Distributed Deep Learning
Parallel Sparse Autoencoder

Abhik Lahiri
Raghav Pasari
Bobby Prochnow

December 10, 2010

1 Introduction

Much of the bleeding edge research in the areas of computer vision, natural language processing, and audio recognition revolves around the painful and time consuming process of hand-picking features from training data. Many researchers spend decades experimenting with complex feature selection processes in the hopes of improving the performance of learning algorithms.

Deep learning approaches attempt to replace the practice of hand-picking features by instead algorithmically determining structure or organization hidden within the training data. Some early deep learning approaches have shown great promise - even outperforming many of the state-of-the-art algorithms that operate on hand-picked features.

Deep learning algorithms, however, are computationally expensive. Even on powerful computers, it can be impractical to have the algorithms learn on a sufficient amount of input data, making these algorithms considerably less practical for many problems. Parallelizing these algorithms and running them in a multi-core or distributed setting could result in a significant speedup. This, in turn, makes it more practical to feed larger amounts of data into the algorithms - which will improve their performance considerably.

Thus, our goal is to understand how to scale deep learning methods to function on large clusters with many cores and machines. For the extent of this paper, we focused on parallelization of the sparse autoencoder learning algorithm. Towards this goal, we first did a survey of serial optimization algorithms for the sparse autoencoder (stochastic gradient descent, conjugate gradient, L-BFGS). We then parallelized the sparse autoencoder using a simple approximation to the cost function (which we have proven is a sufficient approximation). Finally, we performed small-scale benchmarks both in a multi-core environment and in a cluster environment.

2 Serial Sparse Autoencoder

The sparse autoencoder is a deep learning variant of a neural network used to represent the identity function on unlabeled training data. To force the network to find structure in the data, we enforce a sparsity constraint that ensures that each of the hidden nodes fires very infrequently over the course of the training set.

2.1 Stochastic Gradient Descent

Our naive approach used stochastic gradient descent to optimize the standard cost function:

$$\tfrac{1}{2}(h(x^{(i)}) - x^{(i)})^2 + \tfrac{\lambda_W}{2}\sum_l \sum_i \sum_j (W_{ji}^{(l)})^2$$

Additionally, to enforce sparsity, after each iteration of stochastic gradient descent, we performed the following update on the biases for the hidden layer:

$$b_i^{(1)} := b_i^{(1)} - a\beta(\hat{\rho}_i - \rho)$$

where $\hat{\rho}_i$ is a running estimate of the probability of the hidden node i firing and ρ is the desired sparsity.

2.2 Batch Optimization Algorithms

For optimization algorithms that iterate on entire batches of examples (L-BFGS and conjugate gradient), we integrate the sparsity constraint directly into the cost function and use the kL divergence to measure the difference between the current and target sparsities:

$$\tfrac{1}{2m}\sum_{i=1}^{m}(h(x^{(i)}) - x^{(i)})^2 + \frac{\lambda_W}{2}\sum_l \sum_i \sum_j (W_{ji}^{(l)})^2 + \lambda_\rho \sum_j KL(\rho || p_j)$$

where ρ is the desired sparsity and p_j is the current sparsity for hidden node j over the entire batch of examples.

2.3 Comparison of Algorithms

In all benchmarks, the training examples are a random sampling of 8x8 patches from a set of ten 512x512 images (courtesy of Bruno Olhaussen). We restrict the hidden layer of the network to 30 nodes, set $\lambda_W = .002$, set $\lambda_\rho = 4$, and target the probability of a hidden node firing to be .002.

On image input, we expect the learned weights for the hidden nodes of the sparse autoencoder to represent edges of independent orientation.

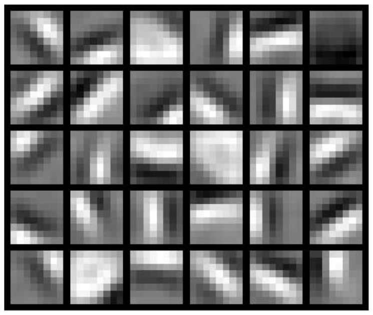

Figure 2.3.1. Sample learned hidden weights

Quantifying how close learned weights were to this goal is difficult - as extremely small differences in the value of the cost function, sparsity, or error can result in highly varied change with respect to how "edge-like" the learned weights are. For our purposes, however, it sufficed to quantitatively analyze how well the algorithms do with respect to minimizing the cost function (and then as a sanity check, visualize the hidden weights to verify the expected output).

In practice, quality output is achieved through 4 million iterations of stochastic gradient descent, or 500 iterations of L-BFGS/conjugate gradient with a 100K batch size. Below are the averaged results for 3 independent trials.

Mean Execution Time (seconds)	
Stochastic Gradient Descent	648
Conjugate Gradient	1098
L-BFGS	653

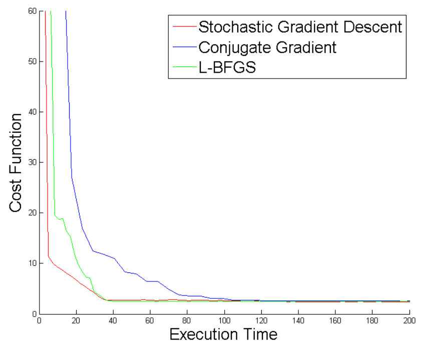

Figure 2.3.2. Average cost function over time

(To calculate the cost function for stochastic gradient descent, we calculated the batch cost function on a set of 100k examples every 8000 iterations).

While stochastic gradient descent performs well in this case, it does not lend itself to much parallelism, as iterations must be performed in sequence, and each iteration is extremely cheap. Both L-BFGS and conjugate gradient operate on batches of examples, allowing for potential parallelism; however, conjugate gradient takes about twice as long as L-BFGS to learn the autoencoder. For this reason, we chose to use L-BFGS in our following parallel implementation.

3 Parallel Sparse Autoencoder

Our parallel algorithm is quite natural - we use a serial implementation of L-BFGS with a parallel cost function:

```
proc ParallelAvgCostFunction($\vec{W}, \vec{X}$) $\equiv$
  foreach t parallel do
    $\vec{X}_t$ := GetThreadData($\vec{X}, t$);
    cost$_t$, grad$_t$ := SerialCostFunction($\vec{W}, \vec{X}_t$);
  end
  avgCost := average(... cost$_t$ ...);
  avgGrad := average(... grad$_t$ ...);
  return avgCost, avgGrad;
.
```

This assumes the $\vec{X}_t$ is the same size for all t - so that all examples are considered with the same weight (when the $cost_t$ values are averaged), but it's not difficult to account for cases where this does not hold. Also note that the algorithm is merely psuedocode here - among other things, in the implemented algorithm, $\vec{X}$ is stored permenantly for each worker once, and does not need to be repeatedly computed or communicated between threads.

At first glance, this algorithm seems trivially correct; however, because of the kL divergence for the sparsity term, the above function does not necessarily compute the correct cost function such that ParallelAvgCostFunction(W,X) = SerialCostFunction(W,X). Regardless, we can prove that ParallelAvgCostFunction is an extremely good approximation - and the results confirm this. Also note that the gradient computed by ParallelAvgCostFunction is correct with respect to the cost function ParallelAvgCostFunction computes.

3.1 Alternatve Algorithm

From the cost function definition, we notice that the error-squared term is trivially parallelizable - each thread can compute the term on a different subset of patches and then the results are averaged - and the weights-squared term is relatively cheap to compute (so it does not need to be parallelized), but the kL divergence term (the last term) is non-trivial to make parallel.

An intuitive way to compute the kL divergence correctly in a parallel setting is with the following algorithm:

```
proc ParallelExactCostFunction($\vec{W}, \vec{X}$) $\equiv$
  foreach t parallel do
    $\vec{X}_t$ := GetThreadData($\vec{X}, t$);
    cost$_t$, grad$_t$, p$_t$, a$_t$ := SerialErrorSquared($\vec{W}, \vec{X}_t$);
  end
  avgCost := average(... cost$_t$ ...);
  avgGrad := average(... grad$_t$ ...);
  p := average(... p$_t$ ...);
  foreach t parallel do
    $\vec{X}_t$ := GetThreadData($\vec{X}, t$);
    sGrad$_t$ := SparsityTermGrad($\vec{W}, \vec{X}_t$, p, a$_t$);
  end
  wCost, wGrad = SumWeightsTerm(W);
  cost = avgCost + wCost + SparsityTermCost($\rho$, p);
  grad = avgGrad + wGrad + sum(sGrad$_t$);
  return cost, grad;
.
```

The complexity (and potential performance hit) from this approach arises from the fact that in order to calculate the gradient with respect to the kL term on a batch of examples, the thread needs the correct value of p_j in addition to all of the activations a_t calculated on the batch. In an actual implementation, a_t would not be communicated between threads. The worker thread would merely store a_t locally in the call to SerialErrorSquared for later use in the SparsityTerm function.

3.2 Parallel Correctness

Fortunately, we can prove high probability bounds on the difference between ParallelAvgCostFunction and SerialCostFunction. We'll start, however, by proving a few basic facts.

Our training set consists of iid examples, and for each training example, the hidden node j activates by a Bernoulli trial with probability p_j - the true probability of hidden node j firing on a random example from the training set. Let $p_j^{(i)}$ be the i'th thread's approximation to p_j. By definition, $p_j^{(i)}$ is the mean of the $\frac{m}{t}$ Bernoulli trials that determine the activation of n_j on thread i's chunk of the training set.

Fact 3.1. With probability at least

$$1 - 2th \exp\left(\frac{-2m}{t} \exp\left(\frac{2\epsilon}{h}\right)\right)$$

there does not exist a thread i and hidden node j such that $|p_j^{(i)} - p_j| > \exp\frac{\epsilon}{h}$.

Proof By a corollary of the Hoeffding inequality (proven using the union bound [1]), since we have $t*h$ independent estimations of p_j, we have the following:

$$P(\neg\exists j \in [h], i \in [t] : |p_j^{(i)} - p_j| > \exp\frac{\epsilon}{h})$$
$$\geq 1 - 2th \exp\left(\frac{-2m}{t} \exp\left(\frac{2\epsilon}{h}\right)\right)$$

■

Fact 3.2. Assume $|x - y| \leq z$. This implies $|\log x - \log y| \leq \log z$.

Proof Without loss of generality, assume $x > y$.

$$|x - y| \leq z \quad (1)$$
$$\Rightarrow x - y \leq z \quad (2)$$
$$\Rightarrow x \leq y + z \quad (3)$$
$$\Rightarrow \log x \leq \log(y + z) \quad (4)$$
$$\Rightarrow \log x \leq \log y + \log z \quad (5)$$
$$\Rightarrow \log x - \log y \leq \log z \quad (6)$$
$$\Rightarrow |\log x - \log y| \leq \log z \quad (7)$$

Step (1) is the assumption made in the fact statement. Step (2) followed from our assumption without loss of generality. Step (4) followed from the fact that log is an increasing function. Step (5) is justified by the fact that log is concave. Step (7) is justified by the fact that $x \geq y$ implies $\log x \geq \log y$ because log is increasing. ■

Fact 3.3. Assume $|p_j^{(i)} - p_j| \leq \frac{\epsilon}{h}$. This implies $|\log p_j^{(i)} - \log p_j| \leq \log\frac{\epsilon}{h}$ and $|\log(1 - p_j^{(i)}) - \log(1 - p_j)| \leq \log\frac{\epsilon}{h}$.

Proof Apply Fact 3.2 with $x = p_j^{(i)}, y = p_j, z = \exp\frac{\epsilon}{h}$.

Similarly, $|p_j^{(i)} - p_j| \leq \frac{\epsilon}{h} \Rightarrow |(1-p_j) - (1-p_j^{(i)})| \leq \exp\frac{\epsilon}{h}$. Apply Fact 3.2 with $x = (1-p_j), y = (1-p_j^{(i)}), z = \exp\frac{\epsilon}{h}$. ■

Theorem 3.1. Let t be the number of threads, m be the total number of training examples, h be the number of hidden nodes, and ϵ be some permissable error. Let C be the actual cost function on W and C^* be the approximation calculated by ParallelAvgCostFunction.

$$P(|C - C^*| \leq \lambda_\rho \epsilon) = 1 - 2th\exp\left(\frac{-2m}{t}\exp\left(\frac{2\epsilon}{h}\right)\right)$$

Proof Consider the cost function split into terms:

$$\begin{aligned}
C &= C_E + C_W + C_P \\
C_P &= \lambda_\rho \sum_j \rho \log\frac{\rho}{p_j} + (1-\rho)\log\frac{1-\rho}{1-p_j} \\
C^* &= C_E^* + C_W^* + C_P^* \\
C_P^* &= \frac{1}{t}\sum_{i=1}^{t} \lambda_\rho \sum_j \rho\log\frac{\rho}{p_j^{(i)}} + (1-\rho)\log\frac{1-\rho}{1-p_j^{(i)}}
\end{aligned}$$

The $C_E^* = C^E$ follows from the assumption that all threads have exactly $\frac{m}{t}$ examples in their chunk. $C_W^* = C_W$ trivially, as W is the same for all threads. This leaves us with needing to bound the value $|C_P - C_P^*|$. Assume that $|p_j^{(i)} - p_j| \leq \exp\frac{\epsilon}{h}$. By Fact 3.1, we know this occurs with probability at least $1 - 2th\exp\left(\frac{-2m}{t}\exp\left(\frac{2\epsilon}{h}\right)\right)$.

$$\begin{aligned}
&|C_P - C_P^*| \\
&\quad = |\lambda_\rho \sum_j \rho\log\frac{\rho}{p_j} + (1-\rho)\log\frac{1-\rho}{1-p_j} \\
&\quad - \frac{1}{t}\sum_{i=1}^{t}\lambda_\rho\sum_j \rho\log\frac{\rho}{p_j^{(i)}} + (1-\rho)\log\frac{1-\rho}{1-p_j^{(i)}}| \\
&\quad = |\lambda_\rho \sum_j [-\rho\log p_j - (1-\rho)\log(1-p_j) \\
&\quad + \frac{1}{t}\sum_{i=1}^{t}\rho\log p_j^{(i)} + (1-\rho)\log(1-p_j^{(i)})]| \\
&\quad = |\lambda_\rho \sum_j [\rho(\log p_j - \frac{1}{t}\sum_{i=1}^{t}\log p_j^{(i)}) \\
&\quad + (1-\rho)(\log(1-p_j) - \frac{1}{t}\sum_{i=1}^{t}\log(1-p_j^{(i)}))]| \\
&\quad = |\lambda_\rho \sum_j \sum_{i=1}^{t} [\frac{\rho}{t}(\log p_j - \log p_j^{(i)}) \\
&\quad + \frac{1-\rho}{t}(\log(1-p_j) - \log(1-p_j^{(i)}))]| \\
&\quad \leq \lambda_\rho \sum_j \sum_{i=1}^{t} [\frac{\rho}{t}|\log p_j - \log p_j^{(i)}| \\
&\quad + \frac{1-\rho}{t}|\log(1-p_j) - \log(1-p_j^{(i)})|] \\
&\quad \leq \lambda_\rho \sum_j \sum_{i=1}^{t} [\frac{\rho}{t}\log\exp\frac{\epsilon}{h} + \frac{1-\rho}{t}\log\exp\frac{\epsilon}{h}] \\
&\quad = \rho\sum_j\sum_{i=1}^{t}[\frac{\rho}{t}\frac{\epsilon}{h} + \frac{1-\rho}{t}\frac{\epsilon}{h}] \\
&\quad = \lambda_\rho\sum_j[\rho\frac{\epsilon}{h} + (1-\rho)\frac{\epsilon}{h}] \\
&\quad = \lambda_\rho h[\rho\frac{\epsilon}{h} + (1-\rho)\frac{\epsilon}{h}] = \lambda_\rho h[\frac{\epsilon}{h}] = \lambda_\rho\epsilon
\end{aligned}$$

Aside from algrebraic manipulation, we used the fact that $|\sum_x f(x)| \leq \sum_x |f(x)|$, and we used substitution using Fact 2.3. ■

This result proves that ParallelAvgCostFunction is an extremely good approximation, so long as $\frac{m}{t}$ is a reasonable value. For instance, in our most typical benchmark, we have $m = 100000$, $h = 30$, and pretend we were testing on $t = 1000$ nodes. The difference between ParallelAvgCostFunction and the true cost on those examples will be no more than $\lambda_\rho 10^{-100}$ with probability at least $1 - 1.77 \times 10^{-82}$.

3.3 Multi-core Benchmarks

Using Parallel Python, we implemented ParallelAvgCostFunction for testing in a multi-core environment - a quad-core, hyper-threading enabled desktop (Intel Core i7 920 @ 2.67GHz, 6.00 GB RAM). According to Intel, hyperthreading improves performance by approximately 30% [2]. We ran benchmarks to demonstrate parallel speedup with respect to batch size. Each execution time was averaged over 3 independent trials.

(We also benchmarked ParallelExactCostFunction. On 100K batch size, ParallelExactCostFunction was an average of 8 to 12 seconds slower than ParallelAvgCostFunction, regardless of the number of threads.)

Total Running Time (seconds)

Workers	1	2	4	8
1K	9	10	13	16
10K	64	57	64	75
100K	650	440	370	350
1M	6431	4049	3425	3360

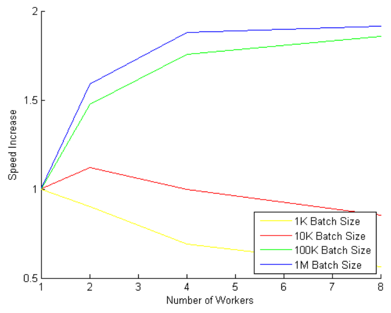

Figure 3.3.1. Average speedup across batch size

3.4 Cluster Benchmarks

The Parallel Python framework used in the multi-core benchmarks is unfortunately ill-suited for learning the sparse autoencoder in the clusters. It is not possible (without modifying the source to Parallel Python) to have worker threads maintain copies of their own example sets in memory, meaning that the threads would have to hit the disk every iteration. Fortunately, another 229 group (see Acknowledgements) developed the QJAM parallel framework for Python.

The following benchmarks were performed on the yggdrasil machines courtesy of the Stanford AI Lab:

Total Running Time (seconds)

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

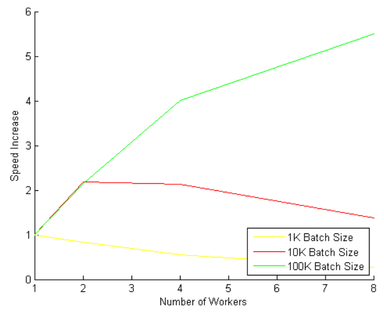

Figure 3.4.1. Average speedup across batch size

While the performance of the framework suffers on smaller batch sizes (because of the high cost of communicating within a cluster), a speed up of 5.5 on 8 cores for a batch size of 100K is quite significant. For more analysis of the cluster benchmarks, see the project paper written by the framework's creators.

4 Conclusions

In testing serial optimization algorithms for the sparse autoencoder, we determined that L-BFGS demonstrated faster convergence than conjugate gradient, and thus elected to use L-BFGS in our parallel implementation.

We also demonstrated that our approximation ParallelAvgCostFunction is an intiuitive and extremely accurate approximation to the actual value of the cost function.

The parallelism obtained on the QJAM framework with our parallel implementation of the sparse autoencoder is quite promising, especially when contrasted with the results obtained by Parallel Python in a multi-core environment. With a batch size of 100K, the parallel python framework could only achieve a 1.75x speedup on 4 workers, compared to the full 4x speed up on 4 workers achieved by the QJAM framework. The difference in serial execution time between our multi-core test machine and yggdrasils is puzzling (100K patch size: 5030 seconds on yggdrasil compared to 650 seconds on our test machine), but the slower serial execution time alone cannot account for the better parallelism achieved on QJAM; with 1M patches and a serial execution time of 6431 seconds, the Parallel Python framework still only achieved a 1.88x speed up on 4 workers.

5 Acknowledgements

We would like to thank the following: Professor Ng and Adam Coates for advising this project; Juan Batiz-Benet, Quinn Slack, Matt Sparks, and Ali Yahya for their work on the QJAM Python parallel framework; Milinda Lakkam and Sisi Sarkizova for their collaboration on the sparse autoencoder.

6 References

[1] http://www.stanford.edu/class/cs229/notes/cs229-notes4.pdf
[2] http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology/