Efficient Distributed Stochastic Dual Coordinate Ascent

Jeff Hajewski
Mingrui Liu

May 3, 2017

University of Iowa

Problem Overview

Problem Overview - The Primal Problem

Many machine learning problems can be formulated as the Regularized Finite Sum Minimization (RFSM) problem.

$$\min_{w \in \mathbb{R}^d} P(w) \tag{1}$$

where

$$P(w) = \frac{1}{n}\sum_{i=1}^{n} \phi(w^\top x_i, y_i) + \lambda g(w)$$

$$w, x_i \in \mathbb{R}^d, \text{ for } i = 1, \ldots, n$$

$$y_i \in \mathbb{R}, \text{ for } i = 1, \ldots, n$$

$$\phi(z, y) \text{ is convex in } z$$

$$g(w) \text{ is convex in } w$$

Problem Overview - The Dual Problem

Summary of the dual problem

Related Work

The two key papers influencing our work are:

- Stochastic Dual Coordinate Ascent (SDCA) [Shalev-Shwartz and Zhang, 2013]
- Distributed SDCA [Yang, 2013, Yang et al., 2013]

What is SDCA?

- **SDCA** - randomly pick a coordinate axis, find update that best improves the objectivei
- **Distributed SDCA** - randomly pick k coordinate axes, simultaneously find updates that best improve the objective (independently)

Parallelizing SDCA

Two approaches:

- Naive approach: simply parallelize all tensor operations

Parallelizing SDCA

Two approaches:

- Naive approach: simply parallelize all tensor operations
- Better: mimic the distributed apporach by [Yang, 2013] on a GPU

Implementation

Main Points

Two key areas of concern:

- Memory
    - Allocation
    - Communication

Main Points

Two key areas of concern:

- Memory
  - Allocation
  - Communication
- Code Abstraction (wrappers)

Dealing with Memory Allocation

Naive approach:

```
void MemSync::PushToGpu(const vector &x) {
  double *dx;
  int n_bytes = sizeof(double) * x.size();

  // Allocate GPU memory
  cudaMalloc((double**)&dx, n_bytes);

  // Copy data to GPU
  cudaMemcpy(dx, &x[0], n_bytes, cudaMemcpyHostToDevice)
}
```

On a small dataset(200 points in $\mathbb{R}^3$) we hit over 200k allocations, which comprised nearly **95%** of the GPU compute time (≈ 13 seconds).

Can we do better?

Can we do better?

Yes!

Dealing with Memory Allocation

Better approach:

```
void MemSync::PushToGpu(const vector &x) {
  int n_bytes = sizeof(double) * x.size();

  // Copy data to GPU
  cudaMemcpy(MemSync::dx_, &x[0],
             n_bytes, cudaMemcpyHostToDevice);
}
```

The use of static class pointers reduced the 200k allocations down to only **3 memory allocations** which comprised only **0.03%** compute time ($\approx 180\mu s$).

What about the cost of communication?

Communication Costs

Copying data to and from the GPU is the next most expensive operation

- About 50% of the compute time, or 768 ms (using the same toy dataset)
- Mostly unnecessary!

Communication Costs

Consider the following algorithm using the GPU:

1: $\Delta\omega \leftarrow f(\mathbf{x}, \omega)$

2: $\omega \leftarrow \omega + \Delta\omega$

To handle this efficiently we should:

- Reuse ω in step 2 since we already moved it to the GPU for step 1
- Perform step 2 on the GPU since the data is already there. No need to pull it off and then move it back to the GPU

Communication Costs

The sad reality is this is quite complicated.

- Lots of book-keeping
- Are there edge cases?
- Need to watch out for memory leaks. Remember, **no** GC!

What’s the solution?

What's the solution?

Wrappers!

Wrappers

We use wrappers (also known as decorators) to add additional functionality to our code. For example,

```
double VectorDotProduct(const vector &x,
                        const vector &y) {
#ifdef GPU
  return VectorDotProduct_gpu(x, y);
#else
  return VectorDotProduct_cpu(x, y);
#endif
}
```

Wrappers

To handle the flow of data from GPU to CPU, as well as book-keeping, we can use something like this:

```
class Data {
  enum DataLocation { Gpu, Cpu };

  // Pointer to GPU memory
  double *dx_
  // Local reference (RAM)
  Eigen::VectorXd x_;
  // DataLocation::Gpu or DataLocation::Cpu
  DataLocation location_;
};
```

What about results?

We are still finalizing results. There is a lot of non-trivial structural code behind this.

- Loading data
- Algorithm
- Algorithm performance tracking

Bottou, L. (2010).
Large-scale machine learning with stochastic gradient descent.
In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer.

Boyd, S., Parikh, N., Chu, E., Peleato, B., and Eckstein, J. (2011).
Distributed optimization and statistical learning via the alternating direction method of multipliers.
Foundations and Trends® in Machine Learning, 3(1):1–122.

Johnson, R. and Zhang, T. (2013).
Accelerating stochastic gradient descent using predictive variance reduction.

In *Advances in Neural Information Processing Systems*, pages 315–323.

Lian, X., Huang, Y., Li, Y., and Liu, J. (2015).
Asynchronous parallel stochastic gradient for nonconvex optimization.
In *Advances in Neural Information Processing Systems*, pages 2737–2745.

Nemirovski, A., Juditsky, A., Lan, G., and Shapiro, A. (2009).
Robust stochastic approximation approach to stochastic programming.
SIAM Journal on optimization, 19(4):1574–1609.

Shalev-Shwartz, S. and Zhang, T. (2013).

Stochastic dual coordinate ascent methods for regularized loss minimization.
Journal of Machine Learning Research, 14(Feb):567–599.

Shalev-Shwartz, S. and Zhang, T. (2014).
Accelerated proximal stochastic dual coordinate ascent for regularized loss minimization.
In *ICML*, pages 64–72.

Xiao, L. and Zhang, T. (2014).
A proximal stochastic gradient method with progressive variance reduction.
SIAM Journal on Optimization, 24(4):2057–2075.

Yang, T. (2013).
Trading computation for communication: Distributed stochastic dual coordinate ascent.

In *Advances in Neural Information Processing Systems*, pages 629–637.

Yang, T., Zhu, S., Jin, R., and Lin, Y. (2013).
Analysis of distributed stochastic dual coordinate ascent.
arXiv preprint arXiv:1312.1031.