Parallelized Sparse Autoencoder Using MPI

Nathaniel Lewis, Zachary Canann
May 8th, 2015

Parallel Computing, Dong Li

Introduction

The sparse autoencoder is a multilayer artificial neural network that attempts to learn distinctive features of input data, which in this case a compressed representation of the input. It accomplishes this by attempting to learn the function $h_k(x) = x$ while minimizing activation via a supervised learning technique, such as backpropagation combined with stochastic gradient descent. The backpropagation algorithm is given a sparsity constraint to enforce minimal activation of the neurons, resulting in the learning of a compressed representation of the input. These features are patterns and underlying input structure that maximally activate nodes of the hidden layer of the network. The objective of this paper is to present an efficient parallelized implementation of the sparse autoencoder.

Implementation

Our implementation is based off the paper written by Professor Andrew Ng [1], as well as the videos he created to supplement his paper [2]. We started with the serial implementation in Python of this paper given by Siddharth Agrawal [3], and modified it to use MPI for Python [4] to support parallelization on a cluster. Provided along with Agrawal's implementation was a set of ten 512x512 black and white images of scenes, with low amounts of detail except for many sharp edges for objects in the scene. By default, ten thousand 8x8 samples are randomly sampled from the set of images which represent the training set. The default neural network uses 64 input nodes (one for each pixel in the input images), 25 hidden nodes which represent the learned features, and 64 output nodes. The number of input nodes has to be equal to the number of output nodes, as we are attempting to learn the function $h_k(x) = x$. The hidden layer can be a variable size, but the larger it is the more features will be learned. The pseudo code for his implementation is as follows:

```
# Pseudo Code for Serial version
hidden_patch_side = 5
visible_patch_side = 8
iterations = 400
training_data = random_samples(10000, visible_patch_side)
$W_{hidden}$ = random_weights(hidden_patch_side, visible_patch_side)
$W_{output}$ = random_weights(visible_patch_side, hidden_patch_side)
$B_{hidden}$ = random_weights(hidden_patch_side, 1)
$B_{output}$ = random_weights(visible_patch_side, 1)

# Learn the input samples (A = activation, W = weights, D = divergence, Grad = gradient)
for i from 1 to iterations
    $A_{hidden}$ = sigmoid($W_{hidden}$ * training_data + $B_{hidden}$)
    $A_{output}$ = sigmoid($W_{output}$ * $A_{hidden}$ + $B_{output}$)
    error = $A_{output}$ - training_data
    square_error = sum($error^2$)
    $A_{average}$ = mean($A_{hidden}$)
    D = divergence(beta, rho, $A_{average}$)
    $D_{grad}$ = $divergence_{grad}$(beta, rho, $A_{average}$)
    $W_{decay}$ = decay(lambda, $W_{hidden}$, $W_{output}$)
    cost = square_error + $W_{decay}$ + D
    Grad = gradient(lambda, $D_{grad}$, $A_{hidden}$, $A_{output}$, $W_{hidden}$, $W_{output}$)
    [$W_{hidden}$, $W_{output,}$ $B_{hidden}$, $B_{output}$] = optimize(cost, Grad)
loop
for i from 1 to hidden_patch_side^2
    image($W_{hidden}$(i,:) as [visual_patch_side, visual_patch_size])
loop
```

Parallelization

Parallelization was inspired by Dahl, McAvinney, and Newhall's paper [5]. They argue that it is possible to parallelize neural network training by running training independently on multiple nodes while keeping a log of the changes made to the weights. After a certain number of iterations, the changes are pushed to all other nodes. This supposedly reduces the amount of time required to train the neural network. Two features of the sparse autoencoder problem present problems for their method. First, the sparse autoencoder requires a minimal activation of the neurons. The operation of combining the changes from each node (adding) greatly increases activation. We attempted to replace addition with using the average of all changes, but this drastically reduced the effectiveness of the algorithm. The second is that Dahl, McAvinney, and Newhall's method requires a method to determine whether more iterations are required. As we cannot provide this, using their method is more difficult.

The second method mentioned (for comparison) in their paper was to split to neurons of the network across multiple nodes. However, this is only practical for very large neural networks, so that the communication overhead between the neurons doesn't exceed the actual computation cost. Since the neural network we are dealing with is small (in our case, less than 250 nodes), we settled on a combination of the two methods. The original implementation of the paper computes the total contribution of all of the training samples with the current network weights per iteration. We used this to our advantage. We duplicate the neural network across all of the nodes in the network, but we divide the training samples evenly amongst the nodes. For instance, if using 10,000 training samples, two processors would each have 5000, four processors would each have 2500, etc. At the start of each iteration, each node receives the new neuron weights from the master node (MPI broadcast). Each node then runs the training samples through the network and computes the local contribution to the squared error, average activation, and the divergence gradients. We then use MPI reduce to sum these element-wise onto the master node. The master then runs the original optimization function locally.

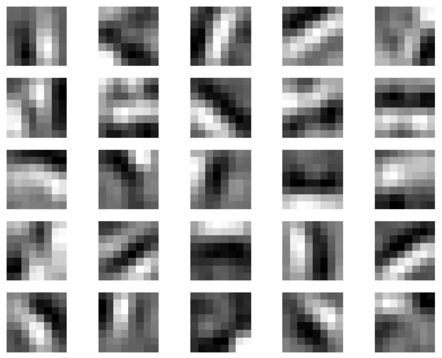
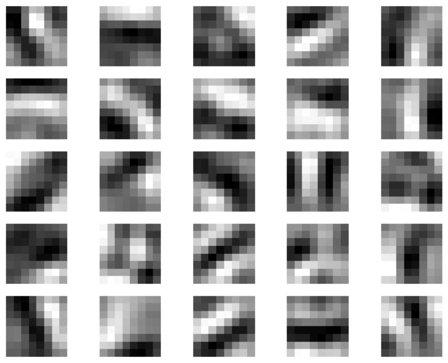
```
# Pseudo Code for Parallel Version (changes are in bold) (unchanged initialization omitted)
training_data = random_samples(**10000 / nodes**, visible_patch_side)

# Learn the input samples (A = activation, W = weights, D = divergence, Grad = gradient)
for i from 1 to iterations
    **[$W_{hidden}$, $W_{output,}$ $B_{hidden}$, $B_{output}$] = broadcast([$W_{hidden}$, $W_{output,}$ $B_{hidden}$, $B_{output}$], root)**
    $A_{hidden}$ = sigmoid($W_{hidden}$ * training_data + $B_{hidden}$)
    $A_{output}$ = sigmoid($W_{output}$ * $A_{hidden}$ + $B_{output}$)
    error = $A_{output}$ - training_data
    square_error = sum($error^2$)
    $A_{average}$ = mean($A_{hidden}$)
    $\textbf{TotalA}_{\textbf{average}}$ = reduce($A_{average}$, root)
    total_square_error = reduce(square_error, root)
    if master
        $W_{decay}$ = decay(lambda, $W_{hidden}$, $W_{output}$)
        D = divergence(beta, rho, $\textbf{TotalA}_{\textbf{average}}$)
        cost = total_square_error + $W_{decay}$ + D
        $D_{grad}$ = $\text{divergence}_{grad}$(beta, rho, $\text{TotalA}_{average}$)
    endif
    $D_{grad}$ = broadcast($D_{grad}$, root)
    Grad = gradient(lambda, $D_{grad}$, $A_{hidden}$, $A_{output}$, $W_{hidden}$, $W_{output}$)
    TotalGrad = reduce(Grad, root)
    if master
        [$W_{hidden}$, $W_{output}$, $B_{hidden}$, $B_{output}$] = optimize(cost, TotalGrad)
    endif
loop
```

Our decision to run the optimization purely on the master was because of the small size of the local problem. The gradient is a visible_patch_side^2 x hidden_patch_side^2 matrix. MPI reduce automatically optimizes by collapsing through groups of nodes. Also, because of the small matrix size, the optimization complexity is small. The computation of the local contributes is orders of magnitude greater than it. However, if the patch sizes (both hidden and visible) are increased, the communication cost is increased and so is the optimization cost. In these cases, it may become necessary to parallelize the optimization as well.

Test Hardware

The experiments were performed on as many as 16 t2.micro instances for the Amazon Elastic Compute Cloud (EC2). The t2.micro instances run on Intel Xeon Processors operating at 2.5GHz with Turbo up to 3.3GHz and 1GiB of ram [6]. The instances were ran on an Amazon Machine Image (AMI) which supports HVM, allowing for enhanced networking [6]. This is ideal for a cluster as it is crucial to minimize the communication time between nodes. These clusters are not meant to be used for sustained CPU usage [6], however these instances were chosen since they are the most powerful free processing tier with an Amazon student account. Despite these limitations, we believe that our experiments were long enough in duration such that the noise was negligible from the bursty CPU performance. The cluster was managed using StarCluster, a cluster-computing toolkit for Amazon EC2 managed by MIT [7].

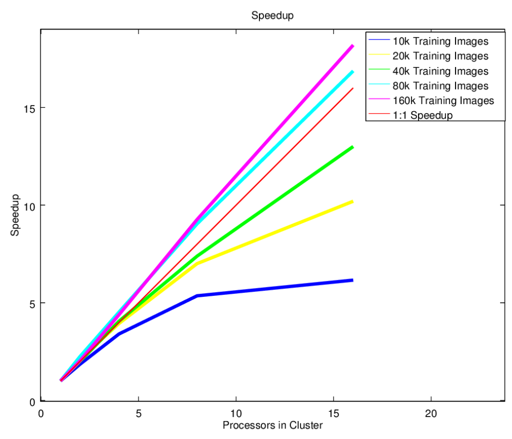
Results

Figures 1.1 - Images generated by the sequential algorithm, and Figure 1.2 - parallel algorithm for a 16 node cluster. Each used 10,000 training images.

The sparse autoencoder attempts to learn a compressed representation of the input by learning the underlying structure in the training data. We were able to show that with our training data, the sparse autoencoder became maximally responsive to edge-like structures. as seen in Figure 1.1. Our implementation attempts to retain the same quality as the serial implementation by distributing the operations performed by the serial version across multiple nodes, versus a method such as Dahl, McAvinney, and Newhall's which sacrifices accuracy per iteration for lower communication cost. The results from our algorithm running on a 16 node cluster performing the same amount of iterations as the serial implementation can be seen in Figure 1.2.

Performance



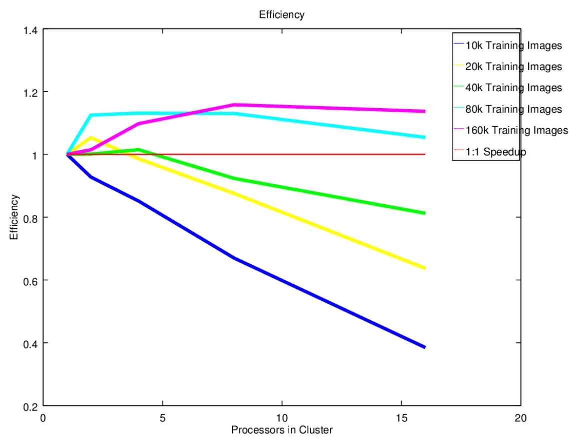


Figure 2.1 - Speedup and Figure 2.2 - Efficiency of the Parallelized Sparse Autoencoder on 1 to 16 processors and 10,000 to 160,000 images in the training set

Our results show that our parallel implementation scales well with a large amount of training data. As seen in Figure 2.1 and 2.2, as the training data size is increased, the speedup becomes more linear. This is due to local computation cost versus the communication cost. Given the encoder configuration in our experiment (8x8 patch size, 25 learned features), the communication consists of broadcasting two 64x25 matrices, a 1x25 matrix, and a 1x64 matrix. It also consists of performing MPI reduce on the same matrices. In the 10k sample image test, the local computation on sixteen nodes would be 625 samples, meaning the communication cost is similar the local computation cost.

The superlinear performance observed for the 80k and 160k sample trials is most likely an artifact of our test environment (Amazon EC2). We don't know how localized the virtual machines we were allocated are, and we don't know the exact details of how the machines are scheduled. One thing is certain - each virtual machine is not assigned its own physical CPU, but rather it has a certain execution priority. We don't know if the constant network communication raises the priority of our instances. This could also be a result of exhausting the cache on a singular node. Running a single node with 160k training samples results in about 50 MB of memory consumption, which is more data than fits in the cache of the highest tier Intel Xeon Processor. Also, the mere fact that we are running in a virtualized environment means we probably lose everything we have in cache when the context is switched to another VM.

Processes / Training Size	10000	20000	40000	80000	160000
1	46.44s	103.81s	205.89s	468.89s	965.79s
2	25.03s	49.28s	102.85s	208.40s	475.97s
4	13.64s	26.32s	50.72s	103.60s	219.94s
8	8.67s	14.82s	27.87s	51.86s	104.26s
16	7.53s	10.18s	15.84s	27.80s	53.09s

Figure 3.1 - Raw Experiment Runtimes

Looking at Figure 3.1 reveals some interesting trends in regards to the communication cost inside the cluster. If we look at the trials where the local workload is the same (10k on 1 node, 40k on 4 nodes, etc.), we notice the execution time increasing a small amount (46.5s for 10k on 1 node, 53s for 160k on 16 nodes). It appears to be growing along the cost = $\log_2(p)$, which supports the notion that MPI reduce does an optimized collapse using all nodes involved, versus just sending everything to be summed on the master. This also means that as long as there is enough training data to load every processor sufficiently, the communication cost will not grow to a prohibitive amount.

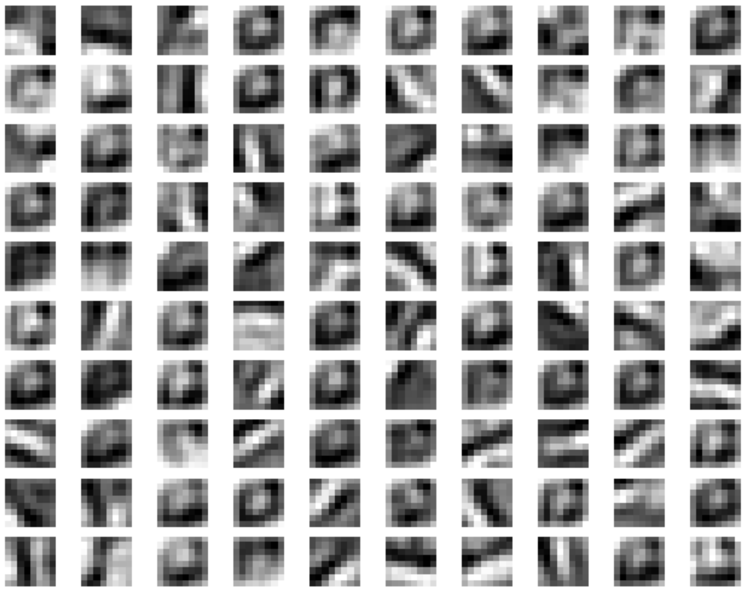


Figure 4.1 - Increasing the hidden size to 100 features

It is possible to tweak the hidden_patch_side parameter to learn more features (Figure 4.1). Increasing the patch sizes increases communication cost between the nodes, but it also generates significantly more to be done on the local node. Increasing the learned features may even increase the efficiency.

Conclusion

Our parallelization of the sparse autoencoder is an efficient algorithm that scales very well with the training data and number of processes, while maintaining the accuracy of the learned features. If there is a sufficient amount of training data there is a clear benefit in using our parallel implementation, as using 16 machines can result in up to an 18x increase in speed in training the neural network. In our implementation there is no mechanism for coping with extremely large training sets with RAM limitations, so we will leave this as future work for handling this problem, while at the same time maintaining efficient speedup. Also, future work could include determining how well our algorithm scales with extremely large training sets and with more machines, and to see if the trends observed in our results continue. In our work our training sample images were no larger than 8x8, so we only run the optimization function on the master node. However, if dealing with larger images, there may be even more future work in parallelizing the optimization.

References

[1] Andrew Ng, CS294A Lecture notes, Stanford University
http://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf
[2] Sparse Autoencoder Lecture Videos Parts 1 and 2
http://web.stanford.edu/class/cs294a/video1.html
http://web.stanford.edu/class/cs294a/video2.html
[3] Sparse Autoencoder Serial Python implementation
https://github.com/siddharth-agrawal/Sparse-Autoencoder
[4] Documentation for mpi4py
http://mpi4py.scipy.org/docs/usrman/tutorial.html
[5] Parallelizing Neural Network Training for Cluster Systems
http://www.cs.swarthmore.edu/~newhall/papers/pdcn08.pdf
[6] Amazon EC2 t2.micro Instances
http://aws.amazon.com/ec2/instance-types/
[7] StarCluster
http://star.mit.edu/cluster/