Incremental Gradient Methods

This project is aimed to be a way for us to better understand and thinker with the recent advances in Stochastic Gradient Descent algorithms, specifically the recent SAGA [1]. This method is similar in spirit to the previously developed SAG, SDCA, MISO and SVRG. This class of algorithms have been developed to solve problems of the form

$$\min_{x \in \mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} f_i(x) + h(x), \tag{1}$$

where each f_i is convex and has Libschitz continuous derivatives with constant L or is strongly convex with constant μ; and h is a convex but potentially non-differentiable function (his proximal operator is however easy to compute). While computing the full gradient would be prohibitive due to large d and n, these iterative stochastic algorithms reduce the computational cost of optimization by only computing the gradient of a subset of the functions f_i at each step.

Many machine learning problems can be cast in (1), such as (constrained) Least-Square or Logistic Regressions with ℓ_1 or ℓ_2 regularization; where x would represent the model parameters, f_i the data fidelity term applied to a particular sample i, and h a regularization or indicator function of a convex set.

In this project we explored two approaches to improve SAGA towards two different goals:

- A **memory-efficient SAGA**, by way of computing and storing gradients over mini-batches instead of single samples, as is done for the infamous Stochastic Gradient Descent. It turns out that this approach is also faster than the original algorithm as it exploits the vectorial capabilities of modern processing units.
- A **time-efficient SAGA**, by way of distributing the gradient computation over many CPU cores while fusing the results and updating the model parameters on the master.

1 SAGA algorithm

The algorithm starts with some known initial vector $x^0 \in \mathbb{R}^d$ and known derivatives $f_i'(\phi_i^0) \in \mathbb{R}^d$ with $\phi_i^0 = x^0$ for each i. These derivatives are stored in a table data-structure of length n, or alternatively a $n \times d$ matrix. It uses a step size of γ[1] and, given the value of x^k and of each $f_i'(\phi_i^k)$ at the end of iteration k, makes the following updates for iteration $k+1$:

1. Pick a j uniformly at random.
2. Take $\phi_j^{k+1} = x^k$, and store $f_j'(\phi_j^{k+1})$ in the table. All other entries in the table remain unchanged. The quantity ϕ_j^{k+1} is not explicitly stored.
3. Update x using $f_j'(\phi_j^{k+1})$, $f_j'(\phi_j^k)$ and the table average:

$$w^{k+1} = x^k - \gamma \left[f_j'(\phi_j^{k+1}) - f_j'(\phi_j^k) + \frac{1}{n} \sum_{i=1}^{n} f_i'(\phi_i^k) \right], \tag{2}$$

$$x^{k+1} = \operatorname{prox}_{\gamma}^{h}(w^{k+1}).$$

As the authors of [1], we tested our algorithms by training a linear regression model (with ℓ_1 or ℓ_2 regularization) on the Million Song dataset[2]. The problem, as stated on the UCI repository[3] where the data was downloaded from, is to predict the release year of a song from audio features.

[1] The authors recommend a learning rate of $\gamma = 1/(2(\mu n + L))$ in the strongly convex case, $\gamma = 1/(3(\mu n + L))$ if the strong convexity requirement only holds on average and $\gamma = 1/(3L)$ for non-strongly convex problems.

[2] http://labrosa.ee.columbia.edu/millionsong

[3] http://archive.ics.uci.edu/ml/datasets/YearPredictionMSD

2 Mini-batch SAGA

We first form $\frac{n}{m}$ mini-batches $\{\mathcal{B}_i\}_{i=1}^{\frac{n}{m}}$ of size $|\mathcal{B}_i| = m$ and take gradients w.r.t. them, such that the gradient matrix is of size $\frac{n}{m} \times d$ instead of $n \times d$. The updates for iteration $k+1$ then becomes:

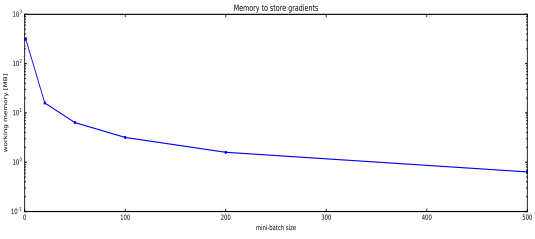
1. Pick a i uniformly at random in $[1, \frac{n}{m}]$.
2. Take $\phi_j^{k+1} = x^k \; \forall \; j \in \mathcal{B}_i$, and store $\frac{1}{m} \sum_{j \in \mathcal{B}_i} f_j'(\phi_j^{k+1})$ in the table.
3. Update $x^{k+1} = \text{prox}_\gamma^h \left\{ x^k - \gamma \left[\frac{1}{m} \sum_{j \in \mathcal{B}_i} f_j'(\phi_j^{k+1}) - \frac{1}{m} \sum_{j \in \mathcal{B}_i} f_j'(\phi_j^k) + \frac{1}{n} \sum_{i=1}^{m} \sum_{j \in \mathcal{B}_i} f_j'(\phi_j^k) \right] \right\}$.

Note that one can usually vectorize the computation of $\sum_{j \in \mathcal{B}_i} f_j'(\phi_j^k)$. E.g. for a least-square problem $\min_x \frac{1}{2}\|Ax - y\|_2^2$, the gradient is given by $A_{\mathcal{B}_i}^T A_{\mathcal{B}_i} x$, where $A_{\mathcal{B}_i}$ represents the m columns of A selected by the mini-batch $\mathcal{B}_i$. That is where the computational advantage of the mini-batch approach comes from.

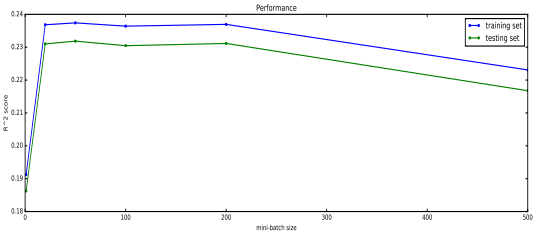
Note that mini-batches can be formed during initialization and kept intact for the whole training. They can alternatively be reformed at the beginning of each training epoch. While this is much more expensive, because the gradient matrix has to be initialized again, we found no difference in performance.

2.1 Experiments

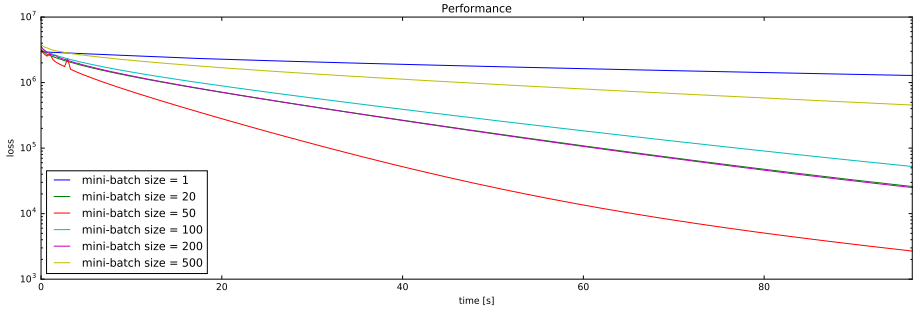
While the convergence rate is the same for all mini-batch sizes given the same learning rate, larger mini-batches need a smaller learning rate to avoid divergence. We therefore ran experiments with various mini-batch sizes m; with an optimized learning rate γ for each. Figure 1 clearly shows that mini-batch SAGA can outperform regular SAGA in convergence time while requiring less memory.



(a) Working memory to store gradients.



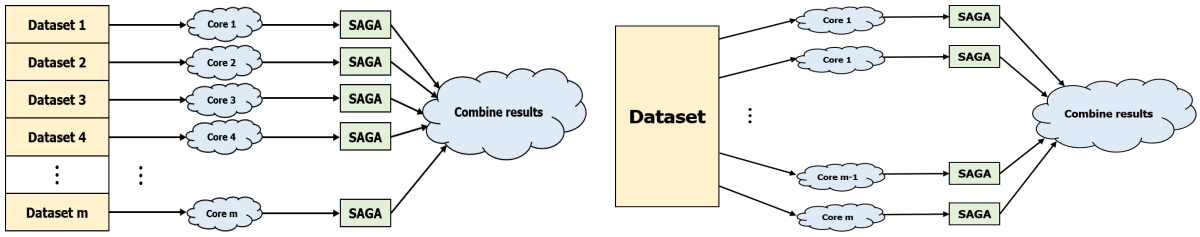
(b) R^2 score on test set after 100s training.



(c) Objective w.r.t. running time.

Figure 1: Performance evaluations of mini-batch SAGA w.r.t. various mini-batch sizes m.

3 Distributed SAGA



(a) First variant. (b) Second variant.

Figure 2: Two variants of distributed SAGA.

Variant I. A first approach, shown in Figure 2(a), is to randomly partition the n data points onto m machines with n/m local data points on each to parallelize the computation of (2). Formally speaking, at each epoch, we first randomly partition the data into m different sets. We then apply the SAGA algorithm presented in Section 1 for each dataset on m different machine. After one pass over each subset, the results are combined by updating the gradient table and candidate solution. The updates for epoch e for each $k+1$ then become:

1. Randomly partition the dataset into m non-overlapping sets $R_1, \cdots, R_m$.
2. For each machine $s \in \{1, \cdots, m\}$, pick a random sample $x^k(s)$ from R_s.
   (a) If $k = 1$ then $x^k(s) = x(e)$.
   (b) Take $\phi_j^{k+1}(s) = x^k(s)$.
   (c) Update $x^{k+1}(s) = \text{prox}_\gamma^h \left\{ x^k(s) - \gamma \left[f_j'(\phi_j^{k+1}(s)) - f_j'(\phi_j^k(s)) + \frac{1}{n} \sum_{j \in R_i} f_j'(\phi_j^k(s)) \right] \right\}$.
   (d) If $k > n/m$, break.
   (e) $k = k + 1$.
3. Merge the results with $x(e+1) = \frac{1}{m} \sum_{s=1}^m x^k(s)$.

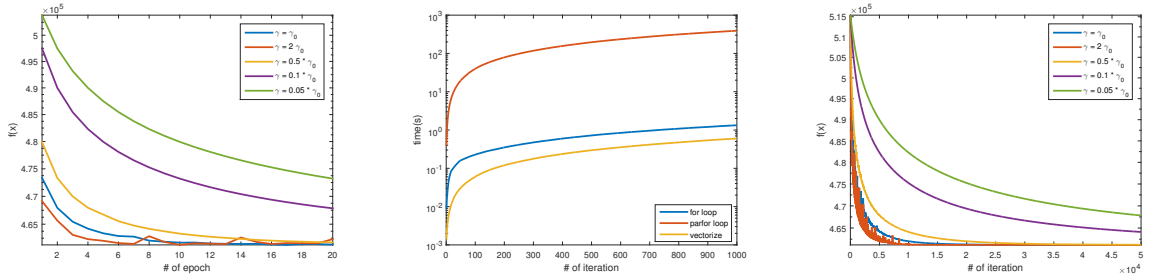
Variant II. A second approach, shown in Figure 2(b), is to randomly pick m data points and send one to each of the m machines. Each machine then computes (2) in parallel. All results are finally averaged. The updates for iteration $k+1$ then become:

1. For each machine $s \in \{1, \cdots, m\}$
   (a) Pick a random number j in $[1, n]$.
   (b) Take $\phi_j^{k+1}(s) = x^k(s)$, and store $f_j'(\phi_j^{k+1}(s))$ in the table.
   (c) Update $x^{k+1}(s) = \text{prox}_\gamma^h \left\{ x^k(s) - \gamma \left[f_j'(\phi_j^{k+1}(s)) - f_j'(\phi_j^k(s)) + \frac{1}{n} \sum_{i=1}^n f_i'(\phi_i^k(s)) \right] \right\}$.
2. Merge the results with $x^{k+1} = \frac{1}{m} \sum_{s=1}^m x^{k+1}(s)$.

3.1 Experiments

In this experiment, we set the ℓ_2 regularization factor to 5×10^{-4} while we use $n = 10^4$ training samples and $m = 8$ different machines (or CPU cores). Figure 3 shows the performance of our distributed methods. A good choice of learning rate, as for all gradient methods, is crucial as shown in Figure 3(a) and Figure 3(c). Moreover, Figure 3(b) presents the required time for

1000 iterations using different models, namely using parfor loop which is the pure distributed implementation, for loop which is a serial implementation, and vectorized implementation which relies on parallel implementation of linear algebra libraries in MATLAB. The results show an heavy communication overhead for parfor.



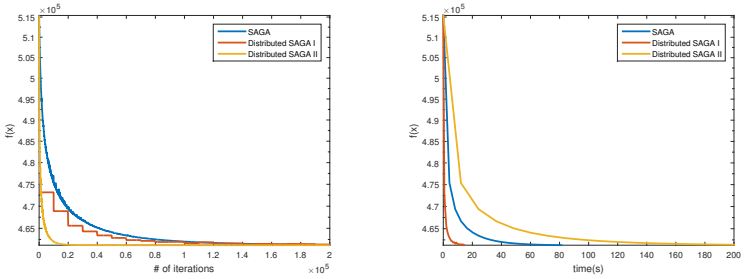
(a) Variant I: learning rate.

(b) Variant II: different parallelizations.

(c) Variant II: learning rate.

Figure 3: Performance measures of distributed SAGA.

Figure 4 compares the two proposed approaches in term of convergence rate. Variant I converges faster in running time while variant II converges faster in the number of iterations.



(a) Convergence w.r.t. the number of iterations.

(b) Convergence w.r.t. running time.

Figure 4: Comparison between two distributed SAGA algorithms and regular SAGA.

References

[1] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. "SAGA: A Fast Incremental Gradient Method With Support for Non-Strongly Convex Composite Objectives". In: *Advances in Neural Information Processing Systems 27*. 2014, pp. 1646–1654. URL: http://papers.nips.cc/paper/5258-saga-a-fast-incremental-gradient-method-with-support-for-non-strongly-convex-composite-objectives.pdf.