Making the most of a distributed perceptron for NLP

Max Whitney and **Ann Clifton** and **Anoop Sarkar** and **Alexandra Fedorova**
Simon Fraser University
Burnaby, British Columbia, Canada
{max_whitney,ann_clifton,anoop_sarkar,alexandra_fedorova}@sfu.ca

1 Introduction

The perceptron algorithm (Rosenblatt, 1958), in particular the global linear model of Collins (2002), has been employed to handle NLP tasks such as part-of-speech tagging, parsing, and segmentation. Many such tasks require vast training data to create robust models, which is very time-consuming and computationally intensive. It is natural to look to distributed systems to speed up training for these tasks.

We implemented distributed perceptron training following McDonald et al. (2002) using MapReduce (Dean and Ghemawat, 2008), a simple and common framework for distributed computation. McDonald et al. (2002) also use MapReduce but do not discuss their implementation or evaluation of choices involved. We achieved similar results to that work in that our implementation is faster than the serial algorithm and also achieves higher accuracy. Chiang et al. (2008) use a parallel setting for training the online learner MIRA (Crammer and Singer, 2003), but do not discuss parameter combining topologies.

We make two novel contributions: We examine two topologies for the combination of separately trained weight vectors, finding that the choice which duplicates computation leads to a shorter runtime. We also vary the number of data shards and observe a tradeoff between runtime and the final testing accuracy. Unlike previous work, we ran our system both on an HPC cluster and on single multi-core systems, finding similar results on both. We report cache miss statistics for the multi-core setting. These findings promise to help to make effective use of the distributed perceptron for NLP applications.

2 Algorithm and MapReduce

Algorithm 1 shows the distributed perceptron of McDonald et al. (2002). OneEpochPerceptron$(T_i, \mathbf{w})$ is one iteration of standard perceptron training on data T_i with initial weights $\mathbf{w}$. The data is split into S separately processed shards. The new weight vectors (from training on each shard) are combined into a single new weight vector for the next iteration. McDonald et al. (2002) show that this has the same convergence guarantees as the serial version.

Algorithm 1 Distributed perceptron

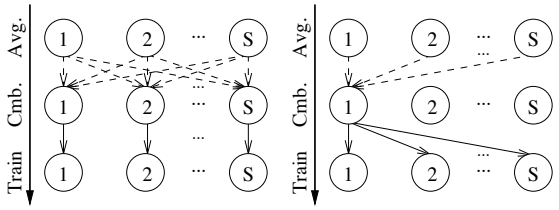
Require: training data $T = \{(\mathbf{x}_t, \mathbf{y}_t)\}_{t=1}^{|T|}$
1: shard T into S pieces $T_1, \ldots, T_S$
2: let $\mathbf{w}$ be the zero weight vector
3: **for** iteration $n : 1..N$ **do**
4: $\quad \mathbf{w}^{(i)} =$ OneEpochPerceptron$(T_i, \mathbf{w})$
5: $\quad \mathbf{w} = \frac{1}{S} \sum_i \mathbf{w}^{(i)}$
6: **end for**
7: return $\mathbf{w}$

MapReduce represents data as a distributed collection of key-value pairs. For the training step (line 4) each of S processes has a copy of the complete weight vector. We use a map operation so that each process trains on a different shard, then outputs new weights as separate feature-weight pairs. The averaging step (part of line 5) does a reduce operation on these pairs to average the feature weights in parallel.

Starting the training step of the next iteration requires combining the distributed feature-weight pairs into complete weight vectors. We call this the combine step, and consider two methods: In single-combine, the feature-weight pairs are all sent to a single process, which makes a single weight vector and sends a copy to each other process. In multi-combine, each feature-weight pair is duplicated $S-1$ times so that each process gets a copy and produces a combined weight vector. Figure 1 shows the communication patterns for the two methods.

3 Experimental setup

We ran our experiments on an HPC cluster and on two multi-core systems (not in the cluster). The cluster consists of 64-bit x86 machines (the precise number of nodes used varied in our experiments), which we could not isolate from the load of other users. The multi-core systems were both AMD Opteron systems with 24 cores over 4 chips. The first was used for exclusive testing; the second



(a) Multi-combine (b) Single-combine

Figure 1: Transmission of weights between processes through the averaging, combine, and train steps. Dashed lines represent individual weights and solid lines represent complete weight vectors.

was used for detailed bottleneck testing via hardware counters. On the cluster, the processes were distributed across multiple machines; on the exclusive systems the processes were on different cores.

We used the `MapReduce-MPI`[1] library running on MPI. We used the `mpi4py`[2] library for other MPI coordination. Besides the S processes described in section 2 we used one extra process for logging.

We tested on a chunking task using the CONLL-2000 standard data and test set [3]. We present only a single repetition of each experiment here, but we did at least 2-3 repetitions of each experiment and found the results to be consistent on the points presented. In our experiments here we sharded the data by taking equally sized sections. The data was made available to all processes on a network file system.

4 Convergence and comparison to serial

To evaluate accuracy and runtime, we ran the parallel algorithm on the cluster with the number S of shards as 5, 10, 15, and 20, and compared against the serial algorithm. In each case the parallel algorithm not only significantly lowers the runtime for 15 iterations, but also improves on the accuracy of the serial algorithm. McDonald et al. (2002) also note this increase in accuracy, and suggest that it is due to the averaging effect of combining weight vectors from each shard. We additionally note that there appears to be a tradeoff between runtime and accuracy as the number of shards varies.

We also ran this experiment on the cluster with different numbers of cores per machine, and on the first exclusive machine, obtaining similar results in each case.

5 Combine step topologies

Both combine methods duplicate and transmit the same number of weights, but with different granularity and communication patterns. To compare them we ran both on the cluster with 5, 10, 15, and 20 shards. Multi-combine is faster in each case.

We believe that this is due to single-combine creating a bottleneck at the combining process. On the cluster we were unable to make bandwidth measurements, so we used the second exclusive system (see section 3) to evaluate this hypothesis. Here MPI used the multi-core environment directly (not through the network). The L3 CPU cache is per-chip, while L1 and L2 are per-core. The L3 miss rate is thus an estimate of traffic between chips, so we used it as an equivalent to network bandwidth.

Running both methods for 15 iterations with 15 shards, multi-combine finished in less than half the time of single-combine. We found that multi-combine maintained a similar miss rate for each process, but with single-combine one process had a much higher miss rate than the others. Moreover, the bandwidth peaks for single-combine were (very roughly) twice as high and twice as long as those for multi-combine. This suggests that the system is indeed bottlenecked on the single combining process.

[1] http://www.sandia.gov/∼sjplimp/mapreduce.html

[2] http://mpi4py.scipy.org/

[3] http://www.cnts.ua.ac.be/conll2000/chunking/

References

David Chiang, Yuval Marton, and Philip Resnik. 2008. Online large-margin training of syntactic and structural translation features. In *EMNLP*.

Michael Collins. 2002. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *EMNLP*.

Koby Crammer and Yoram Singer. 2003. Ultraconservative online algorithms for multiclass problems. *Journal of Machine Learning Research*, 3:951–991.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113.

Ryan McDonald, Keith Hall, and Gideon Mann. 2002. Distributed training strategies for the structured perceptron. In *NAACL*.

Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.