MapReduce/Bigtable for Distributed Optimization

Keith B. Hall
Google Inc.
kbhall@google.com

Scott Gilpin
Google Inc.
sgilpin@google.com

Gideon Mann
Google Inc.
gmann@google.com

Abstract

With large data sets, it can be time consuming to run gradient based optimization, for example to minimize the log-likelihood for maximum entropy models. Distributed methods are therefore appealing and a number of distributed gradient optimization strategies have been proposed including: distributed gradient, asynchronous updates, and iterative parameter mixtures. In this paper, we evaluate these various strategies with regards to their accuracy and speed over MapReduce/Bigtable and discuss techniques and configurations needed for high performance.

1 Introduction

The increasing availability of voluminous training data and the desire to leverage all of this data has lead many people to investigate various strategies for distributed machine learning, where typically gradient optimization is the most computationally costly step. Groups have proposed strategies for distributed training, with varied qualities to advocate for them (e.g. reduced network usage and convergence bounds) but to our knowledge there has not been an exploration of the tradeoffs associated with training these systems on a modern data center: one composed of commodity machines with relatively few cores per machine.

In this paper we experimentally explore one particular training problem – conditional maximum entropy aka logistic regression. We choose this problem since it (1) has been the focus of many studies on distributed training e.g. [3], (2) is used in practice for many large and important prediction problems, and (3) has been extended to structured output settings [6] where many of the lessons learned for this model can be applied. We run experiments that demonstrate iterative parameter mixtures to have appealing runtime/accuracy properties for the data set we examine, and we highlight particular implementation details that can have drastic impact on overall performance.

2 Distributed Optimization for Maximum Entropy Models

Let $\mathcal{X}$ be the input space, $\mathcal{Y}$ the output space, and $\Phi\colon \mathcal{X}\times\mathcal{Y}\to H$ a (feature) mapping to a Hilbert space H, which in many practical settings coincides with $\mathbb{R}^N$, $N=\dim(H)<\infty$.

Let $S=((x_1,y_1),\ldots,(x_m,y_m))$ be a training sample of m pairs in $\mathcal{X}\times\mathcal{Y}$. A conditional maximum entropy model is a conditional probability of the form $p_{\mathbf{w}}[y|x]=\frac{1}{Z(x)}\exp(\mathbf{w}\cdot\mathbf{\Phi}(x,y))$ with $Z(x)=\sum_{y\in Y}\exp(\mathbf{w}\cdot\mathbf{\Phi}(x,y))$, where the weight or parameter vector $\mathbf{w}\in H$ is the solution of the following optimization problem (we include a regularizer R):

$$\mathbf{w}=\operatorname*{argmin}_{\mathbf{w}\in H} F_S(\mathbf{w})=\operatorname*{argmin}_{\mathbf{w}\in H}\lambda R(w)-\frac{1}{m}\sum_{i=1}^{m}\log p_{\mathbf{w}}[y_i|x_i]. \tag{1}$$

This regularized objective function can then be optimized via gradient based methods:

$$\mathbf{w}_i=\mathbf{w}_{i-1}+\alpha\nabla F_S(\mathbf{w}_{i-1})$$

For very large problems, the benefits of online, stochastic gradient methods [1] are significant – for example when the data cannot be stored in memory during the entire run. With these methods, instead of computing the full gradient $\nabla F_S(\mathbf{w}_{i-1})$, a gradient is computed over a subset $D \subseteq S$, and the update is:

$$\mathbf{w}_i = \mathbf{w}_{i-1} + \alpha \nabla F_D(\mathbf{w}_{i-1})$$

In addition to maximum entropy models, we also consider perceptron learning, where online updates are performed (similar to stochastic gradient updates). In contrast to the maximum entropy case, there is a max-margin type of update performed, where the parameters are only updated when the model is in error (i.e., $\alpha = 1$ and ∇F is the perceptron update vector rather than the true gradient). The perceptron is typically faster since it doesn't compute a partition function.

We present empirical results exploring optimization variants using a standard distributed computing framework. Our distributed computation is based on the MapReduce paradigm [4]. To hold our global model parameters, we use a distributed data-store known as Bigtable [2]. By utilizing these distributed technologies, we provide a modular solution to distributed gradient optimization, requiring only a cluster and the MapReduce/Bigtable infrastructures.

2.1 Distributed Gradient

One option for distributed training is to partition the data into non-overlapping sets and to assign one worker to each partition [3]. At each iteration, every worker computes the gradient over its partition and sends its partial gradient to a master which waits until all workers report. The master then computes the global gradient by summing the gradient from the workers, performs an update step using a traditional gradient descent algorithm, and sends the new parameter to all of the workers. Alternatively, the master can perform an update of the model stored in a distributed data-store (e.g., a Bigtable) and the workers can read items out of the model as needed. While distributed gradient can give a substantial improvement in wallclock speed over non-distributed approaches, it requires significant network communication and cannot take advantage of the fast convergence offered by online stochastic gradient updates.

2.2 Asynchronous Updates

Drawing off work in the early 80s [10], recent work has explored asynchronous training where workers simultaneously update a shared parameter vector [7, 5]. There are no fixed epochs, instead all workers simultaneously operate on a shared parameter vector. This strategy has been investigated heavily for the multicore case (< 16 cores), but scaling up to 100s or 1000s of machines connected over a network has not been experimentally explored. One of the problematic issues for scaling up is that for asynchronous updates the network transmission time for the parameters updates from workers becomes prohibitive. Here we consider variants where the parameters are maintained in Bigtable, thus the entire parameter vector doesn't need to be updated to all workers at each change, but can be cached locally and updated as needed. Even so we find that mini-batches are required, where a number of items (as many as 5k items) are processed at a time before updates are sent to a shared parameter vector.

2.3 (Iterative) Parameter Mixtures

Finally, another set of work considers the case where workers operate independently and only share parameters when their local model is fully optimized, at which point their parameter estimates are averaged [8]. In the case of a convex optimization criterion like maximum entropy, the parameters converges at a comparable rate to the distributed gradient strategy. For the perceptron algorithm, a single average will not converge, but an iterative strategy where at each epoch parameters are averaged and then resent does converge, and in this scenario at exact the same rate as standard perceptron training [9]. We use the iterative version of parameter mixtures for the perceptron and for the maximum entropy model.

3 Cluster Configuration

Many modern datacenters are composed of many commodity machines connected over fast fiber. Massively multicore machines have fallen out of favor recently, perhaps due to their increased cost over commodity machines or the lack of ability to resize easily. So while computers with GPUs and massively multicore CPUs are available, computing on a distributed network has become a crucial practical problem. For this computing infrastructure, MapReduce and Bigtable have recently gained advocates, and there now exists a growing community of support for this platform.

3.1 MapReduce

MapReduce [4] is a distributed computation model, consisting of two main phases, a map phase and a reduce phase. In the map phase, each worker processes each item in isolation and produces an intermediate output consisting of a key-value pair. The reduce phase then collects all outputs with matching keys, and performs an operation on that set of values, to yield a final combined output (or set of outputs) for that key. This computation model has been widely applied to a variety of tasks, including machine learning [3].

There are two details about MapReduce which are especially relevant for our purposes. This first is that there is typically an idempotence assumption made on the processing of particular items which allows backup workers to duplicate effort without producing erroneous output. A fully asynchronous system would violate idempotence since each worker would be constantly updating a shared parameter space. Second, there is typically a sorting phase between the Map and Reduce phases where intermediate outputs with the same keys are matched together.

In our implementation, we reduce the idempotence constraint as much as possible by moving all updates to the Reducer and turning off the sorting phase. The Reducer then waits until a sufficient number of updates have occurred before updating the global model. By turning off the sorting phase, we enable the asynchronous strategies to operate more efficiently, but there is additional complexity in maintaining overall system state. In contrast, the backups cannot be turned off and the optimization must handle inconsistent item multiplicity.

3.2 BigTable

For some training strategies, individual workers can run in isolation and communicate only at the end of epochs [3, 8], but for others more frequent communication is required [7]. Additionally, in some cases, the parameters cannot fit into memory (e.g. for a very sparse multiclass problem). When running in a datacenter and not on a massively multicore CPU (e.g. a GPU), a distributed storage solution is required to address these issues.

Bigtable [2] is a widely used distributed storage system. It is not a database, but instead saves items indexed by row/column/time where the resources for old/stale items can be recovered (e.g. only the most recent n items retained). There are multiple layers of caching, both at the machine level and the network level, so as to allow fast access to relevant portions of the data. Though Bigtable supports transaction processing over the items, this introduces significant lag into processing, since workers may block on Read or Write requests (depending on the configuration), and therefore transaction processing is turned off for the experiments.

4 Experimental Results

For the experiments in this paper, we consider a number of metrics: accuracy of the resulting model (evaluated here by AUC), total CPU time taken during training across the entire network, and observed user wall-clock. The last metric takes into account lag that might be introduced by transmission of information over the network. For evaluation we use click-through data from users on a popular search engine. This data has the nice properties that it is (1) near infinite and (2) potentially useful for practical purposes. The data is extremely sparse and the described systems do not perform any additional dimensionality reduction, but operate on the features as given.

The distributed algorithms under comparison were trained on 370 million training examples containing approximately 8.9 million unique raw features. The dataset was broken into 200 similar-sized

non-overlapping units. We used 240 worker machines to train the models. We evaluate all experiments on a set of 185 million examples of click-through data collected after the training data (these events follow the training data temporally). Table 1 shows the results from the first set of experiments. In all experiments, the algorithms stopped training when the difference in the L^2 norm of

	AUC	CPU (rel.)	Wall Clock (rel.)	Network usage (rel.)
Distributed Gradient	.8323	1.0	1.0	1.0
Single-core SGD	.8557	0.10	9.64	0.03
IPM SGD	.8557	0.12	0.13	0.10
Asynchronous SGD	.7283	2.17	7.50	82.47
IPM Perceptron	.8870	0.16	0.20	0.22

Table 1: Experimental results for the training set of 370 million click-through examples. IPM is the iterative parameter mixtures technique and SGD is stochastic gradient descent algorithm. Times are relative to the exact distributed gradient. Network usage representing the number of bytes passed through the network for all communication (including MapReduce and BigTable) is relative to the usage for the exact distributed gradient.

the parameter vector fell below a specified threshold. The Distributed Gradient algorithm ran for 37 iterations and was only able to achieve an accuracy of .8323 AUC, whereas the iterative parameter mixtures SGD ran for 4 iterations achieving a score of .8557 AUC using far less resources. Running the iterative parameter mixture technique with the perceptron algorithm for 10 iterations achieved an accuracy of .8870 AUC while using less than double the resources of the SGD variant.

	AUC	CPU (rel.)	Wall Clock (rel.)	Network usage (rel.)
Distributed Gradient	.8117	1.0	1.0	1.0
IPM SGD	.8904	0.17	0.15	0.15
Asynchronous SGD	.8525	3.71	14.33	133.93
IPM Perceptron	.8961	0.24	0.20	0.38

Table 2: Experimental results for the training set of 1.6 billion click-through examples. Times are relative to the exact distributed gradient.

Table 2 presents the results of experiments in which the algorithms were trained on a set of 1.6 billion examples from the same click-through dataset; this data is split into 900 non-overlapping units. Up to 600 workers were used to train these models. We do not include the relative times for a single-core SGD as training times are prohibitive on the larger dataset. The gains over the standard Distributed Gradient algorithm in terms of CPU usage and network usage are consistent with those in the smaller dataset.

The most interesting results here is that the Asynchronous SGD performs poorly as compared to the other algorithms. Async SGD makes the most extreme of the approximations of the explored algorithms and is very sensitive to hyper-parameter settings. We ran Asynchronous SGD for 7 iterations, with each worker sending updates to the global model every 1000 examples. Network usage is prohibitive in this case (e.g., the latency to the BigTable causes Wall Clock time to exceed that of any of the explored algorithms).

5 Conclusion

We have compared three popular strategies for distributed gradient training, tuning them on a MapReduce/Bigtable computing platform. The Iterative Parameter Mixtures algorithm consistently provides equally-predictive models and take far less time to train. Asynchronous training proves to be relatively unstable during training and is extremely expensive in terms of CPU usage and network usage. This work demonstrates that the best strategies for the multicore case (asynchronous) do not scale to deployment on a datacenter. Furthermore, we observe that while each of these algorithms is optimizing a convex function, they are sensitive to hyper-parameter selection. In conjunction with the recent theoretical guarantees, our work should help clarify the tradeoffs between the choices in this class of distributed learning algorithms.

References

[1] L. Bottou. Stochastic learning. In O. Bousquet and U. von Luxburg, editors, *Advanced Lectures on Machine Learning*, Lecture Notes in Artificial Intelligence, LNAI 3176, pages 146–168. Springer Verlag, Berlin, 2004.

[2] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable : A distributed storage systems for structured data. *ACM TOCS*, 26:2, 2008.

[3] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In *Advances in Neural Information Processing Systems*, 2007.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *CACM*, 51:1, 2008.

[5] K. Gimpel, D. Das, and N. Smith. Distributed asynchronous online learning for natural language processing. In *CoNLL*, 2010.

[6] J. Lafferty, A. McCallum, and F. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML*, 2001.

[7] J. Langford, A. Smola, and M. Zinkevich. Slow learners are fast. *JMLR*, 2009.

[8] G. Mann, R. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*, 2009.

[9] R. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. In *HLT-NAACL*, 2010.

[10] J. N. Tsitsiklis, D. P. Bertsekas, and M. Athans. Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE Transactions on Automatic Control*, 31:9, 1986.

Appendices

A Distributed Gradient Algorithms

We present the general form of the distributed training algorithms used in this paper. A learning-rate parameter, α, is introduced in each of these algorithms. In many cases a decaying learning-rate, rather than a fixed learning-rate, can improve the stability of the algorithm during learning (and is required for theoretical convergence). We omit the details of learning-rate adjustments. Similarly, we omit the computation of the regularization parameters ($R()$ in Equation 1).

The Distributed Gradient Descent algorithm is simply the standard batch gradient descent algorithm in which we distribute the computation of the gradient. The key insight is that the gradient in these problems is linear (e.g., the gradient of the log-likelihood). Therefore, we can parallelize the computation and sum the results.

Algorithm 1 presents pseudo-code for Distributed Gradient Descent. Recall, the gradient is the sum of the gradients for each example. The inner for-loop can be parallelized and is suitable for the map phase of a MapReduce computation. The second for-loop takes the output of the first parallelized job and summarizes the results for each feature. A reducer of a MapReduce computation can be used to aggregate the results of the map phase. In this case, we will sort the output of the map phase (which is the input to the reduce phase) according to the feature id's.

In Algorithm 2 we present the Iterative Parameter Mixtures training algorithm. This differs from Algorithm 1 in that we perform stochastic (online) updates in each of the parallel computations. In order to aggregate the K learned models at the end of an epoch, the reduce phase averages the updates from each of the map phase distributed optimizations.

Finally, we present a form of the Asynchronous gradient descent algorithm in Algorithm 3. As in Algorithm 2 the distributed workers perform stochastic updates independently. What differs is that each independent optimizer is constantly reading a global model, w_t^*, which is stored in a BigTable. Each worker uses an auxiliary thread to constantly read from the BigTable. When a worker has processed an example it queues the update to be sent to the **GlobalUpdateReceiver**. The **GlobalUpdateReceiver** is a set of workers in the cluster processing input as it becomes available (e.g., using a non-sorting reducer phase). When data is available, they aggregate updates for each

Algorithm 1 Distributed Gradient Descent

{Break S into K partitions}
$S = \{D^1, \ldots, D^j, \ldots, D^K\}$
$t = 0$
repeat
  $t = t + 1$
  for $j = 1 \ldots K$ **do** {Process in parallel.}
    $\nabla_0^j = 0$
    for $i = 1 \ldots |D^j|$ **do**
      $\nabla_i^j = \nabla_{i-1}^j + \alpha \nabla F_{D_i^j}(w_{t-1})$
    end for
    $\nabla^j = \nabla_{|D^j|}^j$
  end for
  for $l = 1 \ldots |w|$ **do** {For each feature in the model (can be done in parallel).}
    $w_t(l) = w_{t-1}(l) + \sum_j \nabla^j(l)$
  end for
until converged

Algorithm 2 Iterative Parameter Mixtures

{Break S into K partitions}
$S = \{D^1, \ldots, D^j, \ldots, D^K\}$
$t = 0$
repeat
  $t = t + 1$
  for $j = 1 \ldots K$ **do** {Process in parallel.}
    $w_0^j = w_{t-1}$
    for $i = 1 \ldots |D^j|$ **do**
      $w_i^j = w_{i-1}^j + \alpha \nabla F_{D_i^j}(w_{i-1}^j)$
    end for
    $\Delta^j = w_{|D^j|}^j - w_{t-1}$
  end for
  for $l = 1 \ldots |w|$ **do** {For each feature in the model (can be done in parallel).}
    $w_t(l) = w_{t-1}(l) + \frac{1}{K} \sum_j \Delta^j(l)$
  end for
until converged

Algorithm 3 Distributed Asynchronous Training

{let B be the size of the update batch}
{Break S into K partitions}
$S = \{D^1, \ldots, D^j, \ldots, D^K\}$
$t = 0$
repeat
  $t = t + 1$
  for $j = 1 \ldots K$ **do** {Process in parallel.}
    for $i = 1 \ldots |D^j|$ **do**
      $w^j \leftarrow w_{t-1}^*$ {Asynchronous read from global model}
      $w_i^j = w^j + \alpha \nabla F_{D^j}(w^j)$
      $\Delta^j = w_i^j - w^j$
      GlobaleUpdateReceiver(Δ^j) {Cache up to B items and send asynchronously}
    end for
  end for
until converged

Algorithm 4 GlobalUpdateReceiver

repeat
  for $l = 1 \ldots |w|$ **do** {For each feature update (can be done in parallel).}
    $w'(l) = w(l) + \sum_m \Delta^m(l)$ {m updates for this feature}
  end for
  $w^* \leftarrow w'$ {Asynchronous update to global model}
until training epoch completed

feature, and send the updates to the global model w_t^*. Although it is not necessary to maintain the notion of *epoch* in this setting, it becomes a useful construct for checking for convergence.