Implementation of: Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent

Charles Dognin
charles.dognin@ensae.fr

Mehdi Abbana Bennani
mehdi.abbana.bennani@ensae.fr

1 INTRODUCTION

Stochastic Gradient Descent is a widely used optimisation algorithm in Machine Learning. As opposed to some other optimisation algorithms, it can scale to very large training datasets. Several approaches were considered for its parallelization, however, these approahces require memory locking, which alters the paralelization speedup. Niu et Al. consider a lock-free approach which is adapted for problems with sparse data. First we recall the mathematical framework and we present the algorithm, and its limits. Then we present our experiments and the obtained results. We eventually conclude.

2 MATHEMATICAL FRAMEWORK

2.1 Stochastic Gradient Descent

Presentation

Recall that in statistical learning, the training problem can be written as:

$$\min_{\omega \in \mathcal{R}^d} \frac{1}{n} \sum_{i=1}^{n} l(h_\omega(x^i), y^i) + \lambda R(\omega)$$

With l being the loss we want to minimize, x and y the data and R the regularization term. In order to solve this problem, the reference method is called gradient descent. Note that if we write $f_i(\omega) = l(h_\omega(x^i), y^i) + \lambda R(\omega)$,

$$\bigtriangledown \left(\frac{1}{n} \sum_{i=1}^{n} f_i(\omega) \right) = \frac{1}{n} \sum_{i=1}^{n} \bigtriangledown f_i(\omega)$$

The Gradient Descent Algorithm yields:

Algorithm 1 Gradient Descent Algorithm

1. **Set** $\omega^0 = 0$, choose $\alpha > 0$
2. **For** $t = 0, 1, 2, \cdot, T-1$
3. $\omega^{t+1} = \omega^t - \frac{\alpha}{n} \sum_{i=1}^{n} \bigtriangledown f_i(\omega^t)$
4. Output ω^T

The main problem with this algorithm is that it requires computing the gradient $\bigtriangledown f_i(\omega^t)$ for each data point which can become quickly computationally unsustainable for large datasets.

The idea of the stochastic gradient descent comes from the following intuition: if j is a random index from $\{1, \cdot, n\}$ selected uniformly, Then, we approximately have:

$$E_j[\bigtriangledown f_j(\omega)] = \frac{1}{n}\sum_{i=1}^{n} \bigtriangledown f_i(\omega) = \bigtriangledown f(\omega)$$

We use now the fact that

$$\bigtriangledown f_j(\omega) \approx \bigtriangledown f(\omega)$$

Algorithm 2 Stochastic Gradient Descent Algorithm

1. **Set** $\omega^0 = 0$, choose $\alpha > 0$
2. **For** $t = 0, 1, 2, \cdot, T-1$
3. sample $j \in \{1, \cdot, n\}$
4. $\omega^{t+1} = \omega^t - \alpha \bigtriangledown f_j(\omega^t)$
5. Output ω^T

Convergence Result

Theorem 1 *If $\frac{1}{\lambda} \geq \alpha > 0$ and f strongly convex, then the iterates of the SGD satisfy:*

$$E[||\omega^t - \omega^*||_2^2] \leq (1-\alpha\lambda)^t ||\omega^0 - \omega^*||_2^2 + \frac{\alpha}{\lambda} B^2$$

With B a bound such that

$$E[|| \bigtriangledown f_j(\omega)||_2^2] \leq B^2$$

for all iterates ω^t

Another widely used version of the stochastic gradient descent is the SGD with decreasing stepsize. Instead of having a fixed stepsize α, we take for instance $\alpha_t = \frac{\alpha}{\sqrt{t+1}}$.

2.2 Hogwild

Our goal is to minimise a separable function $f : X \in \mathbf{R}^n \to \mathbf{R}$ of the form :

$$f(x) = \sum_{e \in E} f_e(x_e) \tag{1}$$

where here e denotes a small subset of $\{1, ..., n\}$ and x_e, the values of the vector x on the coordinates indexed by e. For sparse machine learning problems, which are very frequent, each individual f_e acts only on a very small number of components of x. That is x_e contains only a few components of x. Such examples of problems include Sparse SVM, Recommender Systems, Graph Cuts...

Why normal parallelization would not work?

SGD updates the algorithms parameters sequentially. Multi-processing is inherently complicated because each update needs to wait for the previous one to be finished. A first intuitive approach to parallelize SGD using locks would be for each process, draw a random sample j from training data:

1. Acquire the lock on current state of parameters
2. Read the parameter
3. Update the parameter with SGD step
4. Release the lock

The problem with this approach is that successively acquiring locks would actually take much more time than normally updating the parameters.

What Hogwild proposes is to remove all process locks. processes are actually allowed to overwrite what the others did. This algorithm works because of the sparsity of the gradient. It means that updates do not interfere a lot.

The algorithm assumes a shared memory between processes. Let $G_e(x)$ be the gradient of the function f_e:

Algorithm 3 HOGWILD! update for individual thread

1. **loop**
2. Sample e random uniformly from E
3. Read the current state x_e and evaluate $G_e(x_e)$
4. **for** $v \in e$ **do** $x_v \leftarrow x_v - \gamma G_{ev}(x_e)$
5. **end loop**

It is important to note that the process only modifies the components indexed by e, leaving all the other components alone.

Under certain assumptions, given that the processes are run in parallel and without locks, the speedup is linear in the number of processes.

3 EXPERIMENTS

We implemented the SGD for Recommender Systems in Python, and we used the "multiprocessing" library for the parallelization. The input data is a sparse matrix of movies ratings, where the rows represent the users and the columns the movies. The goal is to complete the matrix (i.e. predict the other ratings). The approach we are going to consider is low rank matrix decomposition[1]. The idea is that we make the assumption that the ratings matrix, we call R, can be decomposed into the product of two low rank matrices U and V, we write $R = UV^T$. Our optimization parameters are the matrices U and V, we call $\hat{R}$ the estimated ratings matrix, and $\mathcal{C}$ the set of the observed indexes. We optimize the empirical loss function with respect to U and V.

$$\mathcal{L}_{U,V}(R, \hat{R}) = \frac{1}{2} \sum_{(i,j)\in\mathcal{C}} (R(i,j) - \hat{R}(i,j))^2 + \lambda(||U||^2 + ||V||^2) \tag{2}$$

This loss can be rewritten as follows :

$$\mathcal{L}_{U,V}(R, \hat{R}) = \frac{1}{2} \sum_{(i,j)\in\mathcal{C}} (R(i,j) - (U_{i:} \cdot V_{:j}))^2 + \frac{\lambda}{2}(\sum_{i=1}^{I} ||U_{i:}||^2 + \sum_{j=1}^{J} ||V_{:j}||^2) \tag{3}$$

This loss is composed of an empirical loss term and a regularization term, which aims to avoid over-fitting, and λ is a trade-off parameter, which controls the strenght of regularization.

In our implementation, we use a slightly different version of SGD, which alternates between the gradient over the parameters U and V. We can see from the second expression of the loss that by considering one line or one column, the gradient of the loss depends only on the gradient that the lines of U (resp. columns of V) which are updated correspond to the indexes of the non-zero columns (resp. non-zero lines). The explicit expressions of the gradients are as follows :

$$\nabla_{U_i} \mathcal{L}_{U,V}(R, \hat{R}) = \sum_{j=1}^{J} (R(i,j) - (U_{i:} \cdot V_{:j}))U_{i:} + \lambda U_{i:}$$

$$\nabla_{V_j} \mathcal{L}_{U,V}(R, \hat{R}) = \sum_{i=1}^{I} (R(i,j) - (U_{i:} \cdot V_{:j}))V_{:j} + \lambda V_{j:}$$

The datasets for recommender systems are typically sparse. When picking two lines or columns from the dataset, we have a low probability of an intersection between the non-zero indexes of two picked observations, therefore, the probability of conflict when updating the gradient is low.

The implementation is available under the following repository https://github.com/charlesdognin/Parallel-RecSys.

We considered the MovieLens dataset[1]. The dataset is an aggregation of ratings of users of the movielens.org website. We consider the 1M dataset (1 million ratings, 6000 users and 4000 movies).

The table 1 summarises the information about the datasets :

Dataset	Ratings	Users	Movies	Ratings / user	Ratings / movie
1M	1 million	6000	4000	167	250

Table 1: Summary of the MovieLens 1M dataset

We first used cross validation in order to determine the best hyperparameters, then, we fixed the hyperparameters and varied the number of processes and monitored the behavior of the algorithm.

4 RESULTS

We considered the following experiments :

- The speedup performance at a fixed number of observations in function of the number of processes and in comparison with the use of lock case
- The convergence speedup through iterations
- The learning degradation due to the conflicts in the updates
- Measure the number of potential conflicts through iterations

We observe a convergence speedup through iterations with the number of processes (figure 1). The figure 2 shows that the gradient update conflicts impact slightly the performance for the short and mid term observations, but this impact disappears asymptotically in the number of observations. This observation is related to the previously computed probability of intersection for this dataset.

For the case without lock, we observe (Table 2) a 1.8x speedup when using two processes, and 3.5x when considering 4 processes. It is natural that the speedup factors are not 2 and 4, because the parallelisation implies a part of the algorithm only, also because of the overheads for example the ones due to the waiting time before syncing the processes.

For the case with the lock, we observe (Table 2) a 1.8x speedup when using two processes, and still 1.8x when considering 4 processes. It is possible that the overhead due to the locks waiting time severly slows down the parallelisation.

The possible conflicts counts (Fig 3 and 4) are of the order of 15% of the updates for the worst case in our experiments, corresponding to 4 processes. We observed that the conflicts alter the peformance on the mid-term but does not alter the performance asymptotically. If it was the case, we could for example run the first iterations in parallel, then finetune the learning using locks for the last iterations.

Number of processes	1	2	4
Runtime without lock (in s)	77.04	43	23.3
Runtime with lock (in s)	70	39	38

Table 2: Comparison of the runtimes for 6000 observations and different numbers of processes, with respect to the usage of a lock for the 1M dataset

[1] https://grouplens.org/datasets/movielens/

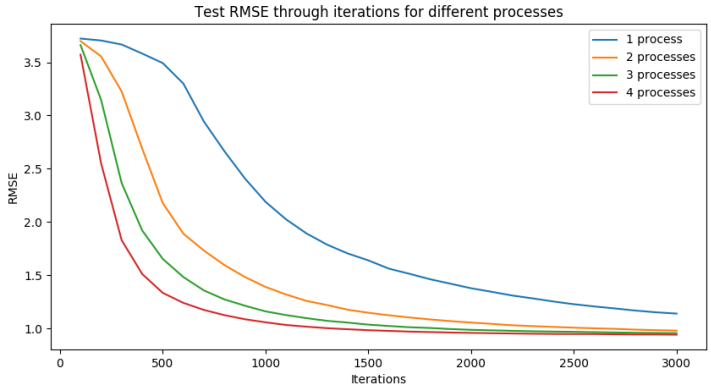

Figure 1: Convergence of the Test RMSE through iterations for runs with different number of processes

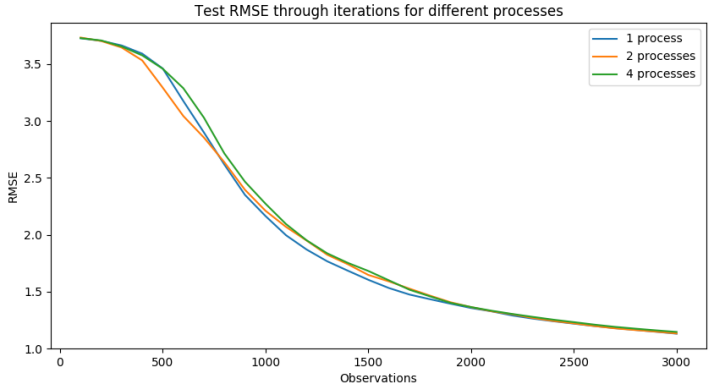

Figure 2: Convergence of the Test RMSE through observations for runs with different number of processes for the 1M dataset

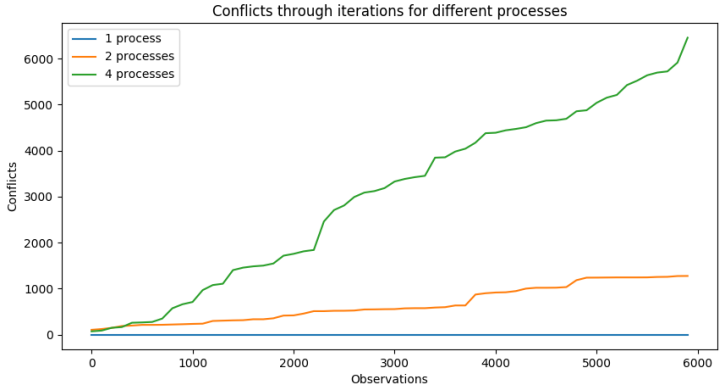

Figure 3: Possible conflicts counts through observations for runs with different number of processes for the 1M dataset

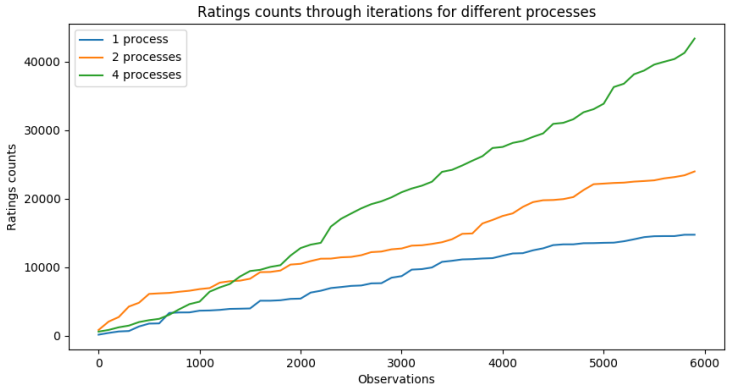

Figure 4: Ratings counts through observations for runs with different number of processes for the 1M dataset

Conclusion

In the context of Recommender Systems, the Hogwell algorithm allows a significant speedup, while preserving the performance of the algorithm. The implementation is also much easier than the Lock case. The algorithm is scalable to datasets of arbitrary size as far as the sparsity is preserved, which is generally the case in the context of Recommender Systems.

References

[1] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, August 2009.

[2] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.