Parallel Unsupervised Feature Learning with Sparse AutoEncoder

Milinda Lakkam, Siranush Sarkizova
{milindal, sisis}@stanford.edu

December 10, 2010

1 Introduction and Motivation

Choosing the right input features for a particular machine learning algorithm is one of the deciding factors for a successful application but often a time consuming manual task. Unsupervised feature learning algorithms provide an alternative solution by automatically learning features; however, they are usually computationally intensive. In this project we explore different implementations of the sparse autoencoder (SAE) learning algorithm, in terms of development platform as well as optimization approach, with the objective of creating a cost- and time-efficient user-friendly parallel implementation. In particular, we first focus on comparing MATLAB and Python as implementation frameworks and stochastic gradient descent, nonlinear conjugate gradient and L-BFGS as optimization techniques. We then evaluate two parallel implementations based on custom parallel frameworks in MATLAB and Python tailored towards similar algorithms.

2 Algorithms Overview

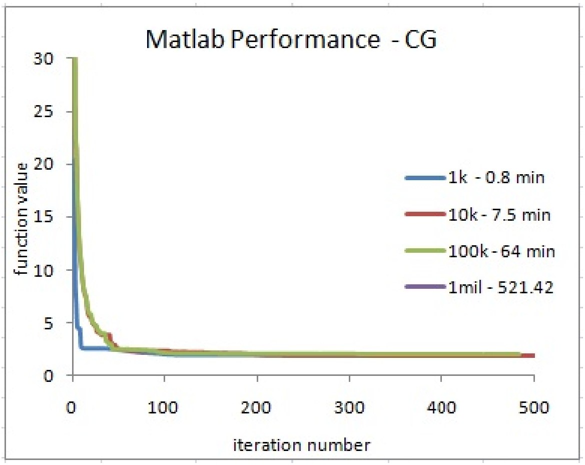
The sparse autoencoder is a self-thought algorithm for learning features derived from unlabeled data. At its core, an autoencoder is simply a supervised learning algorithm, based on neural networks, which tries to learn the identity function i.e. it uses $y^{(i)} = x^{(i)}$ to learn $h_{W,b}(x) \approx x$. A sparse autoencoder is an autoencoder which further tries to minimize the number of active hidden units in the neural network by imposing a sparsity constraint [2]. Given a neural network architecture, a set of unlabeled training examples $\{x^{(i)} \in \mathbb{R}^n, i = 1, \ldots, m\}$ and a cost function $J(W,b;x)$, we train the neural network by performing a minimization of the cost function $J(W,b;x)$, which for the sparse autoencoder is given by:

$$J(W,b;x) := \left(\frac{1}{m} \sum_{i=1}^{m} J(W,b;x^{(l)}) \right) + \lambda_W \sum_j \left\| W_j \right\|^2 + \lambda_{sparsity} \sum_j KL(p^* \| p_j)$$

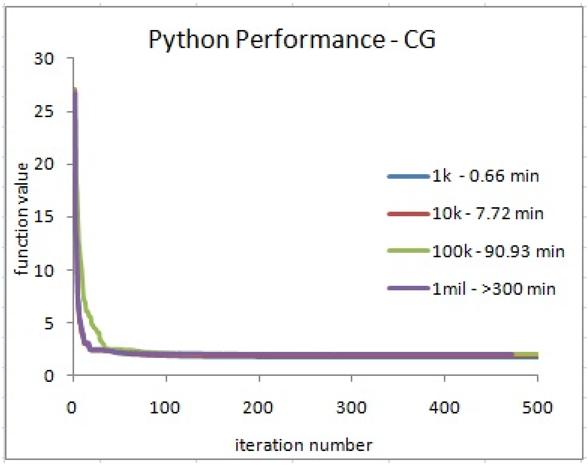
The minimization can be done online (stochastic gradient descent) or offline (using the classical numerical optimization techniques such as the Conjugate Gradient or the L-BFGS). The CG algorithm is a modified SD technique with the successive descent directions chosen to be conjugate to preceding directions and an accurate line minimization is performed along each search direction. The LBFGS algorithm is a quasi-Newton method where gradient information from successive iterations are used to build an approximate Hessian [3].

3 Implementation

We approached the SAE algorithm via a stochastic gradient descent implementation in both MATLAB and Python. As expected, this algorithm ran fast, as it only considers one training example at a time to make an update, but it is not readily parallelizable because every subsequent update is based on the one self contained operation. In view of achieving a parallel implementation, we then trained the neural network using nonlinear conjugate gradient (CG) and L-BFGS to solve the minimization problem. We saw that the line search component of these algorithms takes the majority of the run time as it involves numerous evaluations of the cost function and its gradient. Therefore, we achieved significant speedup in gradient evaluation by finding and using the analytical, rather than numerical gradient. It turned out that the analytical gradient of the cost function, as well as the cost function itself, are summation over all training examples and hence the entire algorithm easily lends itself to parallelization. With this motivation, we used the scipy.optimize optimization package to implemented Python versions of both methods and gauged their performance with benchmark tests against the corresponding MATLAB minFunc implementations. We saw that Python's performance is comparable to MATLAB but also that, for large training sets, L-BFGS is significantly faster than CG as summarized in Figure 1 and Figure 2.

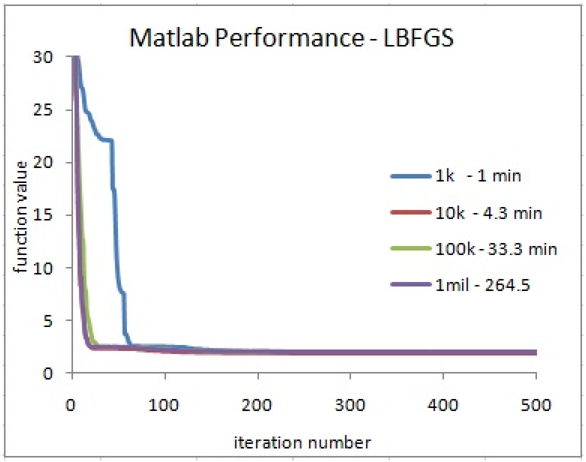


(a) Matlab performance for CG

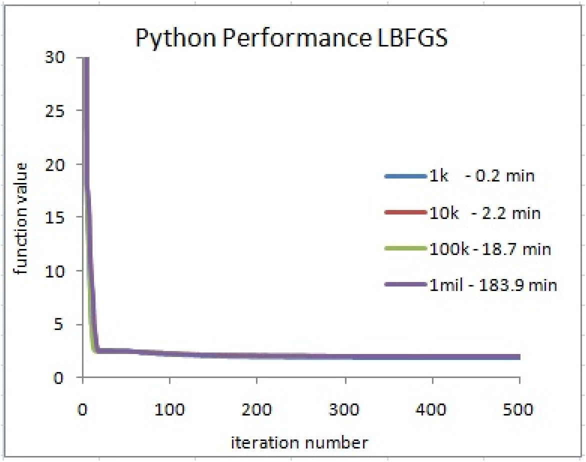


(b) Python performance for CG

Figure 1: MATLAB (a) vs Python (b) performance of CG.



(a) Matlab performance for L-BFGS



(b) Python performance for L-BFGS

Figure 2: MATLAB (a) vs Python (b) performance of L-BFGS.

4 Parallelization

After determining the advantageous performance of LBFGS we proceeded to parallelize the algorithm using the parallel MATLAB framework developed by Adam Coates. The framework is a realization of the map-reduce concept and is tailored towards distributing an algorithm over a large number of nodes with seamless support of data locality. A similar framework in Python was developed by another part of the project group. It will be valuable to asses the performances of the two custom frameworks against a general distributed application framework and we are thus currently working on a Hadoop implementation. We note that we did not initially focus on such standard frameworks since the overhead of adapting our algorithm to run on these systems could be high.

5 Results

The performances of the two parallel implementations was tested with a varying number of workers as well as training examples. We used the Stanford corn cluster for all testing and we note that the results might be slightly skewed due to differences in machine configurations. Table 1 and Table 2 provide a summary of the MATLAB and Python run times respectively. Overall with MATLAB , we achieve speedup in most cases (except for 1K) but, as expected, it is more significant when the number of training examples is larger. The speedup for 100K and 1 million training examples going from 2 to 4 and 4 to 8 workers is close to a factor of two which is a promising result.

Similarly, with Python we achieve speedup for the larger number of training examples. However, with 100K the speedup factor is close to two only between 2 and 4 nodes and lower between 4 and 8 nodes. Further testing is necessary to determine whether this is due to the framework or due to the testing environment.

Workers	1K	10K	100K	1million
1	51	120	739	Java OutOfMemoryError
2	50	98	495	Java OutOfMemoryError
4	66	88	423	2948
8	75	85	229	1494

Table 1: Custom Parallel Matlab Framework - Total running time in seconds for different number of workers and training examples.

Workers	1K	10K	100K
1	76	370	5030
2	92	170	2350
4	137	173	1253
8	275	270	914

Table 2: Custom Parallel Python Framework - Total running time in seconds for different number of workers and training examples.

6 Conclusions and Future Work

In conclusion, we have seen that Python is a feasible development platform as compared to Matlab . Furthermore, amongst readily parallelizable algorithms for neural network training, L-BFGS is more favourable than CG in terms of speed. Parallelizing an efficient implementation of L-BFGS, yields promising results in the setting of both Matlab and Python parallel frameworks. In order to get a more in-depth understanding of performance numbers and trade-offs we would like to run additional tests on standardized machines and with other, possibly more sophisticated, algorithms. We are also interested in testing the parallel implementation against a widely accepted system and are thus currently building a Hadoop implementation.

Acknowledgements

Thanks to prof. Andrew Ng for the opportunity to work on this project and his guidance throughout the quarter. Thanks to Adam Coates for the introduction and code of the parallel Matlab framework. And thanks to everyone in the extended project gourp (in no particular order) - Bobby Prochnow, Abhik Lahiri, Raghav Pasari, Matt Sparks, Quinn Slack, Ali Yahya, Juan Batiz-Benet.

References

[1] Ng, Andrew, "CS294A Lecture Notes: Sparse autoencoder.", Stanford University, Nov 10, 2010, available at `http://www.stanford.edu/class/archive/cs/cs294a/cs294a.1104/`

sparseAutoencoder.pdf.

[2] Boyd, Stephen et. al., "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers.", Stanford University, WORKING DRAFT-November 19, 2010, available at http://www.stanford.edu/~boyd/papers/pdf/admm_notes_draft.pdf

[3] Nocedal, Jorge Wright,Stephen J., "Numerical Optimization", Springer Verlag, Second Edition, available at http://users.eecs.northwestern.edu/~nocedal/book/