Block Splitting for Large-Scale Distributed Learning

Neal Parikh and Stephen Boyd

Department of Computer Science and Information Systems Laboratory, Department of Electrical Engineering, Stanford University

Distributed optimization

- we want to decompose machine learning problems by both features (variables) and training examples, in order to solve large distributed problems
- consider the problem form

$$\begin{array}{ll} \text{minimize} & l(y) + r(x) \\ \text{subject to} & y = Ax \end{array}$$

- here l is a *loss function* and r is a regularizer
- consider partitioning Ax into MN blocks as

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1N} \\ A_{21} & A_{22} & \cdots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{M1} & A_{M2} & \cdots & A_{MN} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_N \end{bmatrix}$$

  where $A_{ij} \in \mathbf{R}^{m_i \times n_j}$
- goal: handle each A_{ij} on a separate machine or process, and never transfer the A_{ij} over the network

Alternating direction method of multipliers

- consider generic constrained convex program

$$\begin{array}{ll} \text{minimize} & f(z) \\ \text{subject to} & z \in \mathcal{C} \end{array}$$

- ADMM, aka Douglas-Rachford splitting:

$$\begin{array}{lll} z^{k+1/2} & := \mathbf{prox}_f(z^k - \tilde{z}^k) & \text{// prox} \\ z^{k+1} & := \Pi_{\mathcal{C}}(z^{k+1/2} + \tilde{z}^k) & \text{// projection} \\ \tilde{z}^{k+1} & := \tilde{z}^k + z^{k+1/2} - z^{k+1} & \text{// dual update} \end{array}$$

Problem transformation

- assuming l and r are block separable, problem becomes

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^M l_i(y_i) + \sum_{j=1}^N r_j(x_j) \\ \text{subject to} & y_i = \sum_{j=1}^N A_{ij} x_j, \quad i = 1, \ldots, M \end{array}$$

- introduce additional variables:

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^M l_i(y_i) + \sum_{j=1}^N r_j(x_j) + \sum_{i=1}^M \sum_{j=1}^N I_{ij}(y_{ij}, x_{ij}) \\ \text{subject to} & x_{ij} = x_j \\ & y_i = \sum_{j=1}^M y_{ij} \end{array}$$

Block splitting algorithm

$$\begin{aligned} y_i^{k+1/2} &:= \mathbf{prox}_{l_i}(y_i^k - \tilde{y}_i^k) \\ x_j^{k+1/2} &:= \mathbf{prox}_{r_j}(x_j^k - \tilde{x}_j^k) \\ (y_{ij}^{k+1/2}, x_{ij}^{k+1/2}) &:= \Pi_{ij}(y_{ij}^k + \tilde{y}_i^k, x_j^k - \tilde{x}_{ij}^k) \\ x_j^{k+1} &:= \mathbf{avg}(x_j^{k+1/2}, \{x_{ij}^{k+1/2}\}_{i=1}^M) \\ (y_i^{k+1}, \{y_{ij}^{k+1}\}_{j=1}^N) &:= \mathbf{exch}(y_i^{k+1/2}, \{y_{ij}^{k+1/2}\}_{j=1}^N) \\ \tilde{x}_j^{k+1} &:= \tilde{x}_j^k + x_j^{k+1/2} - x_j^{k+1} \\ \tilde{y}_i^{k+1} &:= \tilde{y}_i^k + y_i^{k+1/2} - y_i^{k+1} \\ \tilde{x}_{ij}^{k+1} &:= \tilde{x}_{ij}^k + x_{ij}^{k+1/2} - x_j^{k+1} \end{aligned}$$

where $\mathbf{exch}(c, \{c_j\}_{j=1}^N)$ is given by

$$y_{ij}^{k+1} := c_j + (c - \textstyle\sum_{j=1}^N c_j)/(N+1), \; y_i^{k+1} := c - (c - \textstyle\sum_{j=1}^N c_j)/(N+1)$$

and $\mathbf{avg}$ is elementwise averaging

Graph projection

- main work in the algorithm is often projecting onto the graph of the linear operator $x \mapsto Ax$
- projection onto the graph of A_{ij} is denoted Π_{ij}
- evaluating $\Pi_{ij}(c, d)$ involves solving a quasidefinite linear system

$$\begin{bmatrix} I & A^T \\ A & -I \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} d + A^T c \\ 0 \end{bmatrix}$$

- if A is dense, this system can be solved via the updates

$$\begin{aligned} y &:= (I + AA^T)^{-1}(Ad + AA^T c) \\ x &:= d + A^T(c - y) \end{aligned}$$

  or

$$\begin{aligned} x &:= (I + A^T A)^{-1}(c + A^T d) \\ y &:= Ax. \end{aligned}$$

  depending on whether A is fat or skinny (respectively)
- otherwise, we can solve the quasidefinite system directly using a permuted LDL^T factorization (or an iterative or hybrid method)
- this is the *only* operation that touches the data, and it does not involve l_i or r_j, so this operation can be reused for any problem using the same data
- key for implementation: factorization caching (cache the factorization of the coefficient matrix in the linear system being solved, then reuse across all subsequent solves)

Numerical example

- lasso:

$$\text{minimize} \;\; (1/2)\|Ax - b\|_2^2 + \lambda \|x\|_1$$

- in standard form, set $l_i = (1/2)\|\cdot\|_2^2$, $r_j = \lambda \|\cdot\|_1$
- prox operators: elementwise scaling, soft thresholding
- algorithm:

$$\begin{aligned} y_i^{k+1/2} &:= T_\rho(y_i^k - \tilde{y}_i^k) \\ x_j^{k+1/2} &:= S_{\lambda/\rho}(x_j^k - \tilde{x}_j^k) \\ (y_{ij}^{k+1/2}, x_{ij}^{k+1/2}) &:= \Pi_{ij}(y_{ij}^k + \tilde{y}_i^k, x_j^k - \tilde{x}_{ij}^k) \\ x_j^{k+1} &:= \mathbf{avg}(x_j^{k+1/2}, \{x_{ij}^{k+1/2}\}_{i=1}^M) \\ &\vdots \end{aligned}$$

- examples with **dense** $A_{ij} \in \mathbf{R}^{3000 \times 5000}$
  - distributed solver written in C using MPI and GSL (ATLAS)
- computation times (all times in seconds)

$M \times N$	4×2	8×5	8×10
nonzero entries	120MM	600MM	1.2B
# cores	8	40	80
factorization time	15	15	15
iteration time	0.05–0.15	0.05–0.15	0.05–0.15
# iterations	90	230	490
main loop time	10	27	60
total time	28	50	80

Conclusions and extensions

- evaluating Π_{ij} is often the majority of the work
- large speedup via factorization caching
- all communication implementable via 'allreduce' operations
  - MPI, Vowpal Wabbit (Hadoop-compatible), *etc.*
- prox operations (nonlinearities) and data handled separately
  - *e.g.*, can fit multiple models simultaneously, easily compute entire regularization path of models, do model comparison