm, there are two main ways of moving from the current point $x_k$ to a new iterate $x_{k+1}$. One of them is line search. In it, the algorithm picks a descent direction $p_k$ and searches along this direction from the current iterate $x_k$ for a new iterate with a lower function value. The distance $\alpha$ to move along $p_k$ can be found by solving the following one-dimensional minimization problem:

$$
\min_{\alpha>0} f(x_k + \alpha p_k)
$$

Instead of an exact minimization which may be expensive, the line search algorithm generates a limited number of trial step lengths until it finds one that generates a sufficient decrease in function value. At the new point, the process of computing the descent direction and step length is repeated.

The other way is to use a trust region method. In a trust region method, the information about $f$ is used to construct a model function $m_k$, which is supposed to approximate $f$ near the current point $x_k$. Since the model $m_k$ may not approximate $f$ well when $x$ is far from $x_k$, the search for a minimizer of $m_k$ is restricted to some trust region within a radius $\Delta_k$ around $x_k$. The candidate step $p$ approximately solves the following sub-problem:

$$
\min_{p: \|p\| \leq \Delta_k} m_k(x_k + p),
$$

If the candidate solution does not produce a sufficient decrease in $f$, the trust region is considered too large for the model function to approximate $f$ well. So we shrink the trust region and re-solve. Essentially, the line search and trust region approaches differ in the order in which they choose the direction and magnitude of the move to the next iterate. In line search, the descent direction $p_k$ is fixed first, and then the step length $\alpha_k$ to be taken along that direction is computed. In trust region, a maximum distance equal to the trust-region radius $\Delta_k$ is first set, and then a direction is determined within this radius, that achieves the best improvement in the objective value. If such a direction does not yield sufficient improvement, the model function is determined to be a poor approximation to the function, and the trust-region radius $\Delta_k$ is reduced until the approximation is deemed good enough. Conversely, as long as the model function appears to approximate the objective function well, the trust region radius is increased until the approximation is not good enough.

1.2 Limited memory quasi-Newton Methods

Quasi-Newton methods are a useful alternative to Newton’s method in that they do not require computation of the exact Hessian, and yet still attain good convergence. In place of the true Hessian $\nabla^2 f_k$, they use an approximation $B_k$, which is updated after each step based on information gained during the step. At each step, the new Hessian approximation $B_{k+1}$ is required to satisfy the following condition, known as the secant equation:

$$
B_{k+1} s_k = y_k
$$
where
\[ s_k = x_{k+1} - x_k, y_k = \nabla f_{k+1} - \nabla f_k \]

Typically, \( B_{k+1} \) is also required to be symmetric (like the exact Hessian), and the difference between successive approximations \( B_k \) and \( B_{k+1} \) is constrained to have low rank. One of the most popular formulae for updating the Hessian approximation \( B_k \) is the BFGS formula, named after its inventors, Broyden, Fletcher, Goldfarb, and Shanno, which is defined by

\[
B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}
\]

A less well known formula, particularly in the machine learning community, is the symmetric-rank-one (SR1) formula, defined by

\[
B_{k+1} = B_k + \frac{(y_k - B_k s_k)(y_k - B_k s_k)^T}{y_k^T s_k}
\]

The former update is a rank-two update, while the latter is a rank-one update. Both updates satisfy the secant equation and maintain symmetry. The BFGS update always generates positive definite approximations whenever the initial approximation \( B_0 \) is positive definite and \( s_k^T y_k > 0 \). Often, in practical implementations of quasi-Newton methods, the inverse Hessian approximation \( H_k \) is used instead of the \( B_k \), and the corresponding update formulae can be generated using the Sherman-Morrison-Woodbury matrix identity [2].

Limited-memory quasi-Newton methods are useful for solving large problems where computation of Hessian matrices is costly or when these matrices are dense. Instead of storing fully dense \( n \times n \) approximations, these methods save only a few vectors of length \( n \) that capture the approximations. Despite these modest storage requirements, they often converge well. The most popular limited memory quasi-Newton method is L-BFGS, which uses curvature information from only the most recent iterations to construct the inverse Hessian approximation. Curvature information from earlier iterations, which is less likely to be useful to modeling the actual behavior of the Hessian at the current iteration, is discarded in order to save memory.

Limited-memory quasi-Newton approximations can be used with line search or trust region methods. As described in [3], we can derive efficient limited memory implementations of several quasi-Newton update formulae, and their inverses. For instance, if \( k \) updates are made to the symmetric matrix \( B_0 \) using the vector pairs \( \{s_i, y_i\}_{i=0}^{k-1} \) and the SR1 formula, the resulting matrix \( B_k \) can be expressed as [1]

\[
B_k = B_0 + (Y_k - B_0 S_k)(D_k + L_k + L_k^T - S_k^T B_0 S_k)^{-1}(Y_k - B_0 S_k)^T
\]

where \( S_k, Y_k, D_k, \) and \( L_k \) are defined as follows:

\[
S_k = [s_0, \ldots, s_{k-1}], \text{and } Y_k = [y_0, \ldots, y_{k-1}]
\]

\[
(L_k)_{i,j} = \begin{cases} 
  s_{i-1}^T y_{j-1} & \text{if } i > j \\
  0 & \text{otherwise}
\end{cases}
\]

\[
D_k = \text{diag}[s_0^T y_0, \ldots, s_{k-1}^T y_{k-1}]
\]

The self-duality of the SR1 method [1] allows the inverse formula \( H_k \) to be obtained simply by replacing \( B, s, \) and \( y \) by \( H, y, \) and \( s \), respectively, using standard matrix identities. Thus, limited-memory SR1 methods can be derived exactly like in the case of the BFGS method.

2 Pseudocode

Algorithm [1] provides the pseudocode for L-SR1 with trust region method, while Algorithm [2] provides the pseudocode for L-SR1 with line search.
Algorithm 1 L-SR1 with Trust Region Method

Require: \( S_k = [s_0, \cdots, s_k-1], Y_k = [y_0, \cdots, y_k-1] \), starting point \( x_0 \in \mathbb{R}^n \), limited memory size \( m < n \), initial Hessian approximation \( B_0 \) (a diagonal matrix, typically \( \gamma I_n \), \( \gamma \neq 0 \)), initial trust-region radius \( \Delta_0 = \|\nabla f(x_0)\|_2 \), convergence tolerance \( t > 0 \), maximum iterations \( K \), parameters \( \eta \in (0, 10^{-3}) \), \( r \in (0, 1) \), and column dimension \( \text{colDim} \);

1: \( k \leftarrow 0 \)
2: while \( k < K \) and \( \|\nabla f(x_k)\|_2 > t \) and \( \|s_k\|_2 > t \) do
3: \( \text{if } k = 0 \) or \( S_k, \text{colDim} = 0 \) then
4: \( s_k \leftarrow -\nabla f(x_k) \)
5: \( B_k \leftarrow B_0 \)
6: else
7: \( s_k \leftarrow \text{TrustRegionMethod}(\Psi_k, M_k^{-1}, \nabla f(x_k), \Delta_k, \gamma, B_0) \) (Solve the trust-region subproblem)
8: \( B_k s_k \leftarrow B_0 s_k + \Psi_k M_k (\Psi_k^T s_k) \)
9: end if
10: \( \text{pred} \leftarrow - (\nabla f(x_k)^T s_k + \frac{1}{2} s_k^T B_k s_k) \) (predicted reduction)
11: \( \text{ared} \leftarrow f(x_k) - f(x_k + s_k) \) (actual reduction)
12: \( y_k \leftarrow \nabla f(x_k + s_k) - \nabla f(x_k) \)
13: if \( \text{ared/pred} > \eta \) then
14: \( x_{k+1} \leftarrow x_k + s_k \)
15: else
16: \( x_{k+1} \leftarrow x_k \)
17: end if
18: \( \Delta_{k+1} \leftarrow \Delta_k \)
19: if \( \text{ared/pred} > u \) then \( u = 0.75 \) by default
20: \( \text{if } \|s_k\| > \rho \Delta_k \) then \( \rho = 0.8 \) by default
21: \( \Delta_{k+1} \leftarrow 2 \Delta_k \)
22: end if
23: else if \( \text{ared/pred} < l \) then \( l = 0.1 \) by default
24: \( \Delta_{k+1} \leftarrow 0.5 \Delta_k \)
25: end if
26: \( \text{if } \|s_k^T (y_k - B_k s_k)\| \geq r \|s_k\| \|y_k - B_k s_k\| \) then
27: \( S_{k+1} \leftarrow [S_k, s_k], Y_{k+1} \leftarrow [Y_k, y_k] \)
28: if \( S_{k+1}, \text{colDim} > m \) then
29: \( S_{k+1} \leftarrow S_{k+1}[i, 2 : m+1], Y_{k+1} \leftarrow Y_{k+1}[i, 2 : m+1] \)
30: end if
31: while \( S_{k+1}, \text{colDim} > 0 \) do
32: \( \Psi_{k+1} \leftarrow Y_{k+1} - B_0 S_{k+1} \)
33: \( M_{k+1}^{-1} \leftarrow S_{k+1}^T Y_{k+1} - S_{k+1}^T B_0 S_{k+1} \)
34: \( \text{if } \Psi_{k+1}^T \Psi_{k+1} > 0 \) and \( |M_{k+1}^{-1}| \neq 0 \) then
35: \( \text{break} \)
36: else
37: Remove the first columns of \( S_{k+1} \) and \( Y_{k+1} \)
38: end if
39: end while
40: end if
41: \( k \leftarrow k + 1 \)
42: end while
Algorithm 2 L-SR1 with Line Search

Require: $S_k = [s_0, \cdots, s_{k-1}]$, $Y_k = [y_0, \cdots, y_{k-1}]$, starting point $x_0 \in \mathbb{R}^n$, limited memory size $m < n$, initial inverse Hessian approximation $H_0$ (a diagonal matrix, typically $I_n$), initial step length $\lambda_0 = 1$, convergence tolerance $t > 0$, maximum iterations $K$, $r \in (0,1)$, and column dimension $\text{colDim}$:

1: $k \leftarrow 0$
2: while $k < K$ and ($\|\nabla f(x_k)\|_2 > t$ or $\|s_k\|_2 > t$) do
3: if $k = 0$ or $S_k.\text{colDim} = 0$ then
4: $d_k \leftarrow -\nabla f(x_k)$
5: $H_k \leftarrow H_0$
6: else
7: $d_k \leftarrow -H_0 \nabla f(x_k) - \Psi_k M_k (\Psi_k^T \nabla f(x_k))$
8: end if
9: if $k > 0$ then
10: $\lambda_0 = \min\{1, 2\lambda_{k-1}\}$
11: end if
12: $\lambda_k \leftarrow \text{computeStepLength}(f, x_k, d_k, \lambda_0)$ (perform line search)
13: $s_k \leftarrow \lambda_k d_k$
14: $x_{k+1} \leftarrow x_k + s_k$
15: $y_k \leftarrow \nabla f(x_{k+1}) - \nabla f(x_k)$
16: if $k > 0$ and $S_k.\text{colDim} > 0$ then
17: $H_k y_k \leftarrow H_0 y_k + \Psi_k M_k (\Psi_k^T y_k)$
18: end if
19: if $\|y_k^T (s_k - H_k y_k)\| \geq r \|y_k\| \|s_k - H_k y_k\|$ then
20: $S_{k+1} \leftarrow [S_k, s_k]$, $Y_{k+1} \leftarrow [Y_k, y_k]$
21: if $S_{k+1}.\text{colDim} > m$ then
22: $S_{k+1} \leftarrow S_{k+1}[2 : m + 1]$, $Y_{k+1} \leftarrow Y_{k+1}[2 : m + 1]$
23: end if
24: while $S_{k+1}.\text{colDim} > 0$ do
25: $\Psi_{k+1} \leftarrow S_{k+1} - H_0 Y_{k+1}$
26: $M^{-1}_{k+1} \leftarrow Y_{k+1}^T S_{k+1} - Y_{k+1}^T H_0 Y_{k+1}$
27: if $\|M^{-1}_{k+1}\| \neq 0$ then
28: break
29: else
30: Remove the first columns of $S_{k+1}$ and $Y_{k+1}$
31: end if
32: end while
33: end if
34: $k \leftarrow k + 1$
35: end while
3 Network Architectures and Hyperparameter Settings

3.1 MNIST

The layers of the LeNet5 architecture used, are described below, in order. All the batch normalization layers were removed, in the ‘without batch normalization’ case.

- Convolutional Layer - filter size $5 \times 5$, 20 feature maps, stride 1, padding 0, and a ReLU activation function with bias 0 and Gaussian noise with mean 0 and standard deviation 0.1
- Spatial Batch Normalization Layer
- Convolutional Layer - filter size $5 \times 5$, 50 feature maps, stride 1, padding 0, and a ReLU activation function with bias 0 and Gaussian noise with mean 0 and standard deviation 0.1
- Spatial Batch Normalization Layer
- Max Pooling Layer - filter size 2
- Fully Connected Layer - 500 hidden units, and a tangent hyperbolic activation function
- Spatial Batch Normalization Layer
- Outer Output Layer - 10 outputs and output standard deviation of 0.1

Additionally, the network was trained with $L2$ regularization with parameter 0.0001. Training loss was measured as softmax cross entropy, while test loss was measured as misclassification error rate. In the case of the first order methods, learning rate was set to 0.003 where needed, and momentum was set to 0.9, where needed. AdaDelta did not take any parameters.

3.2 CIFAR10

The layers of the architecture used, are described below, in order. All the batch normalization layers were removed, in the ‘without batch normalization’ case.

- Convolutional Layer - filter size $5 \times 5$, 32 feature maps, stride 1, padding 2, and a ReLU activation function with bias 0 and Gaussian noise with mean 0 and standard deviation 0.01
- Spatial Batch Normalization Layer
- Max Pooling Layer - filter size 2
- Activation Layer - ReLU activation function with bias 0 and Gaussian noise with mean 0 and standard deviation 0.1
- Convolutional Layer - filter size $5 \times 5$, 32 feature maps, stride 1, padding 2, and a ReLU activation function with bias 0 and Gaussian noise with mean 0 and standard deviation 0.01
- Spatial Batch Normalization Layer
- Max Pooling Layer - filter size 2
- Convolutional Layer - filter size $5 \times 5$, 64 feature maps, stride 1, padding 2, and a ReLU activation function with bias 0 and Gaussian noise with mean 0 and standard deviation 0.01
- Spatial Batch Normalization Layer
- Max Pooling Layer - filter size 2
- Fully Connected Layer - 64 hidden units, and a ReLU activation function with bias 0 and Gaussian noise with mean 0 and standard deviation 0.1
- Spatial Batch Normalization Layer
- Outer Output Layer - 10 outputs and output standard deviation of 0.1

Additionally, the network was trained with $L2$ regularization with parameter 0.0001. Training loss was measured as softmax cross entropy, while test loss was measured as misclassification error rate. In the case of the first order methods, learning rate was set to 0.01 where needed, and momentum was set to 0.9, where needed. AdaDelta did not take any parameters.
References

