Scaling up sampled matrix factorization for single-cell transcriptomic data

Sumit Mukherjee[1], Joshua Fan[2]

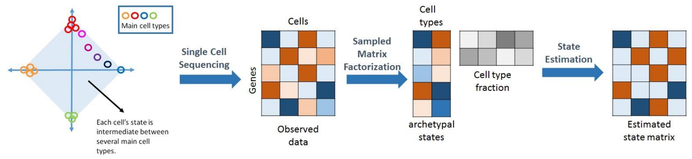
[1]Department of Electrical Engineering, University of Washington, Seattle, USA

[2]Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, USA

Background

With rapid advances in sequencing technologies, large single cell transcriptomic (SCS) datasets are becoming abundant and larger than ever before. In a recent work, 10x Genomics demonstrated their ability to sequence over a million single cells. This has raised the need to scale up the unsupervised learning methods for SCS datasets, most of which aren't designed for such large datasets.

The main problem we focused on was a variant of non-negative matrix factorization called **'Sampled Matrix Factorization'**, which is used to estimate the true transcriptomic states of cells, given heavily sampled data of how much each gene is expressed in a particular cell. In particular, we are interested in efficient online algorithms that can scale well for large datasets.



Basic approach

Non-negative matrix factorization is a well-studied problem that has frequently been used as a dimensionality-reduction technique. Given a data matrix **X**, the goal is to factor **X** into two low-rank non-negative matrices (**M** and **W**) such that some loss function is minimized.

Many matrix factorization approaches assume that the data is drawn from a Gaussian distribution, which is not the case for single-cell RNA-seq data. In our case, we explicitly assume that our observed data matrix **X** is sampled from **SamplingDistribution(MW)**, where SamplingDistribution is usually Poisson or Negative Binomial. In this problem, we minimize the negative **Poisson log likelihood**:

$$\mathbf{MW}^T - \mathbf{X} \bullet \log(\mathbf{MW}^T)$$

The loss function is non-convex for both matrices, but it is convex if you fix one matrix at a time. Thus we can use block coordinate descent: fix **W** and update **M** to decrease the loss function, and then fix **M** and update **W**. Keep alternating until the loss function converges.

For each matrix update, we tried multiple optimization methods.

Algorithms

We implemented six optimization approaches for updating the matrices.

Gradient Descent is the most straightforward approach: we simply find the direction of the likelihood's gradient and adjust the row or column in that direction.

Stochastic Gradient Descent approximates the gradient using a single data point (a single entry in our observed data matrix in our case).

Stochastic Gradient Descent with mini-batching approximates the gradient using a small batch of data points (100 in our case).

SVRG (Stochastic Variance Reduced Gradient) attempts to reduce the inherent variance in stochastic gradient descent. We keep a snapshot of the weight vector after every *m* iterations, which approximates the current weight vector. We also store the average gradient *over the snapshot weights*.

$$\tilde{\mu} = \nabla P(\tilde{w}) = \frac{1}{n}\sum_{i=1}^{n} \nabla \psi_i(\tilde{w})$$

Then, when we update on a new data point, we adjust the point's gradient by how this point's gradient (on the snapshot weights) differs from the average gradient (on the snapshot weights):

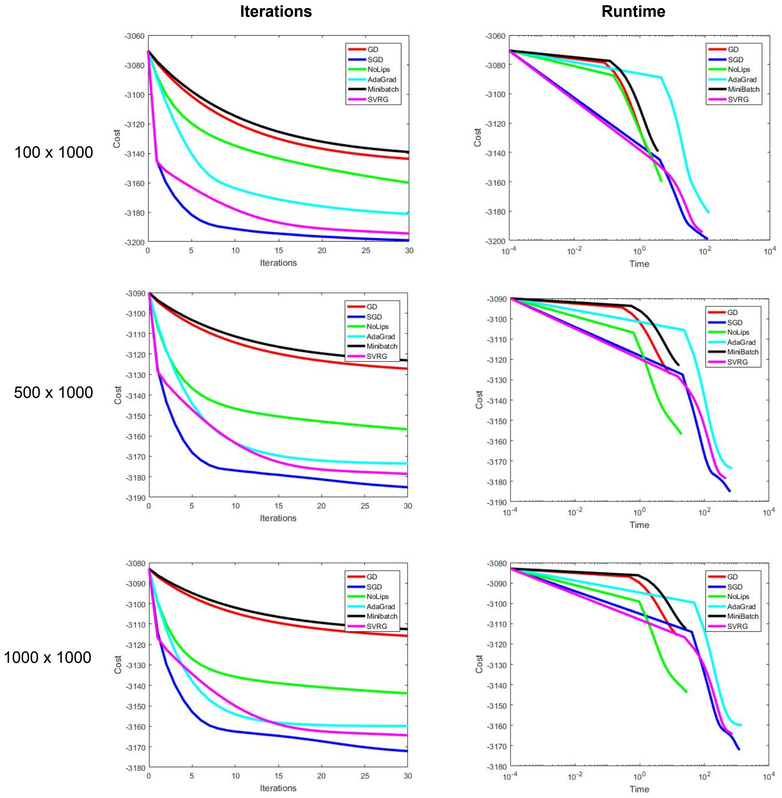
$$w^{(t)} = w^{(t-1)} - \eta_t(\nabla \psi_i(w^{(t-1)}) - \nabla \psi_{i_t}(\tilde{w}) + \tilde{\mu})$$

AdaGrad is an adaptive gradient method which uses feature-specific learning rates which incorporate the past geometry of the data.

NoLip is a specialized algorithm for non-Lipschitz functions. This method utilizes a different defintion of smoothness using Bregman distance and demonstrates that for certain classes of non-lipschitz functions (including our example) it is possible to construct gradient update rules with sub-linear convergence rates.

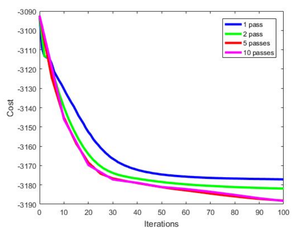
Comparison of methods

We benchmark the various algorithms on synthetic data sets of sizes 100, 500 and 1000. In each case, the optimization procedure involves first using 5 passes through the data to optimize for W and then using 5 passes through the data to optimize for M. We created plots tracking the cost function against the number of iterations and the runtime.



Multiple passes

We experimented with changing the number of **passes through the data set** of updating M before switching to updating W, and so on. Here is an example with a 500 x 1000 matrix (and SGD).

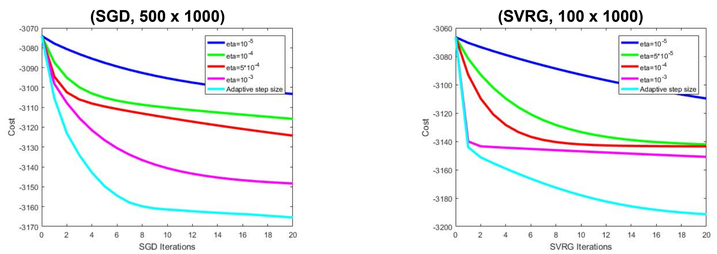


It seems like performing 5 or 10 passes (instead of a single pass) through the data set for each update to M/W yields a slightly faster convergence rate. This could be because if we want to update W in a useful way, we need a good estimate of M (which is fixed).

Choosing step sizes

(Udell et al, 2016) suggests a more nuanced rule for selecting step sizes that allows for different step sizes per row/column. Every time we update a row or column, we check if the objective function (cost) increased or decreased. If the cost decreased, we're moving in the right direction, so we can increase the step size. However, if the cost increased, we jumped too far, so we should decrease the step size for this row/column.

As shown in the graphs below, using this adaptive method yields faster convergence (in SGD and SVRG) than sticking with a single step size throughout. This optimization was used above.



Discussion

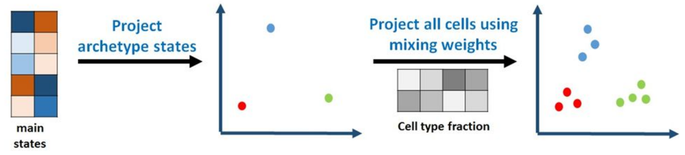
In terms of the number of iterations, the stochastic methods (including SGD, SVRG, and AdaGrad) generally converged faster. SGD reached good results after just several iterations, which was helped using an adaptive method of selecting step sizes.

Surprisingly, regular SGD seemed to do slightly better than SVRG (which explicitly attempts variance reduction) and AdaGrad (which implements feature-specific learning rates). Part of this could simply be due to our choice of learning rates, and because our implementation of SGD already adjusts learning rates per feature, based on their success in decreasing the cost. It also seems like the data is well-behaved enough that variance reduction becomes less necessary.

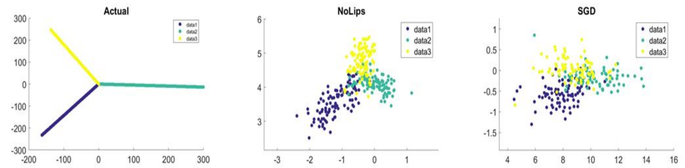
However, in terms of the runtime, the picture is much more mixed. Even though the stochastic methods converge in a small number of iterations, each pass through the data set takes much longer with the stochastic methods. This may be because the batch methods take advantage of large matrix operations when computing gradients, which are more optimized on a hardware level than many small operations. The NoLip method seems to converge in the shortest amount of time, and ordinary Gradient Descent also does well. However, it seems like all the algorithms perform similarly against runtime.

Downstream applications

A potential downstream application of our matrix factorization approach is dimensionality reduction. Our method to perform dimensionality reduction is explained below.



We now demonstrate this method on a synthetic dataset with a branching tree structure (as seen below on unsampled data). We first obtain factorized matrices using different optimization methods and perform dimensionality reduction (see results below).



References

[1] Mukherjee, Sumit, et al. "Prior Knowledge And Sampling Model Informed Learning With Single Cell RNA-Seq Data." *bioRxiv* (2017): 142398.

[2] Dyer, Chris. "Notes on AdaGrad." *School of Computer Science, Carnegie Mellon University* 5000.

[3] Bauschke, Heinz H., Jérôme Bolte, and Marc Teboulle. "A descent Lemma beyond Lipschitz gradient continuity: first-order methods revisited and applications." *Mathematics of Operations Research* (2016).

[4] Johnson, Rie & Zhang, Tong. "Accelerating Stochastic Gradient Descent using Predictive Variance Reduction." *Proceedings of the 26th International Conference on Neural Information Processing Systems* (2013), p. 315-323.

[5] Udell, Madeleine et al. "Generalized Low Rank Models." *Foundations and Trends in Machine Learning* 9.1 (2016), p. 1-118.

[6] Wang, Fei. et al. "Efficient Document Clustering via Online Nonnegative Matrix Factorizations." *Eleventh SIAM International Conference on Data Mining.*