MULTICORE LEARNING ALGORITHM

CHENG-TAO CHU, YI-AN LIN, YUANYUAN YU

1. Summary

The focus of our term project is to apply the map-reduce principle to a variety of machine learning algorithms that are computationally expensive. Instead of using expensive computer clusters, we focus on implementing the framework on multi-core computer environment. On top of that, in order to apply the framework to a variety of modern machine learning algorithms, we focus on parallelize the summation operation in the algorithms by developing customized mappers and reducers. In the following sections, we will show the simplified architecture in Section 2, the implemented algorithms and the formula where the framework is applied in Section 3, and finally the experiments and its result disccision in Section 4.

2. Architecture

There is a list of well-known machine learning algorithms which require a huge amount of independent training examples. To expedite the training process, we will implement a framework which distributes the training operation (majority of which is actually summation) to different threads/processes. The original map reduce paper [1] from Google has outlined a full-blown framework that is specialized in this type of divide and conquer works. Using the same set of principle from [1], our goal is to implement a relatively lightweight architecture that focuses on multi-core environment and provides a flexible interface. The interface should be easily adapted to different learning algorithms, while achieving decent boost in efficiency (i.e. we hope to witness a n-fold boost in a n-core environment).

Figure 1 shows a high level view of the architecture and how it processes the data. In step 0, the map-reduce engine is responsible for splitting the design matrix by samples (rows). The engine then caches the splitted data for the following map-reduce invokes. Every algorithm has its own engine instance, and every map-reduce task will be delegated to the engine (step 1). Similar to the original map-reduce architecture, the engine will run a master (step 1.1) which coordinates the mappers and the reducers. The master is responsible for assigning the splitted data to different mappers, and then collects the processed intermediate data from the mappers (step 1.1.1 and 1.1.2). Right after the intermediate data is collected, the master will in turn invoke the reducer to process them (step 1.1.3). Please notice that some mapper and reducer

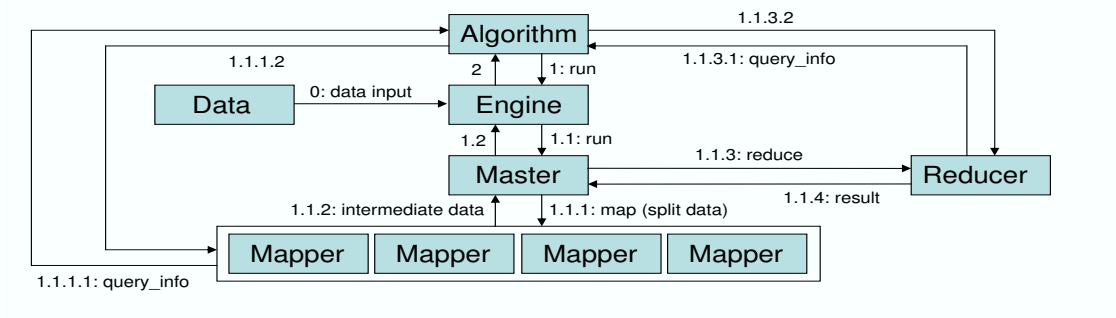

Figure 1. Adopted map-reduce framework

operations requires additional information from the algorithm, in order to support these operations, the algorithm can pass these information through the query_info interface (step 1.1.1.1 and 1.1.3.2).

3. Adopted Algorithms

As mentioned in Section 1, the framework is applied to parallelize operations in the algorithms. Currently, Locally Weighted Linear Regression, Naive Bayes, Gaussian Discriminative Analysis, K means, Logistic Regression, Neural Network, EM, Independent Component Analysis, Principal Component Analysis, are implemented in the system and the detailed description of where the operations parallelized are shown below.

- **Locally Weighted Linear Regression**
  The LWLR [6] problem can be solved by using normal equation $a\theta = b$ where $A = \sum_{i=1}^{m} w_i(x_i x_i^T)$ and $b = \sum_{i=1}^{m} w_i(x_i y_i)$. In summation form, we divide the process among different mappers and customize two sets of mappers/reducers for computing A and b respectively. Since we know that A is a feature by feature matrix and b is a feature by one matrix, as long as the amount of samples is significantly larger than the number of features, the framework would enhance the performance by the parallelization.
- **Naive Bayes**
  In the Naive Bayes algorithm [2], we have to calculate the parameters $\phi_{x_j=k|y=1}$, $\phi_{x_j=k|y=0}$ and ϕ_y from the training data. In order to find the ϕs, we need to sum over $x_j = k$ and y from training data to calculate $P(x|y)$. As in the summation form, we can divide the labor using our map-reduce framework among multi processors to expedite the summation process. The reducer then sums up all the intermediate results and get final maximum likelihood for prediction.
- **Gaussian Discriminative Analysis** Classic GDA [2] algorithm needs to "learn" the following four parameters ϕ, μ_0, μ_1 and Σ. For all the summation form involved in these computation, we may leverage the map-reduce framework to parallerize the process. Each mapper will handle the summation (i.e. $\Sigma_1 y_i = 1, \Sigma_1 y_i = 0, \Sigma_1 y_i = 0 * X_i$, etc) for part of the training samples, that is, suppose we split the training set into n pieces, the framework will typically (though not necessarily) launch n mappers, each of which handles one piece of the data. Finally, the reducer will aggregate the intermediate sums and calculate the final result for the parameters.
- **K-Means** K-Means is a very common unsupervised learning algorithm. The goal is to cluster the training samples into k virtual categories. The algorithm iteratively computes the centroids of every cluster and reassigns each sample to its closest cluster. This process continues until it converges or reaches a preset maximum number of training rounds. It is clear that the operation of computing the Euclidean distance between the sample vector and the centroids can be parallerized by splitting the data into individual blocks and clustering samples in each block separately (by the mapper). Then, the reducer will collect the mapper's work and recompute the centroids for each cluster.
- **Logistric Regression** Approaching a classification problem using logistic regression [6], we choose the form of hypothesis as $h_\theta(x) = g(\theta^T x) = 1/(1 + \exp(-\theta^T x))$ The learning is done by training θ to classify the data, and the likelihood function can be optimized by using gradient ascent rule. We choose a batch gradient ascent rule so the summation form $\theta = \theta + \alpha(\sum(y(i) - h_\theta(x(i))x_j(i))$ can be easily parallelized using our map-reduce framework.
- **Neural Network** Neural Network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target function. Our work focus on the back propagation algorithm, which uses gradient ascent to tune network parameters to best fit a training set of input-output pairs. By defining a network structure (we use a 3 layer network with 2 output neurons classifying the data into 2 categories), each mapper propagates its set of data through the network. for each data, the error is back propagated to calculate the gradient for each of the weights in the network. The reducer then sums the partial gradient from each mapper and does a batch gradient ascent to update the weights of the network. In spite that back propagation algorithm is usually done by stochastic gradient ascent, the batch gradient ascent also converges and performs fairly well.

- **Principal Component Analysis** Principal components analysis [5] bascically tries to identify the subspace in which the data set lies. The "principal components" are essentially the "most dorminant" vectors that span this subspace. Mathematically, we can prove that the principal eigenvectors of the empirical covariance matrix of the data are exactly what we want. Looking at the empirical covariance matrix: $\frac{1}{m}\sum_{i=1}^{m} x^{(i)} * x^{(i)^T}$, it is clear that we can parallelize the computation of this matrix. That is, by dividing the training set into smaller groups, we can compute $\sum_{subgroup}(x^{(i)} * x^{(i)^T})$ for each group using a separate mapper, and then the reducer will sum up the partial results to produce the final empirical covariance matrix.
- **Independent Component Analysis** Unlike PCA, which find the "most dominant" vectors, ICA tries to identify the independent sources vectors based on the assumption that the observed data are linearly transformed from the source data. Refer to [3] for the "cocktail party problem" as a classic motivating example (and one of the most important practical application of ICA). The main goal is to compute the unmixing matrix W. In [3], the stochastic gradient descent was adopted to maximize the likelihood function. However, that will pose additional challenge as we will discussed that in the end of the section. We implement batch gradient descent instead to optimize the likelihood. In this scheme, we can independently calculate the $\begin{bmatrix} 1 - 2g(w_1^T x^{(i)}) \\ \vdots \end{bmatrix} x^{(i)^T}$ in the mappers and sum them up in the reducer.
- **Expectation Maximization** EM [4] is typically a two-phase algorithm. In our implementation, we use Mixture of Gaussian as the underlying model. For parallelization: In E-phase, every mapper just process its subset of the training data and compute the corresponding $w_j^{(i)}$ (expected psuedo count). In M-phase, three sets of parameters needs to be updated: ϕ, μ, and σ. For ϕ, every mapper will compute $\sum_{subgroup}(w_j^{(i)})$, and the reducer will sum up the partial result and divide in by m. For μ, each mapper will compute $\sum_{subgroup}(w_j^{(i)} * x^{(i)})$ and $\sum_{subgroup}(w_j^{(i)})$, and the reducer will sum up the partial result and divide them. For σ, every mapper will compute $\sum_{subgroup}(w_j^{(i)} * (x^{(i)} - \mu_j) * (x^{(i)} - \mu_j)^T)$ and $\sum_{subgroup}(w_j^{(i)})$, and the reducer will again sum up the partial result and divded them.

It turns out that many Machine Learning algorithms, e.g. ICA, that use stochastic gradient descent all pose an interesting challenge for parallelization of their algorithms. The problem is that in every step of the gradient descent, the algorithm updates a common target, e.g. the unmixing W matrix in ICA. This becomes a prime example of the "race-condition" scenario in many multithreading applications. That is, when one gradient descent step (involving one training sample) is updating W, it has to lock down this matrix, read it, compute the gradient, update W, and finally release the lock. This "lock-release" block is essential and inevitable in order to prevent corrupted or outdated W from being used. On the other hand, however, this clearly creates a fatal bottleneck for the parallelized algorithm.

We believe that parallelizing this type of machine learning algorithm that involves procedures like stochastic gradient descent will be an interesting topics to investigate. This is certainly one of our main concerns for future work.

4. Experiments

To compare the performance improvement, we implement each algorithms with two versions, one equipped with the map-reduce framework and the other run without the framework. To ensure the faireness, both versions of an algorithm use the same way to load the data and the same algorithm to train the classifier. In the experiment, the machine, cs229-1, we used has two Intel Pentium III CPU 700 GHz and 1GB memory and is installed with Linux RedHat 7 Kernel 2.4.20-smp.

Table 1 lists the performance difference of each algorithm run on cs229-1. In the experiments, we used a 1000000x14 desgin matrix for the LWLR, GDA, NB, PCA. For K-Means and Logistic Regression, we used a 100000x14, for NN, we used a 10000x14, and for EM, we used a 32562x14, design matrix in order to constrain the running time of the experiment. For ICA, we used a 53442x5 data matrix. The tables show the average

	LWLR	GDA	NB	K-Means (10 iters)
original	14.29, (99.80%)	16.55, (99.78%)	16.35, (99.84%)	76.13, (99.83%)
map-reduce	9.87, (161.08%)	9.33, (160.28%)	10.26, (161.99%)	50.55, (159.90%)
improvement	144.78%, (161.40%)	177.38% (160.63%)	159.35%, (162.24%)	150.60%, (160.17%)

Logistic	NN	PCA	ICA	EM
10.30, (99.69%)	254.29, (99.68%)	19.80, (99.89%)	81.09, (99.73%)	15.67, (99.90%)
6.88, (160.35%)	143.63, (189.94%)	13.92, (144.14%)	47.27, (181.40%)	8.77, (177.72%)
149.70%, (160.84%)	177.08%, (190.54%)	142.24%, (144.29%)	171.54%, (181.89%)	178.59%, (177.91%)

TABLE 1. Performance comparison run on cs229-1

time taken to run and the average cpu usage for each algorithm with and without the map-reduce framework. For each algorithm, we run it ten times and get the average statistics. In particular, for the cpu usage, it is actually the real running time divided by the summation of the time spent in both the system and the user space.

As shown in Table 1, we can see the running time of all the algorithms improve by factors from 142.24% (PCA) to 177.08% (NN). As only the summation in each algorithm is parallelized, in terms of the level of parallelization, the improvement varies as we expected, e.g. in PCA, to solve SVD decreases the level of parallelization and in LWLR, to solve the normal equationn also limits the improvement. On the other hand, NN almost doubles its performance by adopting the framework. In addition, if we examine the cpu utilization rate, it is obvious that further parallelization is possible. In particular, PCA only benefits from the framework by 144.29% cpu utilization. Although it is still a significant improvement, if we further parallelize SVD, we could expect a even better performance.

The benefit of parallelizing machine learning algorithms is significant as shown in the experiments. NN, in particular, has a nearly double performance boost up. By adopting the map-reduce framework, the algorithms can be easily extended by parallelized operations. Although in our implementaion, we only parallelize the summation for the sake of simplicity. In addition, applying the framework on a multi-core machine further eliminates tremendous communication overhead as in [1], it takes around one minute to startup the framework. Moreover, theoretically, as the number of cores gets larger, we can expect even more performance enhancement and we would like to treat that as a future work. As the multi-core computers are getting more and more popular, we believe a better architecture which exploits this parallel computation environment would attract more attention and play an important role in the machine learning field.

5. Future works

In only one quarter, what we can implement is pretty constrained. Therefore, we leave the followingas the future works.

- SVM and Pegasus parallelization.
- Experiments on computers with more than two processing units.

6. Acknowledgement

The project is supervised under professor Andrew Ng, many thanks to him for his enthusiasm and devotion in supporting us through the process. We would also like to thank Gary Bradski for the knowledge in parallelization implementation and the experimental environments. Finally, we would like to thank Skip Macy for sharing his valuable profiling experience in Vtune.

References

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplifed data processing on large clusters. *Operating Systems Design and Implementation*, pages 137–149, 2004.
[2] Andrew Y. Ng. Generative learning algorithms. `http://cs229.stanford.edu/`.
[3] Andrew Y. Ng. Independent components analysis. `http://cs229.stanford.edu/`.

[4] Andrew Y. Ng. Mixtures of gaussians and the em algorithm. http://cs229.stanford.edu/.
[5] Andrew Y. Ng. Principal components analysis. http://cs229.stanford.edu/.
[6] Andrew Y. Ng. Supervised learning. http://cs229.stanford.edu/.

E-mail address: chengtao@stanford.edu, ianl@stanford.edu, yuanyuan@stanford.edu