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. Alongside HDFS, resource managers like Mesos [9], Corona [8] and YARN [15] 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) [15].

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, 15], 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, 16, 13, 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) [10] 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.
\section{Resource Elastic Linear Learning}

We consider the following convex objective function,

\[ f(w; X, Y) = \sum_{p} l_p(w^p X_p, Y_p) + \frac{\lambda}{2} w^t w, \]  

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 \([12]\) and Trust Region Newton \([11]\).

\textbf{Algorithm 1: Distributed Batch Gradient Descent (Distr-BGD)}

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

The master first passes on model \( w^r \) to all the slave nodes using a \textbf{Broadcast} operation. The slave partitions then compute the partial gradients \( g_p^r \) and perform the \textbf{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 \( \eta^r \) is either a constant or decays with time\(^1\). This relies on two main communication operators: \textbf{Broadcast} and \textbf{Reduce}. The readers are referred to \([14]\) for the details of these operators. In this paper, we use a simple binary tree as in \([4]\) to implement them.

\textbf{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\(^2\).

\textbf{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 \textbf{Broadcast} and \textbf{Reduce} elastic, and 2) We approximate the loss function on the missing partitions and use this information in the overall optimization.

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

\textbf{Approximation for vanished partitions:} We approximate the sum of loss functions \( l_Q = \sum_{p \in Q} l_p \) by first order Taylor expansion, \( \hat{l}_Q = \hat{g}_Q^r (w - w^{r-1}) \), 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.

\textbf{Computing} \( \hat{g}_Q \): Once the Master observes that partitions have vanished, it broadcasts \( w^{r-1} \) to slave partitions and gets \( \hat{g}_Q^{r-1} \) with the \textbf{Reduce} operation. It then calculates \( \hat{g}_Q = g^{r-1} - \hat{g}_Q^{r-1} \). 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.

\footnotesize
\(^1\) One can also do a line search.

\(^2\) 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\(^3\) is left for future work.

---

**Algorithm 2: Elastic Distr-BGD**

```plaintext
begin
1. Master: \( FNQueue \leftarrow \emptyset; \) // initialize vanished partition queue to empty
2. Master: \( FN \leftarrow \emptyset; \) // initialize set of vanished partitions to empty
3. Master: \( \hat{g}_{FN} \leftarrow 0; \)
4. Master: Choose \( w^0; \)
5. for \( r = 0, 1, \ldots \) do
   5.1. Master: Broadcast \( w^r \) to all the slaves;
   5.2. Partition \( p \): Receive \( w^r \) and compute partial gradient \( g^r_p = \nabla l_p \) at \( w^r \);
   5.3. Partition \( p \): Perform Reduce operation on \( g^r_p; \)
   5.4. Master: Receive \( g^r = \sum p g^r_p \) and \( Q \) from the Reduce operation.; // \( Q \) is set of vanished partitions
   5.5. if \( Q \neq \emptyset; \) // check if some partition has vanished then
      5.5.1. Master: Broadcast \( w^{r-1} \) to all the slaves;
      5.5.2. Partition \( p \): Receive \( w^{r-1} \) and compute partial gradient \( g^{r-1}_p = \nabla l_p \) at \( w^{r-1} \);
      5.5.3. Partition \( p \): Perform Reduce operation on \( g^{r-1}_p; \)
      5.5.4. Master: Receive \( g^{r-1}_Q = \sum p g^{r-1}_p \) and \( Q \) from the Reduce operation.; // \( Q \) also includes vanished partitions from previous reduce in Step 8
      5.5.5. Master: \( \hat{g}_Q \leftarrow g^{r-1}_r - g^{r-1}_r; \)
      5.5.6. Master: \( \hat{g}_{FN} \leftarrow \hat{g}_{FN} + \hat{g}_Q, FN \leftarrow FN \cup Q; \)
      5.5.7. Master: \( FNQueue.Add(Q, \hat{g}_Q); \) // Add approximated gradient and set of faulty nodes to queue
      5.5.8. Master: \( w^{r+1} \leftarrow w^r, w^r \leftarrow w^{r-1} \)
   5.5 else
      5.6. Master: \( w^{r+1} \leftarrow w^r - \eta(g^r + \hat{g}_{FN} + \lambda w^r); \)
5.7. Master\( (Q, \hat{g}_Q) \leftarrow FNQueue.top; \)
end
6. Master: if number of free nodes available \( \geq |Q| \) then
   6.1. Master: \( FNQueue.pop; \)
6.2. Master: Request to bring the partitions with indices in \( Q \) up;
6.3. Master: \( \hat{g}_{FN} \leftarrow \hat{g}_{FN} - \hat{g}_Q, FN \leftarrow FN - Q; \)
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 \( Q \) to the master, which enables the elasticity treatment in Algorithm 2.

**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. We derive the binary presence or absence of \( n \)-grams at specific locations of the string with \( n \in [1, 4] \) as features. The dimensionality of the feature space is 47,028 and the overall data size is around 16GB.

\(^3\)Convergence proof will require modifying the algorithm to do proper line search (with AGW conditions) when the node comes up again (Steps 17-19).
Figure 1: Plots showing two different preemption/failure scenarios. a) 7 nodes are taken away at the 100th iteration due to preemption. b) 3 nodes each are taken away at iteration nos. 50 and 100. The two sets of nodes come back at 150th and 200th iteration respectively.

**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 also compare our results with a simple baseline, Stall, in which Distr-BGD algorithm waits for all the nodes to come up again.

**Simulated Scenarios:** We simulate two different scenarios. In the first simulation, we preempt the algorithm by taking away 7(50\%) nodes at iteration 100 and giving them back at iteration 200. In the second scenario, two sets, each with 3 nodes, vanish in iterations 50 and 100. However, they come up again at iterations 150 and 200 respectively.

**Observations:** Figure 1 shows the variation of \( \log((f - f^*)/f^*) \) as a function of the number of iterations, where \( f^* \) is the optimal function value. We obtained \( f^* \) by running the algorithm to get a very accurate solution. Note that both the iterations and overall time taken are directly proportional to each other. Our method shows considerable improvement over the baseline for both the scenarios. The gap is particularly significant in (b) where the baseline is stalled from iterations 50 to 200. 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.

---

4If accepted, results on the actual cluster with few thousand cores will be made available for the camera ready version
References