Towards Resource-Elastic Machine Learning

Shravan Narayanamurthy, Markus Weimer, Dhruv Mahajan
Tyson Condie, Sundararajan Sellamanickam, Keerthi Selvaraj
Microsoft
[shravan|mweimer|dhrumaha|tcondie|ssrajan|keerthi]@microsoft.com

1 Introduction

The availability of powerful distributed data platforms and the widespread success of Machine Learning (ML) has led to a virtuous cycle wherein organizations are investing in gathering a wider range of (even bigger!) datasets and addressing an even broader range of tasks. The Hadoop Distributed File System (HDFS) is being provisioned to capture and durably store these datasets. Along side HDFS, resource managers like Mesos [10], Corona [8] and YARN [16] enable the allocation of compute resources "near the data," where frameworks like REEF [3] can cache it and support fast iterative computations. Unfortunately, most ML algorithms are not tuned to operate on these new cloud platforms, where two new challenges arise: 1) **scale-up:** the need to acquire more resources dedicated to a particular algorithm, and 2) **scale-down:** the need to react to resource preemption. This paper focuses on the scale-down challenge, since it poses the most stringent requirement for executing on cloud platforms like YARN, which reserves the right to preempt compute resources dedicated to a job (tenant) [16].

YARN exposes compute resources (called *containers*) through a central Resource Manager (RM). The RM mediates container requests from multiple tenants that want to execute some form of a job i.e., MapReduce, Pregel, SQL, Machine Learning. YARN assigns containers according to some policy that is typically based on fairness, priority or monetary compensation. These local policies are used to derive a global scheduling decision among multiple tenants that ensures each tenant is given a satisfactory container allocation, and that (ideally) cluster utilization is kept high. The challenge is satisfying these global properties as new tenants enter and leave the system. The key mechanism for achieving this goal is preemption [7, 16], which allows the RM to recall (preempt) previously assigned containers so that they may be given to another tenant that improves Pareto optimality.

Jobs need to react to these preemption requests. Trivially, preemption can be viewed as a (task/container) failure, which systems such as Hadoop MapReduce [1] can well accommodate through aggressive task check-pointing and restart. However, such levels of check-pointing have been frequently found to be detrimental to the performance of jobs that are (somewhat) immune to such fine-grained failures e.g., small/short jobs as well as Machine Learning systems that blatantly sacrifice fault tolerance for the sake of performance [4, 17, 14, 5]. These systems rely on restarting the whole computation on a failure, requiring the user to execute several attempts of an ML job until one of them succeeds. Unfortunately, this strategy is bound to fail in the presence of preemption, which is likely to be more common than faults; especially in a system under heavy load; making it unlikely to ever complete a job that employs such a restart strategy.

In this abstract, we present our initial findings in integrating resource elasticity as a first-class citizen into ML algorithms. We present a resource-elastic linear learning algorithm as a stand-in for statistical query model (SQM) [11] algorithms in Section 2. It assumes random partitioning of the data onto containers; giving up a container therefore is equivalent to drawing a random sample of the whole dataset. We assume that we can choose when to give up a container within some bounded time, such is the case for YARN [2]. In Section 3 we then describe initial results on our implementation on REEF [3]. Section 4 offers our plans for future work.

2 Resource Elastic Linear Learning

We consider the following convex objective function,

$$f(w; X, Y) = \sum_p l_p(w^t X_p, Y_p) + \frac{\lambda}{2} w^t w, \tag{1}$$

where l_p,$\{X_p, Y_p\}$ are part of loss function and data respectively in partition p. For ease of exposition, we use the distributed Batch Gradient Descent Algorithm 1 here to optimize Equation 1 in lieu of other well known methods like SGD [6], LBFGS [13] and Trust Region Newton [12].

Algorithm 1: Distributed Batch Gradient Descent (Distr-BGD)

Master: Choose w^0;
for $r = 0, 1 \ldots$ **do**
1. **Master**: `Broadcast` w^r to all the slaves.;
2. **Partition p**: Receive w^r and compute partial gradient $g_p^r = \nabla l_p$ at w^r;
3. **Partition p**: Perform `Reduce` operation on g_p^r.;
4. **Master**: Receive the output, $g^r = \sum_p g_p^r$ of the reduce operation.;
5. **Master**: Update the weight vector, $w^{r+1} = w^r - \eta^r(g^r + \lambda w^r)$.;

end

The master first passes on model w^r to all the slave nodes using a `Broadcast` operation. The slave partitions then compute the partial gradients g_p^r and perform the `Reduce` operation to aggregate them with the master as the root node. The master then receives the overall gradient and updates w^r. The step η^r is either a constant or decays with time. Alternatively, one can also do a line search. This relies on two main communication operators: `Broadcast` and `Reduce`. The readers are referred to [15] for the details of these operators. In this paper, we use a simple binary tree as in [4] to implement them.

Elasticity Model: We make the following assumptions: 1) Container removal can occur at any step of the algorithm, 2) It occurs due to preemption or failure of nodes themselves rather than issues with the algorithm implementation or data, 3) At some future time we are guaranteed to get the containers back, and 4) Only leaf nodes in the binary tree vanish[1].

Our Approach: Let us say, during iteration r the partitions with indices in set Q disappear. We react to this on two levels: 1) We make the `Broadcast` and `Reduce` elastic, and 2) We approximate the loss function on the missing partitions and use this information in the overall optimization.

Elastic `Broadcast` and `Reduce`: Elastic `Broadcast` means that all the active slave containers in $\bar{Q}$ will still receive the broadcast from the master. Similarly, the output of `Reduce` will return the aggregated result of active containers only.

Approximation for vanished partitions: We approximate the sum of loss functions $l_Q = \sum_{p \in Q} l_p$ by first order Taylor expansion, $\hat{l}_Q = \hat{g}_Q^T(w - w^{r-1})$, where $\hat{g}_Q = \sum_{q \in Q} g_q^{r-1}$. The master uses this approximation in r^{th} and subsequent iterations till the partitions come up again. This has two advantages: 1) The quality of our solution will be better because we do not ignore the vanished nodes completely, and b) We do not have to wait for the partitions to return.

Computing $\hat{g}_Q$: Once the Master observes that partitions have vanished, it broadcasts w^{r-1} to slave partitions and gets $g_{\bar{Q}}^{r-1}$ with the `Reduce` operation. It then calculates $\hat{g}_Q = g^{r-1} - g_{\bar{Q}}^{r-1}$. Note that the master needs to store the previous gradient g^{r-1} only. If extra partitions vanish during the reduce operation, the master will compute the approximation of all vanished partitions together.

Algorithm 2 contains this approach. FN denotes the set of partitions vanished so far in the algorithm. The master maintains a $FIFO$ queue that stores Q as well as approximation $\hat{g}_Q$ of vanished partitions in a given iteration. The variable $\hat{g}_{FN}$ maintains the aggregated approximate gradient i.e.

[1]This last assumption is made for the sake of ease of exposition only.

$\hat{g}_{FN} = \sum_{q \in FN} \hat{g}_q$. The master reschedules the partitions in the $FIFO$ queue as new containers become available and updates $\hat{g}_{FN}$ and queue.

Although theoretically possible, failures or preemption don't occur in every iteration of the algorithm in practice. A proper analysis and modeling of the interval between the consecutive resource allocation changes and formal proof of convergence of the algorithm[2] is left for future work.

Algorithm 2: Elastic Distr-BGD

begin
- 1.**Master**: $FNQueue \longleftarrow \emptyset$; `// initialize vanished partition queue to empty`
- 2 **Master**: $FN \longleftarrow \emptyset$; `// initialize set of vanished partitions to empty`
- 3.**Master**: $\hat{g}_{FN} \longleftarrow 0$;
- 4. **Master**: Choose w^0;
- **for** $r = 0, 1 \ldots$ **do**
  - 5. **Master**: `Broadcast` w^r to all the slaves;
  - 6. **Partition p**: Receive w^r and compute partial gradient $g_p^r = \nabla l_p$ at w^r;
  - 7. **Partition p**: Perform `Reduce` operation on g_p^r;
  - 8. **Master**: Receive $g^r = \sum_p g_p^r$ and Q from the `Reduce` operation.; `//` Q `is set of vanished partitions`
  - **Master**: **if** $Q \neq \emptyset$`// check if some partition has vanished` **then**
    - 9. **Master**: Broadcast w^{r-1} to all the slaves.;
    - 10. **Partition p**: Receive w^{r-1} and compute partial gradient $g_p^{r-1} = \nabla l_p$ at w^{r-1};
    - 11. **Partition p**: Perform `Reduce` operation on g_p^{r-1}.;
    - 12. **Master**: Receive $g_{\bar{Q}}^{r-1} = \sum_p g_p^r$ and Q from the `Reduce` operation.; `//` Q `also includes vanished partitions from previous reduce in Step 8`
    - 13. **Master**: $\hat{g}_Q \longleftarrow g^{r-1} - g_{\bar{Q}}^{r-1}$;
    - 14. **Master**: $\hat{g}_{FN} \longleftarrow \hat{g}_{FN} + \hat{g}_Q$, $FN \longleftarrow FN \cup Q$;
    - 15. **Master**: $FNQueue.Add(Q, \hat{g}_Q)$.; `// Add approximated gradient and set of faulty nodes to queue`
    - 16. **Master**: $w^{r+1} \longleftarrow w^r$, $w^r \longleftarrow w^{r-1}$
  - **end**
  - **else**
    - 17. **Master**: $w^{r+1} \longleftarrow w^r - \eta^r(g^r + \hat{g}_{FN} + \lambda w^r)$;
  - **end**
  - 18. **Master**$(Q, \hat{g}_Q) \longleftarrow FNQueue.top$;
  - **Master**:**if** *number of free nodes available* $\geq |Q|$ **then**
    - 19. **Master**: $FNQueue.pop$;
    - 20. **Master**: Request to bring the partitions with indices in Q up;
    - 21. **Master**: $\hat{g}_{FN} \longleftarrow \hat{g}_{FN} - \hat{g}_Q$, $FN \longleftarrow FN - Q$;
  - **end**
- **end**

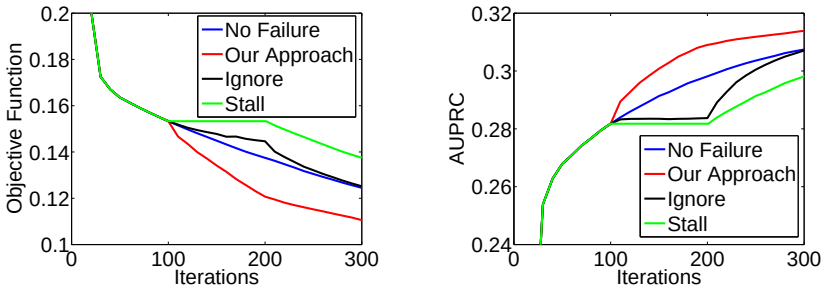
end

3 Experimental Results

Implementation: We implemented Algorithm 2 on REEF [3], which offers event-driven abstractions on top of resource managers. Crucially, it provides events for container (de-)allocation, which then trigger reconfiguration of our elastic `Broadcast` and `Reduce` operators. The latter is implemented as a binary tree whose inner nodes keep an approximation for their inputs available, which facilitates seamless container deallocations. Our `Reduce` function also returns the set of active partitions $\bar{Q}$ to the master, which enables the elasticity treatment in Algorithm 2. Moreover, we perform line search in Algorithms 1 and 2 to find the step η^r.

Dataset: We use a subset of 4 million examples of the splice dataset described in [4]. The raw data consists of strings of length 141 with 4 (A, T, C, G) alphabets. The train and test sizes are 3.6M and 400K respectively. We derive the binary presence or absence of $n-grams$ at specific locations

[2]Convergence proof will require modifying the algorithm to do proper line search (with AGW conditions) when the node comes up again (Steps 17-19).



(a) Objective Function (b) AUPRC

Figure 1: Plots showing the preemption scenarios. 7 nodes are taken away at the 100^{th} iteration due to preemption and are given back again at iteration 200.

of the string with $n = [1, 4]$ as features. The dimensionality of the feature space is $47,028$ and the overall data size is around $16GB$.

Experimental setup: We run our experiments on a 12 core machine with hyper-threading and $96GB$ RAM. We use $P = 14$ in all our experiments. We preempt the algorithm by taking away $7(50\%)$ nodes at iteration 100 and giving them back at iteration 200.

Baselines and evaluation criteria: We compare our results with two baselines, a) $Stall$: Distr-BGD algorithm waits for all the nodes to come up again, and, b) $Ignore$: Distr-BGD continues with only the remaining containers until they become available again. We use training objective function value and Area under Precision-Recall Curve (AUPRC) on the test set as evaluation metrics.

Observations: Figure 1a shows the objective function value as a function of the number of iterations. Note that both the iterations and overall time taken are directly proportional to each other. Our method shows considerable improvement over the baselines. Surprisingly, we even do better than the no failure case. This can be explained by the fact that the use of the past gradient has similarities to adding the momentum term [9] for gradient descent algorithm which is well known to have a beneficial effect. Similar observations can be made for AUPRC metric on test data in Figure 1b. The experiments clearly show the utility of designing fault-aware ML algorithms.

4 Conclusions and Future work

In this abstract, we explored the idea of treating resource elasticity as a first class tenant in machine learning algorithms. To the best of our knowledge, this is the first time this connection has been made. Our initial results confirmed that doing so can yield substantial improvements over the state of the art: Forcing the runtime to absorb the resource elasticity yields high overheads (e.g. in MapReduce) and treating any container preemption as a job failure wastes compute cycles.

This encouraging result motivated our current and future work in this area: the findings can and need to be substantiated on larger datasets. We also need to compare against stronger baselines like, continuing the optimization while ignoring the vanished nodes completely. The insight will be generalized to other SQM algorithms and graphical models. All of this will move us closer to predictable and reliable performance of machine learning on the cloud.

Acknowledgments

Our implementation makes heavy use of REEF[3], a new distributed computing framework the authors of the paper are involved in. We'd like to thank our collaborators on that project without which the present work would not have been possible.

References

[1] Hadoop MapReduce. `http://hadoop.apache.org/`.

[2] Preemption and restart of MapReduce tasks. `https://issues.apache.org/jira/browse/MAPREDUCE-4584`.

[3] REEF: The retainable evaluator execution framework. `http://www.reef-project.org`.

[4] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system, 2011.

[5] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM '12: Proceedings of the fifth ACM international conference on Web search and data mining*, pages 123–132, New York, NY, USA, 2012. ACM.

[6] L. Bottou. Large-Scale Machine Learning with Stochastic Gradient Descent. In Y. Lechevallier and G. Saporta, editors, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, Paris, France, Aug. 2010. Springer.

[7] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin. Natjam: Design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. October 2013.

[8] Facebook Engineering. Under the Hood: Scheduling MapReduce jobs more efficiently with Corona, November 2012.

[9] S. Haykin. *Neural Networks and Learning Machines (3rd Edition)*. Prentice Hall, 3 edition, Nov. 2008.

[10] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22. USENIX Association, 2011.

[11] M. Kearns. Efficient noise-tolerant learning from statistical queries. *J. ACM*, 45(6):983–1006, Nov. 1998.

[12] C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust region newton method for logistic regression. *J. Mach. Learn. Res.*, 9:627–650, June 2008.

[13] D. C. LIU and J. NOCEDAL. On the limited memory BFGS method for large scale optimization. *Math. Programming*, 45(3, (Ser. B)):503–528, 1989.

[14] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, California, July 2010.

[15] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[16] V. Vavilapalli and et. al. Apache hadoop yarn: Yet another resource negotiator. In *ACM Symposium on Cloud Computing*, SoCC'13, October 2013.

[17] M. Weimer, S. Rao, and M. Zinkevich. A convenient framework for efficient parallel multipass algorithms. In *LCCC : NIPS 2010 Workshop on Learning on Cores, Clusters and Clouds*, December 2010.