Asynchronous Distributed Neural Network Training using Alternating Direction Method of Multipliers

Christopher B. Choy  
Stanford University  
Department of Electrical Engineering  
chrischoy@stanford.edu

Wonhui Kim  
Stanford University  
Department of Electrical Engineering  
wonhui@stanford.edu

Abstract

Since the first appearance of a large-scale dataset [4] and powerful computational resources such as GPUs, Convolutional Neural Networks (CNN) became the essential machine learning algorithm for image classification, detection, and many application. As the popularity of CNN increases, the size of CNN increased as well [10, 13]. For instance, AlexNet has more than 60 million parameters. The performance of such large networks improves steadily yet training these networks takes longer.

In this paper, we propose a novel asynchronous distributed neural network optimization using Alternating Direction Method of Multipliers (ADMM) [6]. Unlike previous works on distributed optimization for neural network training [1, 5, 3, 12] which rely solely on the primal optimization, we formulate the problem into a global consensus optimization and distribute the neural network training in a principled fashion. We evaluate our framework on CIFAR10 dataset and analyze the effect of hyperparameters introduced in ADMM.

1. Introduction

As computing power increases and the large-scale image data becomes available [4], Convolutional Neural Networks (CNNs) architecture has recently become one of the most powerful learning tool in computer vision, achieving state-of-the-art results in various computer vision tasks. It is true that CNNs handle much smaller number of parameters than the standard neural networks by sharing weights (convolution layers), but several state-of-the-art CNNs such as AlexNet [10] and VGGNet [13] still require a lot of parameters. The large number of parameters makes training hard, causing fatal issues such as overfitting and local optima.

In order to overcome these challenges, it is common to feed a large amount of data into the network, and also several variants of CNN training algorithm have been proposed such as Dropout [14]. However, several passes of forward and backward propagation over the entire network for every training sample spend excessive amount of training time, ranging from days to a week [10, 13]. Additionally, more sophisticated techniques to reduce overfitting such as Dropout add extra computation, thereby slowing down the training even more [10, 14].

There have been several attempts to reduce the training time by using the computing power of multi-GPUs with parallelizing either data or parameters [2, 12, 3]. The data parallelization scheme simply distributes training data minibatches to multiple processors and propagates gradients over the replicated networks with the same structure. The model parallelization scheme breaks down the network into smaller pieces and distributes the pieces of parameters to multiple processors. This scheme is especially useful to train a network that is too large to cache on a single processor [1, 5, 3]. The hybrid of two parallelization schemes distributes both the data and parameters [3]. To successfully merge the trained parameters over multiple processors for both schemes, the processors are usually enabled to share the intermediate training results via one server processor, which gathers updates from processors and synchronously broadcasts the merged result. However, synchronous updating yields overhead in time due to the waiting time until all the processors transmit the intermediate results to the server. While HogWild! [12] algorithm suggests a simple lock-free approach to asynchronously update the parameters, it assumes the sparsity on the data, and suffers from data collision, otherwise.

In this paper, we present a novel asynchronous distributed neural network training framework based on Alternating Direction Method of Multipliers (ADMM) [6] technique and the data parallelization scheme. In Section 2, we discuss related work and briefly introduce equations for ADMM, which has been widely used in variety of optimization problems [16, 15, 11]. In Section 3, we present our ADMM-based parameter update equations adapted to
neural network training context. In Section 4, we suggest two asynchronous models for the efficient implementation of our framework. The rest of the sections present experimental setup, results, analysis, and future plan.

2. Related Work

Our distributed neural network training framework is motivated by the work in [10], which solves the distributed linear classification problem by applying ADMM. The underlying idea in our proposed framework is based on data parallelism [9] and distributed optimization theories [11], which we briefly discuss below.

Data Parallelism. The underlying idea is based on splitting the entire training data into \( m \) disjoint subsets and providing each subset to one processor. Each processor shares the same network structure, but trains the network independently with different data samples. The key challenge is how to merge the results from independent networks. For the naive data parallelization scheme, the server periodically synchronizes the results from all the processors and then broadcasts to them.

Our framework is also based on similar idea, but there is a difference. The data parallelization implementation in [9] splits one minibatch into \( m \) disjoint microbatches, and pushes each microbatch to the queue. Therefore, each of \( m \) processors is fed with different data samples at every pass. However, our framework requires to fix the subset of data samples for each processor. By mapping deterministically mapping from each microbatch to a processor, we can adapt the data parallelization scheme to our framework.

Dual Ascent. Consider the equality-constrained convex optimization problem where \( f \) is the objective function and \( w \) is a primal variable.

\[
\begin{align*}
\text{minimize} & \quad f(w) \\
\text{subject to} & \quad Aw = b
\end{align*}
\]  

The dual problem of Eq. 1 is to maximize \( g(\lambda) \) where \( \lambda \) is a dual variable. We can solve this dual problem instead of primal problem by using gradient ascent algorithm. Since \( \nabla \lambda g(\lambda) = Aw^* - b \) where \( w^* = \arg \min_w \mathcal{L}(w, \lambda) \), the update equations can be written as follows:

\[
\begin{align*}
w^{k+1} & \leftarrow \arg \min_w \mathcal{L}(w, \lambda^k) \quad \text{(w-minimization step)} \\
\lambda^{k+1} & \leftarrow \lambda^k + \eta(Aw^{k+1} - b) \quad \text{(dual update)}
\end{align*}
\]  

With an appropriate step size \( \eta \), alternately repeating the above update equations for \( w \) and \( \lambda \) until convergence would result in the optimal solution.

Dual Decomposition. If \( f \) is separable as \( f(w) = \sum_{i=1}^m f_i(w) \), then the lagrangian \( \mathcal{L}(w, \lambda) \) is also separable in \( w \) as \( \mathcal{L}(w, \lambda) = \sum_{i=1}^m \mathcal{L}_i(w_i, \lambda) \).

Method of Multipliers. By adding an additional penalty term to the Lagrangian, we can get the augmented Lagrangian:

\[
\mathcal{L}_\rho(w, \lambda) = f(w) + \lambda^T(Aw - b) + \frac{\rho}{2} \|Aw - b\|^2
\]  

Method of multipliers solves the optimization problem defined in Eq. 4 by applying dual ascent algorithm with this augmented lagrangian. The role of a penalty term is to make dual function \( g \) shown to be differentiable.

Alternating Direction Method of Multipliers. ADMM is a combination of dual decomposition with the method of multipliers. If \( f(w) = f_1(w_1) + f_2(w_2) \) and the equality constraint is \( A_1w_1 + A_2w_2 = b \), then we can apply the dual ascent algorithm as follows:

\[
\begin{align*}
w_1^{k+1} & \leftarrow \arg \min_{w_1} \mathcal{L}_\rho(w_1, w_2^{k}, \lambda^k) \\
w_2^{k+1} & \leftarrow \arg \min_{w_2} \mathcal{L}_\rho(w_1^{k+1}, w_2, \lambda^k) \\
\lambda^{k+1} & \leftarrow \lambda^k + \rho(A_1w_1^{k+1} + A_2w_2^{k+1} - b)
\end{align*}
\]  

If \( f \) consists of \( m \) separate terms, then we can apply the dual ascent algorithm with \( m \) separate \( w \)-minimization steps followed by one dual update for \( \lambda \). Each iteration of ADMM first minimizes \( \mathcal{L}_\rho \) jointly with respect to \( m \) primal variables, and then updates the dual variable \( \lambda \).

Global Consensus Optimization. Consider the situation where we decompose the objective \( f \) into \( m \) terms as \( f(w) = \sum_{i=1}^m f_i(w) \), and solve the optimization problem with \( m \) processors. After \( f_i \)'s are distributed to each processor, \( m \) subproblems are being solved independently. In order to determine the global solution from this \( m \) sub-solutions, we can introduce the global consensus variable \( z \), and pose identity constraints such that \( z = w_i \) to all the subproblems where \( w_i \) is the local variable corresponding to the \( i \)-th processor. The global consensus optimization problem is defined as follows:

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=1}^m f_i(w_i) \\
\text{subject to} & \quad w_i - z = 0, \quad i = 1, \ldots, m
\end{align*}
\]  

The resulting ADMM algorithm with the consensus variable \( z \) is the following:

\[
\begin{align*}
w_i^{k+1} & \leftarrow \arg \min_{w_i} \mathcal{L}_\rho(w_1^k, \cdots, w_i, \cdots, w_m^k, z^k, \lambda^k) \\
z^{k+1} & \leftarrow \frac{1}{m} \sum_{i=1}^m (w_i^{k+1} + \lambda_i^k / \rho)
\end{align*}
\]  

\[
\lambda_i^{k+1} \leftarrow \lambda_i^k + \rho(w_i^{k+1} - z^{k+1})
\]
3. A Distributed CNN Training Framework Using ADMM

We can apply ADMM to CNN training because the objective function is separable. Let \( W \) denote a two-dimensional parameter matrix, and \( X = \{ x_i \}_{i=1}^{N}, Y = \{ y_i \}_{i=1}^{N} \) be the set of train data and the set of corresponding class labels. The final \( L_2 \)-regularized loss can be expressed in terms of a linear sum of individual losses over the entire train data. Suppose we split the dataset \( D \) into \( m \) subsets \( \{ D_j \}_{j=1}^{m} \). The objective function can be partitioned into \( m \) disjoint terms as follows:

\[
J(W; D) = \frac{1}{N} \sum_{i=1}^{N} l(W; x_i, y_i) + \frac{\lambda}{2} \| W \|^2
\]

\[
= \frac{1}{N} \sum_{j=1}^{m} \sum_{i \in D_j} l(W; x_i, y_i) + \frac{\lambda}{2} \| W \|^2
\]

\[
= \sum_{j=1}^{m} \left[ J_j(W; D_j) + \frac{\lambda}{2m} \| W \|^2 \right]
\]

To apply the data parallelization scheme, each of \( m \) terms should be fed into one of the processors. Although the objective is divided into \( m \) separate terms and would be processed independently, all the processors are actually solving the same problem. Therefore, we can combine ADMM with the global consensus optimization problem by introducing the consensus variable \( Z \).

\[
\text{minimize}_{W_j} \sum_{j=1}^{m} \left[ J_j(W_j; D_j) + \frac{\lambda}{2m} \| W_j \|^2 \right]
\]

subject to \( W_j = Z, \quad j = 1, ..., m \) \hspace{1cm} (17)

The augmented Lagrangian of the objective function is

\[
\mathcal{L}_\rho(W, Z; A) = \sum_{j=1}^{m} \left[ J_j(W_j; D_j) + \frac{\lambda}{2m} \| W_j \|^2 \right]
+ \text{tr} \left( A_j^T (W_j - Z) \right) + \frac{\rho}{2} \| W_j - Z \|^2
\]

where \( \Lambda_j \) is the \( j \)-th Lagrange dual multiplier matrix, \( \rho \) is the augmented Lagrange dual variable, and \( \text{tr}(A_j^T (W_j - Z)) \) is the sum of all the matrix elements after multiplying \( \Lambda_j \) with \( W_j - Z \).

Finally, the unscaled version of ADMM iteration for the \( j \)-th data subset is

\[
W_j^{k+1} = \arg \min_W \mathcal{L}_{\rho,j}(W, Z^k, \Lambda_j^k)
\]

\[
Z^{k+1} = \arg \min_Z \mathcal{L}_{\rho,j}(W_j^{k+1}, Z, \Lambda_j^k)
\]

\[
\Lambda_j^{k+1} = \Lambda_j^k + \rho(W_j^{k+1} - Z^{k+1})
\]

\hspace{1cm} (19) \hspace{1cm} (20) \hspace{1cm} (21)

By replacing \( \Lambda_j = \rho U_j \), we can simplify the update equations. This is called scaled version of ADMM.

\[
W_j^{k+1} = \arg \min_W \mathcal{L}_{\rho,j}(W, Z^k, \Lambda_j^k)
\]

\[
Z^{k+1} = \arg \min_Z \mathcal{L}_{\rho,j}(W_j^{k+1}, Z, \Lambda_j^k)
\]

\[
\Lambda_j^{k+1} = \Lambda_j^k + \rho(W_j^{k+1} - Z^{k+1})
\]

\hspace{1cm} (19) \hspace{1cm} (20) \hspace{1cm} (21)

4. Asynchronous Models

Our proposed framework can be implemented in two different ways depending on the communication strategy among processors. One is the client-server model, where
Figure 2. Peer-to-peer model implementation where the number of processors is \( m = 4 \). The processor 0 wants to update its local \( Z \) so the figure shows the process of gathering \( W + U \) values from the rest \( m - 1 \) processors. The \( Z \) computing process is repeated periodically, and is proceeded asynchronously in different processors.

one server processor asynchronously receives requests from each client, updates the consensus variable \( Z \), and broadcasts the updated \( Z \) to every client. More specifically, suppose we have \( m \) number of client processors. The train data is partitioned into \( m \) disjoint batches, and each batch is fed into the corresponding GPU processor. Every GPU processor shares the same network structure, but trains their own network independently. To transfer the intermediary trained parameters from \( m \) processors, each processor asynchronously reports its \( W \) and \( U \) to the server and gets back the newly updated \( Z \) from the server at every given period. Fig. 1 shows how the server and the client 0 communicate to transfer parameters.

The other is the peer-to-peer model, where client processors intercommunicate directly to transfer parameters without the server processor. Instead of reporting \( W \) and \( U \) to the server and receiving the updated \( Z \) from the server as in the client-server model, \( Z \) is defined in all the processors, and is updated locally. When the client reaches certain period, it gathers up the \( U + W \) values from every client, and computes \( Z \) by using the Eq. 23. Fig. 2 shows how the processor 0 gathers up parameters from other processors.

Although the peer-to-peer model requires larger number of communications than the client-server model to update \( Z \) once because the processor has to fetch \( U + W \) from the rest \( m - 1 \) processors, fast GPU-GPU communication may overcome this communication overhead.

5. Experiments

The experiments consist of the following.

- Compare Client-Server communication model with Peer-to-Peer model
- Compare several variants of ADMM update equations

The last two experiments are still ongoing.

5.1. Implementation

Since forward and backward propagation involve a plethora of element-wise matrix operations which are parallelizable, GPU programming provides significant speedup. cuda-convnet2 package \(^8\) supports fast CNN training in C++/CUDA, and provides different types of parallelization schemes described in \(^9\). Our implementation of asynchronous distributed neural network training based on ADMM and data parallelism is extended from this package.

5.2. Experimental Setup

We used two GPUs (GeForce GTX 680), and experimented with a fixed network structure described in Fig. 3 on the CIFAR-10 \(^7\) dataset. The dataset consists of 60,000 32x32 RGB images in 10 classes, with 6,000 images per class. 50,000 images are used for training and 10,000 images are used for validation.

5.3. Evaluation

The ultimate goal of this work is to reduce the training time by using the proposed distributed framework. Therefore, we use as an evaluation metric the total training time to achieve certain level of training accuracy.

5.4. Results

Cross-validation with different hyperparameters. To find the optimal parameters, we trained the CNN with different combinations of hyperparameters. There are two major hyperparameters to consider for our framework: penalty constant \((\rho)\), and Number of mini-batch iterations for \( W \) optimization \((N_W)\).

- Penalty constant

For a large penalty constant \( \rho \), it pulls the gradient (Eq. 25) more toward the consensus \( Z \). Thus (Fig. 4), large \( \rho \) \((= 10^{-3})\) impedes convergence in the beginning but it recovers as \( Z \) placed itself in the vicinity of the global optimal.

- Number of iterations for \( W \) optimization: \( N_W \)

In the early stage of training, each solver uses parameters \( Z \) and \( U \), which have not converged to the solution, to solve for \( W_i \) in Eq. 22. Since solving the \( W_i \) for incorrect \( Z \) and \( U \) is not meaningful, we terminate the optimization Eq. 22 after a specified number of iterations \(^{16}\). We denote the number of iteration as

4
Figure 3. An illustration of the architecture of the CNN used for our experiments. Dimensions shown in the figure are for CIFAR-10 dataset. The network consists of conv-relu, max-pool, fully-connected, locally-connected, and normalization layers. The softmax loss is used for computing the final score. \((\text{conv}5-N)\) denotes a convolutional layer with \(N\) neurons, each having \(5 \times 5 \times D\) filters, where \(D\) is the depth of the activation volume at the previous layer, \(\text{morm}-N\) denotes a local response normalization layer with the size \(N\) of the region to use for normalization, and \(\text{local}\) denotes a locally-connected layer with unshared weights.

Figure 4. Training and validation accuracies for various penalty values \(\rho\). Having the penalty value \(\rho = \lambda = 10^{-3}\) yields the best performance on the validation. \(N_W = 10^3\) (every \(\sim 22\) second) for all experiments.

As the number of iterations \(N_W\) gets larger, each thread solves the subproblem Eq. 22 more accurately thus the dual variable \(U^+ = U + W - Z\) tracks the problem accurately. On the other hand, if we set \(N_W\) to be small, it constantly update \(Z\) and \(U\) that accumulates incorrect error terms and results in large error during training. As the \(Z\) and \(U\) converge to global optima, the effect of frequent update subsides since Eq. 22 produces close to optimal solution in few steps \(N_W\).

Comparison with Naive data parallelism. To validate our framework, we trained the CNN defined as in Fig. 3 by using naive data parallelism over multiple GPUs on CIFAR-10 dataset. The results can be seen in Fig. 6 which shows data
parallelism over multiple GPUs does accelerate the CNN training procedure.

6. Discussion

The naive data parallelism implemented in cuda-convnet2 is highly optimized so the improvement from our ADMM was hard to measure. Moreover, due to the limitation in our model, we could not perform experiments with 4 GPUs yet. We expect that more speedup would come if we use more GPUs or threads since the number of communications required for naive data parallelism linearly increases. Currently, we are re-implementing the naive data parallelism, other various distributed optimization techniques, and ADMM on Caffe without excessive code optimization to measure accurate performance improvement.

We expect that the amount of time reduced would increase as the network becomes more complicated and the scale of the dataset gets larger. Since CIFAR-10 dataset consists of small images, our experiments were limited to small network. With larger network and dataset, our framework would reduce the communication overhead which occurs for the naive data parallelization. We plan to conduct further experiments on larger networks and different datasets.

7. Conclusion

Instead of naively distributing the computation over multiple processors, we attempted to apply one of the distributed optimization techniques to train CNNs. Our framework is based on the combination of ADMM and global consensus optimization, with suggesting two implementation models. While we haven’t yet finished the experiments with various settings, we observed a speedup of our framework compared to the data parallelism implemented in cuda-convnet2 package, which opens up more possibility for our approach. To our knowledge, this is the first time to apply ADMM to neural network training, so it seems meaningful to further explore the problem. We plan to continue working on this project to make the final conclusion of our approach.

One of the major challenges of this work was highly optimized implementation of cuda-convnet2 package. As we extend the package, our implementation seemed to lose the implementation-level optimality. Since our framework involves GPU computing which should be carefully implemented to optimize the program, we also plan to spend much time on implementation-level optimization.

For the future steps, we will compare the performance of Client-Server model and Peer-to-Peer model, to conclude which method minimizes the communication overhead. We also plan to experiment on different datasets with larger networks where the communication overhead of the naive data parallelism becomes worse.

There are still many other options for our framework. Experimenting with many different types of gradient-based parameter solver would be the one way. The current implementation use momentum gradient descent with momentum=0.9, and we hope to get the numerical results with RMSprop, AdaGrad, L-BFGS, etc..

We can also consider slight variations in the way to combine the global consensus optimization with ADMM, which is described in Section 3. In the previous formulation in Section 3, we replaced $\sum_j \frac{\lambda}{m} \|W_j\|^2_F$ with $\sum_j \frac{\lambda}{m} \|\Lambda_j\|^2_F$. Instead, by replacing it with $\frac{1}{2n} \|Z\|^2_F$, we can define the problem in a different way as follows:

$$ W^{k+1}_j = \arg\min_W J_j(W; D_j) + \frac{\rho}{2} \|W - Z^k + U^k_j\|^2_F $$ (28)

$$ Z^{k+1} = \sum_{j=1}^m (W^{k+1}_j + U^k_j) / m + \lambda / \rho $$ (29)

The update rule for $W_j$ becomes:

$$ \frac{\partial}{\partial W_{j,pq}} \mathcal{L}(W, Z^k, \Lambda^k) = s_j^{(i)} \delta_j^{(i+1)} + \rho(W_{j,pq} + U_{j,pq}^{(i)} - Z_{\Lambda_j}^{(i)}) $$ (30)

We plan to observe what happens if we define the problem in this way.
References