Large-Scale Learning with AdaGrad on Spark

Asmelash Teka Hadgu
L3S Research Center
Hannover, Germany
teka@L3S.de

Aastha Nigam
University of Notre Dame
Indiana, USA
anigam@nd.edu

Ernesto Diaz-Aviles
IBM Research
Dublin, Ireland
e.diaz-aviles@ie.ibm.com

***Abstract*—Stochastic Gradient Descent (SGD) is a simple yet very efficient online learning algorithm for optimizing convex (and often non-convex) functions and one of the most popular stochastic optimization methods in machine learning today. One drawback of SGD is that it is sensitive to the learning rate hyper-parameter. The Adaptive Sub-gradient Descent, AdaGrad, dynamically incorporates knowledge of the geometry of the data observed in earlier iterations to calculate a different learning rate for every feature. In this work, we implement a distributed version of AdaGrad for large-scale machine learning tasks using Apache Spark. Apache Spark is a fast cluster computing engine that provides similar scalability and fault tolerance properties to MapReduce, but in contrast to Hadoop's two-stage disk-based MapReduce paradigm, Spark's multi-stage in-memory primitives allow user programs to load data into a cluster's memory and query it repeatedly, which makes it ideal for building scalable machine learning applications. We empirically evaluate our implementation on large-scale real-world problems in the machine learning canonical tasks of classification and regression. Comparing our implementation of AdaGrad with the SGD scheduler currently available in Spark's Machine Learning Library (MLlib), we experimentally show that AdaGrad saves time by avoiding manually setting a learning-rate hyperparameter, converges fast and can even achieve better generalization errors.**

***Keywords-* Distributed machine learning; Adaptive gradient; Spark.**

I. INTRODUCTION

In many machine learning methods, the task is to optimize an objective function

$$f(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{w}; \boldsymbol{x}_i, y_i) + \lambda R(\boldsymbol{w}),$$

where $\boldsymbol{x}_i \in \mathbb{R}^d$ are training examples, for $1 \leq i \leq n$, and $y_i \in \mathbb{R}$ are the corresponding labels. The objective function f has two parts: (i) The loss $L(\boldsymbol{w};.)$ that measures the error of the model on the training data, and (ii) The regularizer $R(\boldsymbol{w})$ controls the complexity of the model, e.g., the L2 regularizer which is given by $\frac{1}{2}||\boldsymbol{w}||_2^2$. The loss function is typically a convex function in $\boldsymbol{w}$. For example in support vector machines, SVM, the hinge loss $L(\boldsymbol{w};.)$ is defined as $\max\{0, 1 - y\boldsymbol{w}^T\boldsymbol{x}\}, y \in \{-1, +1\}$ and the gradient or sub-gradient is given by $-y\boldsymbol{x}$ if $y\boldsymbol{w}^T\boldsymbol{x} \leq 1$ and 0 otherwise. For regression, it is the squared loss $\frac{1}{2}(\boldsymbol{w}^T\boldsymbol{x} - y)^2$, $y \in \mathbb{R}$ whose gradient is $(\boldsymbol{w}^T\boldsymbol{x} - y)\boldsymbol{x}$. The scalar regularization parameter λ controls the trade-off between minimizing the loss (i.e., training error) and the model complexity (i.e., to avoid overfitting).

Solving this optimization problem usually involves updating parameters $\boldsymbol{w}$ in some iterative procedure. In standard Stochastic Gradient Descent [1], SGD, given $\boldsymbol{w}_t$ the set of parameters at the t-th iteration, the update rule is given by $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta\boldsymbol{g}_t$, where $\boldsymbol{g}_t$ is the gradient of the parameters and η is a scalar learning-rate that determines the step to take in the direction of the negative gradient.

Adaptive Sub-gradient Descent [2], `AdaGrad` is a family of sub-gradient methods that dynamically adapt the learning-rate in gradient descent based learning. In particular, `AdaGrad` alters the scheduling of the learning-rate to adapt based on historical information. Unlike the standard sub-gradient descent that gives one global learning rate η, `AdaGrad` provides for each feature i, a separate learning-rate at each time step t

$$\eta_{t,i} = \frac{\eta_0}{\beta + \sqrt{\sum_{i=1}^{t} g_{t,i}^2}}.$$

Here, η_0 is an initial learning rate, e.g., $\eta_0 = 1.0$; $g_{t,i}^2$ is the squared gradient history of the ith dimension of all previous gradients for that dimension and β is a smoothing constant set to 1.0 in our experiments.

The performance of SGD is sensitive to using an appropriate learning-rate. This usually involves a grid search over potential values during model training. The `AdaGrad` scheduler has several advantages:

- It does not require manually setting the learning-rate. When using `AdaGrad` the initial value of learning-rate is less important as it only affects how much a feature changes the first time it is seen. In the consequent iterations, the algorithm will adapt the learning-rate to the problem. In our experiments, we set the initial learning-rate for `AdaGrad` to 1.0. This saves a tremendous amount of time in practice when building models.
- It provides a per-feature, instead of one global learning-rate, which adapts well to the geometry of the dataset. This means that each feature receives different weights. Rarely occurring features gain more weight than more frequent and less informative features.
- It also gives faster convergence rates at the cost of only a minimum additional computational overhead.

Despite these advantages to `AdaGrad` popular distributed frameworks such as Spark use SGD with a standard decaying learning-rate scheduler $\eta_t = \frac{\eta_0}{\sqrt{t}}$. One reason could be that Apache Spark is still under active development. Our contribution in this work is the implementation of `AdaGrad` scheduler in Spark and an empirical evaluation to compare our implementation with the SGD implementation in MLlib, the machine learning library in Spark [3].

II. Adaptive Sub-gradient Descent in Spark

Before we dive into our implementation of `AdaGrad` let us briefly look at the two abstractions Spark provides to make it easy to build scalable distributed systems. At a high level, every Spark application consists of a *driver* program that runs the user's main function and executes various parallel operations on *worker* nodes part of the cluster. The main abstraction Spark provides is a *resilient distributed dataset* (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs automatically recover from node failures. RDDs support two types of operations: *transformations,* which create a new RDD from an existing one and *actions,* which return a value to the driver program after running a computation on the RDD.

For example, *map* is a transformation that passes each element in an RDD through a function and returns a new RDD representing the result. On the other hand, *reduce* is an action that aggregates all the elements of an RDD using some function and returns the final result to the driver program. In the MapReduce paradigm, the driver builds new models at the end of each iteration by aggregating partial updates collected from executors. *Aggregate* is a transformation that allows the user to specify two combine functions and a neutral "zero value" to aggregate the elements of each partition using the first combine function, and then the results for all the partitions using the second combine function. The neutral "zero value" is used at both levels of reduce. One performance issue with the *aggregate* function is that the aggregation time scales linearly with respect to the number of partitions of data (due to the CPU cost in merging partial results and the network bandwidth limit). *TreeAggregate* is a transformation similar to *aggregate*, except it aggregates the elements of the RDD in a multi-level tree pattern. In this setup, model updates are combined partially on a small set of executors before they are sent to the driver, which dramatically reduces the load the driver has to deal with.

A second abstraction in Spark is shared variables that can be used in parallel operations. By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task. Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. Spark supports two types of shared variables: broadcast variables, which can be used to cache a value in memory on all nodes, and accumulators, which are variables that are only added to, such as counters and sums.

Equipped with the necessary components, we are ready to explain how the `AdaGrad` implementation highlighted in Algorithm 1 works. It is entirely based on the source code in Spark MLlib[1] as reference implementation. The algorithm

[1]**Spark MLlib**: https://github.com/apache/spark .

Algorithm 1 Distributed AdaGrad on Spark

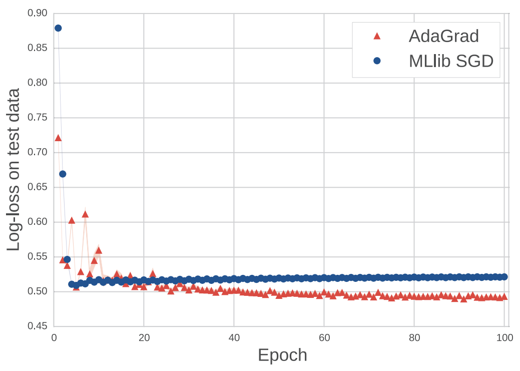
```
Input: Initial learning-rate $\eta_0 > 0$, regularization parameter $\lambda \geq 0$,
    mini-batch fraction $mb > 0$, iterations (epochs) $T$
Output: Model $\boldsymbol{w}$
1: Initialize: weights $\boldsymbol{w}_0 = \mathbf{0}$ as broadcast variable, gradient
   history $\boldsymbol{g}_0^2 = \mathbf{0}$
2: for t = 1 to $T$ do
3:     Compute mini-batch
4:     Compute gradient            ▷ distributed across worker nodes
5:     Aggregate gradient: $g_t$  ▷ multiple levels using TreeAggregate
6:     Update gradient history: $g_{t,i}^2 = \sum_{t'=1}^{t} g_{t',i}^2$
7:     Update weights: $w_{t+1,i} = w_{t,i} - \frac{\eta_0}{\beta + \sqrt{\sum_{i=1}^{t} g_{t,i}^2}} g_{t,i}$
8: end for
9: return $\boldsymbol{w}$
```

requires some parameters: initial learning-rate η_0 which is fixed to 1.0 in our experiments, regularization parameter λ and mini-batch fraction $mb > 0$. The initial weights $\boldsymbol{w}_0 = 0$ are declared as broadcast variable. This helps avoid unnecessary network traffic to communicate the weights from the driver to the worker nodes and back. For each iteration, the driver computes the mini-batch which is $mb * |n|$. Then through *treeAggregate* the worker nodes compute the gradient on this batch. The gradients are aggregated at multi-tree levels. Finally, the gradient history update and the weight update take place at the driver program.

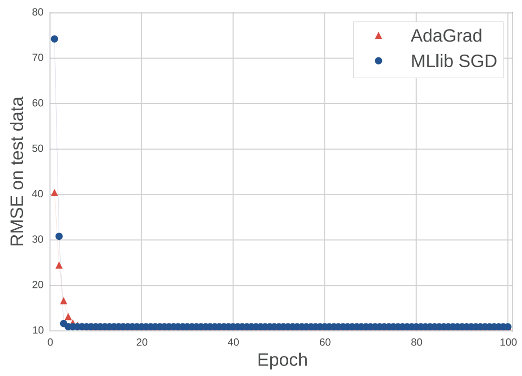
III. Experimental Evaluation

In this section, we describe the experimental evaluation setup. Specifically, we apply `AdaGrad` on multiple machine learning tasks in classification and regression to empirically evaluate the performance against the SGD scheduling implementation on Spark MLlib. For ease of reproducibility and dataset accessibility, we adapt lab sessions from the edX MOOC class Scalable Machine Learning [4].

Classification Task. We leverage a subset of the dataset that is used for the Display Advertising Challenge hosted by Kaggle [5]. The dataset contains feature values and click feedback of display ads. The purpose is to benchmark algorithms for click-through rate (CTR) prediction. Given a user and the page they are visiting, the task is to predict the probability that they will click on a given ad. In our experiments, we use a sample of the dataset that contains 100,000 display ads and 39 (categorical and numerical) features. The preprocessing involves one-hot-enocding, OHE, to transform categorical features to numerical. This results in 233,286 OHE features for each ad. To reduce dimensionality of this large feature vector, we use feature hashing with a number of buckets = 2^{15} which reduces the feature vector to 32,768 entries. These features are then represented using sparse vectors to reduce storage and computational overhead. We trained an SVM model to solve this task. We used ten-fold cross validation to determine the best hyperparameters for the learning-rate and also for the regularization parameter in the case of SVM implementation in Spark, which correspond to $\eta_0 = 1.0$ and $\lambda = 1 \times 10^{-7}$.



(a) Classification.



(b) Regression.

Figure 1. MLlib SGD vs. AdaGrad for SVM (left) and Ridge Regression (right).

For `AdaGrad` we did not tune up the learning rate, which was set to $\eta_0 = 1.0$, and experimentally set $\lambda = 1 \times 10^{-7}$. The mini batch fraction is set to $mb = 0.1$ for both algorithms.

Regression Task. We use a subset of the Million Song Dataset [6], a collection of audio features and metadata for 515,345 music tracks. Each track has 90 numeric audio features. The task is to train models to predict the year a song was released from these set of features. We perform only two simple preprocessing on the dataset: the first one is to shift the labels such that they start from zero. In addition, we scale all feature values to be between 0.0 and 1.0 inclusive. Finally, we built ridge regression models for this task. As in the classification task, we used ten-fold cross validation to determine the best parameters in a grid-search approach. For `AdaGrad` the initial learning-rate is again fixed to $\eta_0 = 1.0$ and we found that a regularization parameter $\lambda = 1 \times 10^{-3}$ led to good results. For SGD MLlib the parameters are $\eta_0 = 0.1$ and $\lambda = 1 \times 10^{-3}$. The mini batch fraction is set to $mb = 1 \times 10^{-3}$.

For the classification and regression tasks the number of iterations was set to $T = 100$, and for the optimal hyperparameters settings, we retrain the models on a 80-20 split and repeat the experiment for 30 runs to account for variability.

Evaluation metrics. For the CTR classification task, we use the logarithmic loss, *log-loss*, to assess the performance of the learning algorithms. Log-loss is a standard evaluation metric when predicting rare-events, which is defined as $\textit{log-loss} = -\frac{1}{n}\sum_{i=1}^{n}(y_i \, log(\hat{y}_i) + (1-y_i)\, log(1-\hat{y}_i))$, where y is the actual boolean value indicating whether an ad was clicked and $\hat{y}$ is the predicted probability of click on each ad.

For the MSD Regression task to represent the sample standard deviation of the differences between the predicted year ($\hat{y}$) values and the actual release years, y, of a song. We use the root-mean-squared error to assess model performance, which is defined as follows $\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}$.

Results. One way to compare the schedulers is to compare the learning curves during training. Yet a stronger and better measure is to compare the performance on test data, i.e., to compare the generalization error. Figure 1 shows the generalization error of `AdaGrad` and the SGD implementation on Spark for different epoches. We can see that `AdaGrad` indeed has a good generalization error and is better for the classification task. We also observe that `AdaGrad` converges fast on both tasks. More importantly, the initial learning-rate for `AdaGrad` is fixed to $\eta_0 = 1.0$. The SGD implementation in Spark is sensitive to the value of the learning-rate. Typically a user may perform a grid search over potential values to fine-tune the learning-rate parameter e.g., $\{1 \times 10^{-7}, 1 \times 10^{-6}, \ldots, 0.1, 1.0\}$. This implies that `AdaGrad` offers a significant speed-up for experimental evaluations as it avoids fine-tuning the learning-rate hyperparameter.

IV. Conclusion

In this work, we implemented `AdaGrad` on Spark for large scale learning. We compared its performance against the default scheduling implementation for SGD in MLlib. Our experiments show that `AdaGrad` gives fast convergence and better generalization errors. In addition, it eases model selection time by avoiding having to do grid search over potential values to fine-tune the learning-rate hyperparameter. Future work includes an exploration of a stream variant of AdaGrad on Spark streaming for recommender systems applications (e.g., [7]).

References

[1] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[2] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *The Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011.

[3] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," *arXiv preprint arXiv:1505.06807*, 2015.

[4] A. Talwalkar. Berkeleyx cs190-1x: Scalable machine learning. https://www.edx.org/. Accessed: 2015.

[5] Kaggle display advertising challenge. http://labs.criteo.com/downloads/2014-kaggle-display-advertising-challenge-dataset. Accessed: 2015.

[6] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere, "The million song dataset," in *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[7] E. Diaz-Aviles, L. Drumond, L. Schmidt-Thieme, and W. Nejdl, "Real-time top-n recommendation in social streams," in *Proceedings of Conference on Recommender Systems – RecSys*, 2012.