Training Neural Networks Without Gradients:
A Scalable ADMM Approach

Gavin Taylor[1]   TAYLOR@USNA.EDU
Ryan Burmeister[1]
Zheng Xu[2]   XUZH@CS.UMD.EDU
Bharat Singh[2]   BHARAT@CS.UMD.EDU
Ankit Patel[3]   ABP4@RICE.EDU
Tom Goldstein[2]   TOMG@CS.UMD.EDU

[1]United States Naval Academy, Annapolis, MD USA
[2]University of Maryland, College Park, MD USA
[3]Rice University, Houston, TX USA

Abstract

With the growing importance of large network models and enormous training datasets, GPUs have become increasingly necessary to train neural networks. This is largely because conventional optimization algorithms rely on stochastic gradient methods that don't scale well to large numbers of cores in a cluster setting. Furthermore, the convergence of all gradient methods, including batch methods, suffers from common problems like saturation effects, poor conditioning, and saddle points. This paper explores an unconventional training method that uses alternating direction methods and Bregman iteration to train networks without gradient descent steps. The proposed method reduces the network training problem to a sequence of minimization substeps that can each be solved *globally* in closed form. The proposed method is advantageous because it avoids many of the caveats that make gradient methods slow on highly non-convex problems. The method exhibits strong scaling in the distributed setting, yielding linear speedups even when split over thousands of cores.

1. Introduction

As hardware and algorithms advance, neural network performance is constantly improving for many machine learning tasks. This is particularly true in applications where extremely large datasets are available to train models with many parameters. Because big datasets provide results that (often dramatically) outperform the prior state-of-the-art in many machine learning tasks, researchers are willing to purchase specialized hardware such as GPUs, and commit large amounts of time to training models and tuning hyperparameters.

Gradient-based training methods have several properties that contribute to this need for specialized hardware. First, while large amounts of data can be shared amongst many cores, existing optimization methods suffer when parallelized. Second, training neural nets requires optimizing highly non-convex objectives that exhibit saddle points, poor conditioning, and vanishing gradients, all of which slow down gradient-based methods such as stochastic gradient descent, conjugate gradients, and BFGS. Several mitigating approaches to avoiding this issue have been introduced, including rectified linear units (ReLU) (Nair & Hinton, 2010), Long Short-Term Memory networks (Hochreiter & Schmidhuber, 1997), RPROP (Riedmiller & Braun, 1993), and others, but the fundamental problem remains.

In this paper, we introduce a new method for training the parameters of neural nets using the Alternating Direction Method of Multipliers (ADMM) and Bregman iteration. This approach addresses several problems facing classical gradient methods; the proposed method exhibits linear scaling when data is parallelized across cores, and is robust to gradient saturation and poor conditioning. The method decomposes network training into a sequence of sub-steps that are each solved to global optimality. The scalability of the proposed method, combined with the ability to avoid local minima by globally solving each substep, can lead to dramatic speedups.

We begin in Section 2 by describing the mathematical notation and context, and providing a discussion of several

weaknesses of gradient-based methods that we hope to address. Sections 3 and 4 introduce and describe the optimization approach, and Sections 5 and 6 describe in detail the distributed implementation. Section 7 provides an experimental comparison of the new approach with standard implementations of several gradient-based methods on two problems of differing size and difficulty. Finally, Section 8 contains a closing discussion of the paper's contributions and the future work needed.

2. Background and notation

Though there are many variations, a typical neural network consists of L layers, each of which is defined by a linear operator W_l, and a non-linear neural activation function h_l. Given a (column) vector of input activations, a_{l-1}, a single layer computes and outputs the non-linear function $a_l = h_l(W_l a_{l-1})$. A network is formed by layering these units together in a nested fashion to compute a composite function; in the 3-layer case, for example, this would be

$$f(a_0; W) = W_3(h_2(W_2 h_1(W_1 a_0))) \quad (1)$$

where $W = \{W_l\}$ denotes the ensemble of weight matrices, and a_0 contains input activations for every training sample (one sample per column). The function h_3 is absent as it is common for the last layer to not have an activation function.

Training the network is the task of tuning the weight matrices W to match the output activations a_L to the targets y, given the inputs a_0. Using a loss function ℓ, the training problem can be posed as

$$\min_W \quad \ell(f(a_0; W), y) \quad (2)$$

Note that in our notation, we have included all input activations for all training data into the matrix/tensor a_0. This notation benefits our discussion of the proposed algorithm, which operates on all training data simultaneously as a batch.

Also, in our formulation the tensor W contains *linear operators*, but not necessarily dense matrices. These linear operators can be convolutions with an ensemble of filters, in which case (1) represents a convolutional net.

Finally, the formulation used here assumes a feed-forward architecture. However, our proposed methods can handle more complex network topologies (such as recurrent networks) with little modification.

2.1. What's wrong with backprop?

Most networks are trained using stochastic gradient descent (SGD, i.e. backpropagation) in which the gradient of the network loss function is approximated using a small number of training samples, and then a descent step is taken using this approximate gradient. Stochastic gradient methods work extremely well in the serial setting, but lack scalability. Recent attempts to scale SGD include Downpour, which runs SGD simultaneously on multiple cores. This model averages parameters across cores using multiple communication nodes that store copies of the model. A conceptually similar approach is elastic averaging (Zhang et al., 2015), in which different processors simultaneously run SGD using a quadratic penalty term that prevents different processes from drifting too far from the central average. These methods have found success with modest numbers of processors, but fail to maintain strong scaling for large numbers of cores. For example, for several experiments reported in (Dean et al., 2012), the Downpour distributed SGD method runs slower with 1500 cores than with 500 cores.

The scalability of SGD is limited because it relies on a *large* number of inexpensive minimization steps that each use a *small* amount of data. Forming a noisy gradient from a small mini-batch requires very little computation. The low cost of this step is an asset in the serial setting where it enables the algorithm to move quickly, but disadvantageous in the parallel setting where each step is too inexpensive to be split over multiple processors. For this reason, SGD is ideally suited for computation on GPUs, where multiple cores can simultaneously work on a small batch of data using a shared memory space with virtually no communication overhead.

When parallelizing over CPUs, it is preferable to have methods that use a *small* number of *expensive* minimization steps, preferably involving a large number of data. The work required on each minimization step can then be split across many worker nodes, and the latency of communication is amortized over a large amount of computation. This approach has been suggested by numerous authors who propose batch computation methods (Ngiam et al., 2011), which compute exact gradients on each iteration using the entire dataset, including conjugate gradients (Towsey et al., 1995; Møller, 1993), BFGS, and Hessian-free (Martens & Sutskever, 2011; Sainath et al., 2013) methods.

Unfortunately, all gradient-based approaches, whether batched or stochastic, also suffer from several other critical drawbacks. First, gradient-based methods suffer from the *vanishing gradients*. During backpropagation, the derivative of shallow layers in a network are formed using products of weight matrices and derivatives of non-linearities from downstream layers. When the eigenvalues of the weight matrices are small and the derivatives of non-linearities are nearly zero (as they often are for sigmoid and ReLU non-linearities), multiplication by these terms annihilates information. The resulting gradients in shallow lay-

ers contain little information about the error (Bengio et al., 1994; Riedmiller & Braun, 1993; Hochreiter & Schmidhuber, 1997).

Second, backprop has the potential to get stuck at local minima and saddle points. While recent results suggest that local minimizers of SGD are close to global minima (Choromanska et al., 2014), in practice SGD often lingers near saddle points where gradients are small (Dauphin et al., 2014).

Finally, backprop does not easily parallelize over layers, a significant bottleneck when considering deep architectures. However, recent work on SGD has successfully used model parallelism by using multiple replicas of the entire network (Dean et al., 2012).

We propose a solution that helps alleviate these problems by separating the objective function at each layer of a neural network into two terms: one term measuring the relation between the weights and the input activations, and the other term containing the nonlinear activation function. We then apply an alternating direction method that addresses each term separately. The first term allows the weights to be updated without the effects of vanishing gradients. In the second step, we have a non-convex minimization problem that can be solved *globally* in closed-form. Also, the form of the objective allows the weights of every layer to be updated independently, enabling parallelization over layers.

This approach does not require any gradient steps at all. Rather, the problem of training network parameters is reduced to a series of minimization sub-problems using the alternating direction methods of multipliers. These minimization sub-problems are solved globally in closed form.

2.2. Related work

Other works have applied least-squares based methods to neural networks. One notable example is the method of auxiliary coordinates (MAC) (Carreira-Perpiñán & Wang, 2012) which uses quadratic penalties to approximately enforce equality constraints. Unlike our method, MAC requires iterative solvers for sub-problems, whereas the method proposed here is designed so that all sub-problems have closed form solutions. Also unlike MAC, the method proposed here uses Lagrange multipliers to exactly enforce equality constraints, which we have found to be necessary for training deeper networks.

Another related approach is the expectation-maximization (EM) algorithm of (Patel et al., 2015), which is derived from the Deep Rendering Model (DRM), a hierarchical generative model for natural images. They show that feed-forward propagation in a deep convolutional net corresponds to inference on their proposed DRM. They derive a new EM learning algorithm for their proposed DRM that

employs least-squares parameter updates that are conceptually similar to (but different from) the Parallel Weight Update proposed here (see Section 5). However, there is currently no implementation nor any training results to compare against.

Note that our work is the first to consider alternating least squares as a method to distribute computation across a cluster, although the authors of (Carreira-Perpiñán & Wang, 2012) do consider implementations that are "distributed" in the sense of using multiple threads on a single machine via the Matlab matrix toolbox.

3. Alternating minimization for neural networks

The idea behind our method is to decouple the weights from the nonlinear link functions using a splitting technique. Rather than feeding the output of the linear operator W_l directly into the activation function h_l, we store the output of layer l in a new variable $z_l = W_l a_{l-1}$. We also represent the output of the link function as a vector of activations $a_l = h_l(z_l)$. We then wish to solve the following problem

$$
\begin{aligned}
\underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}} \quad & \ell(z_L, y) \\
\text{subject to} \quad & z_l = W_l a_{l-1}, \text{ for } l = 1, 2, \cdots L \\
& a_l = h_l(z_l), \text{ for } l = 1, 2, \cdots L - 1.
\end{aligned}
\tag{3}
$$

Observe that solving (3) is equivalent to solving (2). Rather than try to solve (3) directly, we relax the constraints by adding an ℓ_2 penalty function to the objective and attack the unconstrained problem

$$
\begin{aligned}
\underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}} \quad & \ell(z_L, y) + \beta_L \|z_L - W_L a_{L-1}\|^2 \\
& + \sum_{l=1}^{L-1} \left[\gamma_l \|a_l - h_l(z_l)\|^2 + \beta_l \|z_l - W_l a_{l-1}\|^2 \right]
\end{aligned}
\tag{4}
$$

where $\{\gamma_l\}$ and $\{\beta_l\}$ are constants that control the weight of each constraint. The formulation (4) only *approximately* enforces the constraints in (3). To obtain exact enforcement of the constraints, we add a Lagrange multiplier term to (4), which yields

$$
\begin{aligned}
\underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}} \quad & \ell(z_L, y) \\
& + \langle z_L, \lambda \rangle + \beta_L \|z_L - W_L a_{L-1}\|^2 \\
& + \sum_{l=1}^{L-1} \left[\gamma_l \|a_l - h_l(z_l)\|^2 + \beta_l \|z_l - W_l a_{l-1}\|^2 \right].
\end{aligned}
\tag{5}
$$

where λ is a vector of Lagrange multipliers with the same dimensions as z_L. Note that in a classical ADMM formulation, a Lagrange multiplier would be added for each constraint in (3). The formulation above corresponds more

Algorithm 1 ADMM for Neural Nets
___
Input: training features $\{a_0\}$, and labels $\{y\}$,
Initialize: allocate $\{a_l\}_{l=1}^{L-1}$, $\{z_l\}_{l=1}^{L}$, and λ
repeat
for $l = 1, 2, \cdots, L-1$ **do**
   $W_l \leftarrow z_l a_{l-1}^{\dagger}$
   $a_l \leftarrow (\beta_{l+1} W_{l+1}^T W_{l+1} + \gamma_l I)^{-1} (\beta_{l+1} W_{l+1}^T z_{l+1} + \gamma_l h_l(z_l))$
   $z_l \leftarrow \arg\min_z \gamma_l \|a_l - h_l(z)\|^2 + \beta_l \|z_l - W_l a_{l-1}\|^2$
end for
$W_L \leftarrow z_L a_{L-1}^{\dagger}$
$z_L \leftarrow \arg\min_z \ell(z, y) + \langle z_L, \lambda \rangle + \beta_L \|z - W_L a_{L-1}\|^2$
$\lambda \leftarrow \lambda + \beta_L(z_L - W_L a_{L-1})$
until converged
___

closely to Bregman iteration, which only requires a Lagrange correction to be added to the objective term (and not the constraint terms), rather than classical ADMM. We have found the Bregman formulation to be far more stable than a full scale ADMM formulation. This issue will be discussed in detail in Section 4.

The split formulation (4) is carefully designed to be easily minimized using an alternating direction method in which each sub-step has a simple closed-form solution. The alternating direction scheme proceeds by updating one set of variables at a time – either $\{W_l\}$, $\{a_l\}$, or $\{z_l\}$ – while holding the others constant. The simplicity of the proposed scheme comes from the following observation: The minimization of (4) with respect to both $\{W_l\}$ and $\{a_{l-1}\}$ is a simple linear least-squares problem. Only the minimization of (4) with respect $\{z_l\}$ is nonlinear. However, there is no coupling between the entries of $\{z_l\}$, and so the problem of minimizing for $\{z_l\}$ decomposes into solving a large number of one-dimensional problems, one for each entry in $\{z_l\}$. Because each sub-problem has a simple form and only 1 variable, these problems can be solved *globally* in closed form.

The full alternating direction method is listed in Algorithm 1. We discuss the details below.

3.1. Minimization sub-steps

In this section, we consider the updates for each variable in (5). The algorithm proceeds by minimizing for W_l, a_l, and z_l, and then updating the Lagrange multipliers λ.

Weight update We first consider the minimization of (4) with respect to $\{W_l\}$. For each layer l, the optimal solution minimizes $\|z_l - W_l a_{l-1}\|^2$. This is simply a least squares problem, and the solution is given by $W_l \leftarrow z_l a_{l-1}^{\dagger}$ where $a_{l-1}^{\dagger}$ represents the pseudoinverse of the (rectangular) activation matrix a_{l-1}.

Activations update Minimization for a_l is a simple least-squares problem similar to the weight update. However, in this case the matrix a_l appears in two penalty terms in (4), and so we must minimize $\beta_l \|z_{l+1} - W_{l+1} a_l\| + \gamma_l \|a_l - h_l(z_l)\|^2$ for a_l, holding all other variables fixed. The new value of a_l is given by

$$(\beta_{l+1} W_{l+1}^T W_{l+1} + \gamma_l I)^{-1} (\beta_{l+1} W_{l+1}^T z_{l+1} + \gamma_l h_l(z_l)) \quad (6)$$

where W_{l+1}^T is the adjoint (transpose) of W_{l+1}.

Outputs update The update for z_l requires minimizing

$$\min_z \gamma_l \|a_l - h_l(z)\|^2 + \beta_l \|z - W_l a_{l-1}\|^2. \quad (7)$$

This problem is non-convex and non-quadratic (because of the non-linear term h). Fortunately, because the non-linearity h works entry-wise on its argument, the entries in z_l are de-coupled. Solving (7) is particularly easy when h is piecewise linear, as it can be solved in closed form; common piecewise linear choices for h include rectified linear units (ReLUs) and non-differentiable sigmoid functions given by

$$h_{relu}(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}, h_{sig}(x) = \begin{cases} 1, & \text{if } x \geq 1 \\ x, & \text{if } 0 < x < 1 \\ 0, & \text{otherwise} \end{cases}.$$

For such choices of h, the minimizer of (7) is easily computed using simple if-then logic. For more sophistical choices of h, including smooth sigmoid curves, the problem can be solved quickly with a lookup table of pre-computed solutions because each 1-dimensional problem only depends on two inputs.

Lagrange multiplier update After minimizing for $\{W_l\}$, $\{a_l\}$, and $\{z_l\}$, the Lagrange multiplier update is given simply by

$$\lambda \leftarrow \lambda + \beta_L(z_L - W_L a_{L-1}). \quad (8)$$

We discuss this update further in Section 4.

4. Lagrange multiplier updates via method of multipliers and Bregman iteration

The proposed method can be viewed as solving the constrained problem (3) using Bregman iteration, which is closely related to ADMM. The convergence of Bregman iteration is fairly well understood in the presence of linear constraints (Yin et al., 2008). The convergence of ADMM is fairly well understood for convex problems involving only two separate variable blocks (He & Yuan, 2015). Convergence results also guarantee that a local minima is obtained for two-block non-convex objectives under certain smoothness assumptions (Nocedal & Wright, 2006).

Because the proposed scheme involves more than two coupled variable blocks and a non-smooth penalty function, it lies outside the scope of known convergence results for ADMM. In fact, when ADMM is applied to (3) in a conventional way using separate Lagrange multiplier vectors for each constraint, the method is highly unstable because of the de-stabilizing effect of a large number of coupled, non-smooth, non-convex terms.

Fortunately, we will see below that the Bregman Lagrange update method (8) does not involve any non-smooth constraint terms, and the resulting method seems to be extremely stable.

4.1. Bregman interpretation

Bregman iteration (also known as the method of multipliers) is a general framework for solving constrained optimization problems. Methods of this type have been used extensively in the sparse optimization literature (Yin et al., 2008). Consider the general problem of minimizing

$$\min_u J(u) \quad \text{subject to} \quad Au = b \qquad (9)$$

for some convex function J and linear operator A. Bregman iteration repeatedly solves

$$u^{k+1} \leftarrow \min D_J(u, u^k) + \frac{1}{2}\|Au - b\|^2 \qquad (10)$$

where $p \in \partial J(u^k)$ is a (sub-)gradient of J at u^k, and $D_J(u, u^k) = J(u) - J(u^k) - \langle u - u^k, p \rangle$ is the so-called Bregman distance. The iterative process (10) can be viewed as minimizing the objective J subject to an inexact penalty that approximately obtains $Ax \approx b$, and then adding a linear term to the objective to weaken it so that the quadratic penalty becomes more influential on the next iteration.

The Lagrange update described in Section 3 can be interpreted as performing Bregman iteration to solve the problem (3), where $J(u) = \ell(z_L, y)$, and A contains all the constraints in (3). On each iteration, the outputs z_l are updated immediately before the Lagrange step is taken, and so z_{l-1} satisfies the optimality condition

$$0 \in \partial_z \ell(z_L, y) + \beta_L(z_L - W_L a_{L-1}) + \lambda.$$

It follows that

$$\lambda + \beta_L(z_L - W_L a_{L-1}) \in -\partial_z \ell(z_L, y).$$

For this reason, the Lagrange update (8) can be interpreted as updating the sub-gradient in the Bregman iterative method for solving (3). The combination of the Bregman iterative update with an alternating minimization strategy makes the proposed algorithm an instance of the split Bregman method (Goldstein & Osher, 2009).

4.2. Interpretation as method of multipliers

In addition to the Bregman interpretation, the proposed method can also be viewed as an approximation to the method of multipliers, which solves constrained problems of the form

$$\min_u J(u) \quad \text{subject to} \quad A(u) = b \qquad (11)$$

for some convex function J and (possibly non-linear) operator $A(\cdot)$. In its most general form (which does not assume linear constraints) the method proceeds using the iterative updates

$$\begin{cases} u^{k+1} & \leftarrow \min J(u) + \langle \lambda^k, A(u) - b \rangle + \frac{\beta}{2}\|A(u) - b\|^2 \\ \lambda^{k+1} & \leftarrow \lambda^k + \partial_u\{\frac{\beta}{2}\|A(u) - b\|^2\} \end{cases}$$

where λ^k is a vector of Lagrange multipliers that is generally initialized to zero, and $\frac{\beta}{2}\|A(u) - b\|^2$ is a quadratic penalty term. After each minimization sub-problem, the gradient of the penalty term is added to the Lagrange multipliers. When the operator A is linear, this update takes the form $\lambda^{k+1} \leftarrow \lambda^k + \beta A^T(Au - b)$, which is the most common form of the method of multipliers.

Just like in the Bregman case, we now let $J(u) = \ell(z_L, y)$, and let A contain the constraints in (3). After a minimization pass, we must update the Lagrange multiplier vector. Assuming a good minimizer has been achieved, the derivative of (5) should be nearly zero. All variables *except* z_L appear only in the quadratic penalty, and so these derivatives should be negligibly small. The only major contributor to the gradient of the penalty term is z_L, which appears in both the loss function and the quadratic penalty. The gradient of the penalty term with respect to z_L, is $\beta_L(z_L - W_L a_{L-1})$, which is exactly the proposed multiplier update.

When the objective is approximately minimized by alternately updating separate blocks of variables (as in the proposed method), this becomes an instance of the ADMM (Boyd et al., 2011).

5. Distributed implementation using data parallelism

The main advantage of the proposed alternating minimization method is its high degree of scalability. In this section, we explain how the method is distributed.

Consider distributing the algorithm across N worker nodes. The ADMM method is scaled using a data parallelization strategy, in which different nodes store activations and outputs corresponding to different subsets of the training data. For each layer, the activation matrix is broken into columns subsets as $a_i = (a_1, a_2, \cdots, a_N)$. The output matrix z_l and Lagrange multipliers λ decompose similarly.

The optimization sub-steps for updating $\{a_l\}$ and $\{z_l\}$ do not require any communication and parallelize trivially. The weight matrix update requires the computation of pseudo-inverses and products involving the matrices $\{a_l\}$ and $\{z_l\}$. This can be done effectively using *transpose reduction* strategies (Goldstein et al., 2016) that reduce the dimensionality of matrices before they are transmitted to a central node.

Parallel Weight update The weight update has the form $W_l \leftarrow z_l a_l^\dagger$, where $a_l^\dagger$ represents the pseudoinverse of the activation matrix a_l. This pseudoinverse can be written $a_l^\dagger = a_l^T(a_l a_l^T)^{-1}$. Using this expansion, the W update decomposes across nodes as

$$W_l \leftarrow \left(\sum_{n=1}^{N} z_l^n(a_l^n)^T\right)\left(\sum_{n=1}^{N} a_l^n(a_l^n)^T\right)^{-1}.$$

The individual products $z_l^n(a_l^n)^T$ and $a_l^n(a_l^n)^T$ are computed separately on each node, and then summed across nodes using a single reduce operation. Note that the width of a_l^n equals the number of training vectors that are stored on node n, which is potentially very large for big data sets. When the number of features (the number of rows in a_l^n) is less than the number of training data (columns of a_l^n), we can exploit transpose reduction when forming these products – the product $a_l^n(a_l^n)^T$ is much smaller than the matrix a_l^n alone. This dramatically reduces the quantity of data transmitted during the reduce operation.

Once these products have been formed and reduced onto a central server, the central node computes the inverse of $a_l a_l^T$, updates W_l, and then broadcasts the result to the worker nodes.

Parallel Activations update The update (6) trivially decomposes across workers, with each worker computing

$$a_l^n \leftarrow (\beta_{l+1}W_{l+1}^T W_{l+1}+\gamma I)^{-1}(\beta_{l+1}W_{l+1}^T z_{l+1}^n+\gamma_l h_l(z_l^n)).$$

Each server maintains a full representation of the entire weight matrix, and can formulate its own local copy of the matrix inverse $(\beta_{l+1}W_{l+1}^T W_{l+1} + \gamma I)^{-1}$.

Parallel Outputs update Like the activations update, the update for z_l trivially parallelizes and each worker node solves

$$\min_{z_l^n} \gamma_l\|a_l^n - h_l(z_l^n)\|^2 + \beta_l\|z_l^n - W_l a_{l-1}^n\|^2. \quad (12)$$

Each worker node simply computes $W_l a_{l-1}^n$ using local data, and then updates each of the (decoupled) entries in z_l^n by solving a 1-dimensional problem in closed form.

Parallel Lagrange multiplier update The Lagrange multiplier update also trivially splits across nodes, with worker n computing

$$\lambda^n \leftarrow \lambda^n + \beta_L(z_L^n - W_L a_{L-1}^n) \quad (13)$$

using only local data.

6. Implementation details

Like many training methods for neural networks, the ADMM approach requires several tips and tricks to get maximum performance. The convergence theory for the method of multipliers requires a good minimizer to be computed before updating the Lagrange multipliers. When the method is initialized with random starting values, the initial iterates are generally far from optimal. For this reason, we frequently "warm start" the ADMM method by running several iterations without Lagrange multiplier updates.

The method potentially requires the user to choose a large number of parameters $\{\gamma_i\}$ and $\{\beta_i\}$. We choose $\gamma_i = 10$ and $\beta_i = 1$ for all trials runs reported here, and we have found that this choice works reliably for a wide range of problems and network architectures. Note that in the classical ADMM method, convergence is guaranteed for any choice of the quadratic penalty parameters.

We use training data with binary class labels, in which each output entry a_L is either 1 or 0. We use a separable loss function with a hinge penalty of the form

$$\ell(z, a) = \begin{cases} \max\{1 - z, 0\}, & \text{when } a = 1, \\ \max\{a, 0\}, & \text{when } a = 0. \end{cases}$$

This loss function works well in practice, and yields minimization sub-problems that are easily solved in closed form.

Finally, our implementation simply initializes the activation matrices $\{a_l\}$ and output matrices $\{z_l\}$ using i.i.d Gaussian random variables. Because our method updates the weights before anything else, the weight matrices do not require any initialization. The results presented here are using Gaussian random variables with unit variance, and the results seem to be fairly insensitive to the variance of this distribution. This seems to be because the output updates are solved to global optimality on each iteration.

7. Experiments

In this section, we present experimental results that compare the performance of the ADMM method to other approaches, including SGD, conjugate gradients, and L-BFGS on benchmark classification tasks. Comparisons are made across multiple axes. First, we illustrate the scaling
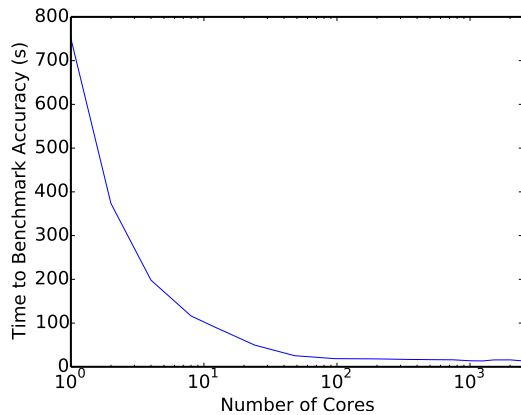
(a) **Time required for ADMM to reach 95% test accuracy vs number of cores**. This problem was not large enough to support parallelization over many cores, yet the advantages of scaling are still apparent (note the x-axis has log scale). In comparison, on the GPU, L-BFGS reached this threshold in 3.2 seconds, CG in 9.3 seconds, and SGD in 8.2 seconds.
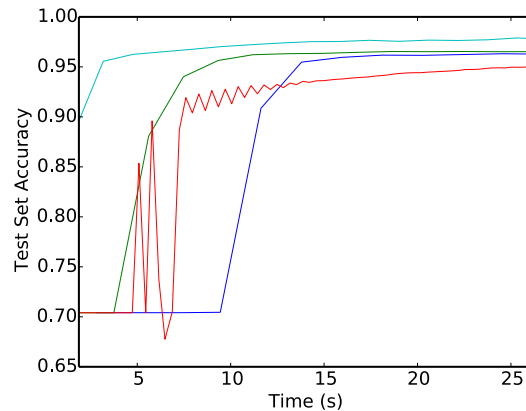
(b) **Test set predictive accuracy as a function of time in seconds** for ADMM on 2,496 cores (blue), in addition to GPU implementations of conjugate gradients (green), SGD (red), and L-BFGS (cyan).

Figure 1. Street View House Numbers (subsection 7.1)

of the approach, by varying the number of cores available and clocking the compute time necessary to meet an accuracy threshold on the test set of the problem. Second, we show test set classification accuracy as a function of time to compare the rate of convergence of the optimization methods. Finally, we show these comparisons on two different data sets, one small and relatively easy, and one large and difficult.

The new ADMM approach was implemented in Python on a Cray XC30 supercomputer with Ivy Bridge processors, and communication between cores performed via MPI. SGD, conjugate gradients, and L-BFGS are run as implemented in the Torch optim package on NVIDIA Tesla K40 GPUs. These methods underwent a thorough hyperparameter grid search to identify the algorithm parameters that produced the best results. In all cases, timings indicate only the time spent optimizing, excluding time spent loading data and setting up the network.

Experiments were run on two datasets. The first is a subset of the Street View House Numbers (SVHN) dataset (Netzer et al., 2011). Neural nets were constructed to classify pictures of 0s from 2s using histogram of gradient (HOG) features of the original dataset. Using the "extra" dataset to train, this meant 120,290 training datapoints of 648 features each. The testing set contained 5,893 data points.

The second dataset is the far more difficult Higgs dataset (Baldi et al., 2014), consisting of a training set of 10,500,000 datapoints of 28 features each, with each datapoint labelled as either a signal process producing a Higgs boson or a background process which does not. The testing set consists of 500,000 datapoints.

7.1. SVHN

First, we focus on the problem posed by the SVHN dataset. For this dataset, we optimized a net with two hidden layers of 100 and 50 nodes and ReLU activation functions. This is an easy problem (test accuracy rises quickly) that does not require a large volume of data and is easily handled by gradient-based methods on a GPU. However, Figure 1a demonstrates that ADMM exhibits linear scaling with cores. Even though the implementations of the gradient-based methods enjoy communication via shared memory on the GPU while ADMM required CPU-to-CPU communication, strong scaling allows ADMM on CPU cores to compete with the gradient-based methods on a GPU.

This is illustrated clearly in Figure 1b, which shows each method's performance on the test set as a function of time. With 1,024 compute cores, on an average of 10 runs, ADMM was able to meet the 95% test set accuracy threshold in 13.3 seconds. After an extensive hyperparameter search to find the settings which resulted in the fastest convergence, SGD converged on average in 28.3 seconds, L-BFGS in 3.3 seconds, and conjugate gradients in 10.1 seconds. Though the small dataset kept ADMM from taking full advantage of its scalability, it was nonetheless sufficient to allow it to be competitive with GPU implementations.

7.2. Higgs

For the much larger and more difficult Higgs dataset, we optimized a simple network with ReLU activation functions and a hidden layer of 300 nodes, as suggested in (Baldi et al., 2014). The graph illustrates the amount of time required to optimize the network to a test set pre-
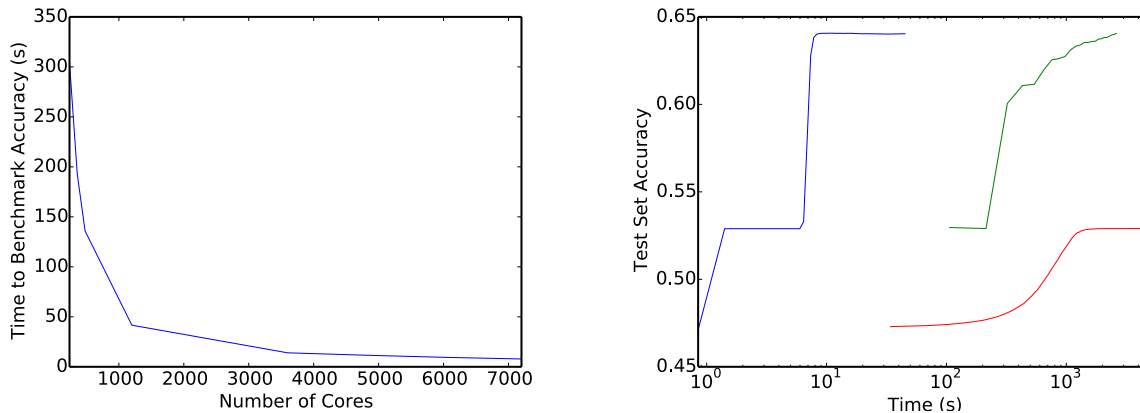
(a) **Time required for ADMM to reach 64% test accuracy when parallelized over varying levels of cores**. L-BFGS on a GPU required 181 seconds, and conjugate gradients required 44 minutes. SGD never reached 64% accuracy.

(b) **Test set predictive accuracy as a function of time** for ADMM on 7200 cores (blue), conjugate gradients (green), and SGD (red). Note the x-axis is scaled logarithmically.

Figure 2. Higgs (subsection 7.2)

diction accuracy of 64%; this parameter was chosen as all batch methods being tested reliably hit this accuracy benchmark over numerous trials. As is clear from Figure 2a, parallelizing over additional cores decreases the time required dramatically, and again exhibits linear scaling.

In this much larger problem, the advantageous scaling allowed ADMM to reach the 64% benchmark much faster than the other approaches. Figure 2b illustrates this clearly, with ADMM running on 7200 cores reaching this benchmark in 7.8 seconds. In comparison, L-BFGS required 181 seconds, and conjugate gradients required 44 minutes.[1] In seven hours of training, SGD never reached 64% accuracy on the test set. These results suggest that, for large and difficult problems, the strong linear scaling of ADMM enables it to leverage large numbers of cores to (dramatically) out-perform GPU implementations.

8. Discussion & Conclusion

We present a method for training neural networks without using gradient steps. In addition to avoiding many difficulties of gradient methods (like saturation and choice of learning rates), performance of the proposed method scales linearly up to thousands of cores. This strong scaling enables the proposed approach to out-perform other methods on problems involving extremely large datasets.

---

[1]It is worth noting that though L-BFGS required substantially more time to reach 64% than did ADMM, it was the only method to produce a superior classifier, doing as well as 75% accuracy on the test set.

8.1. Looking forward

The experiments shown here represent a fairly narrow range of classification problems and are not meant to demonstrate the absolute superiority of ADMM as a training method. Rather, this study is meant to be a proof of concept demonstrating that the caveats of gradient-based methods can be avoided using alternative minimization schemes. Future work will explore the behavior of alternating direction methods in broader contexts.

We are particularly interested in focusing future work on recurrent nets and convolutional nets. Recurrent nets, which complicate standard gradient methods (Jaeger, 2002; Lukoševičius, 2012), pose no difficulty for ADMM schemes whatsoever because they decouple layers using auxiliary variables. Convolutional networks are also of interest because ADMM can, in principle, handle them very efficiently. When the linear operators $\{W_l\}$ represent convolutions rather than dense weight matrices, the least squares problems that arise in the updates for $\{W_l\}$ and $\{a_l\}$ can be solved efficiently using fast Fourier transforms.

Finally, there are avenues to explore to potentially improve convergence speed. These include adding momentum terms to the weight updates and studying different initialization schemes, both of which are known to be important for gradient-based schemes (Sutskever et al., 2013).

Acknowledgements

References

Baldi, Pierre, Sadowski, Peter, and Whiteson, Daniel. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5, 2014.

Bengio, Yoshua, Simard, Patrice, and Frasconi, Paolo. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5 (2):157–166, 1994.

Boyd, Stephen, Parikh, Neal, Chu, Eric, Peleato, Borja, and Eckstein, Jonathan. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1–122, 2011.

Carreira-Perpinán, Miguel A and Wang, Weiran. Distributed optimization of deeply nested systems. *arXiv preprint arXiv:1212.5921*, 2012.

Choromanska, Anna, Henaff, Mikael, Mathieu, Michael, Arous, Gérard Ben, and LeCun, Yann. The loss surfaces of multilayer networks. *arXiv preprint arXiv:1412.0233*, 2014.

Dauphin, Yann N, Pascanu, Razvan, Gulcehre, Caglar, Cho, Kyunghyun, Ganguli, Surya, and Bengio, Yoshua. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pp. 2933–2941, 2014.

Dean, Jeffrey, Corrado, Greg, Monga, Rajat, Chen, Kai, Devin, Matthieu, Mao, Mark, Ranzato, Marc'aurelio, Senior, Andrew, Tucker, Paul, Yang, Ke, Le, Quoc V., and Ng, Andrew Y. Large scale distributed deep networks. In Pereira, F., Burges, C.J.C., Bottou, L., and Weinberger, K.Q. (eds.), *Advances in Neural Information Processing Systems 25*, pp. 1223–1231. Curran Associates, Inc., 2012.

Goldstein, Tom and Osher, Stanley. The split bregman method for l1-regularized problems. *SIAM Journal on Imaging Sciences*, 2(2):323–343, 2009.

Goldstein, Tom, Taylor, Gavin, Barabin, Kawika, and Sayre, Kent. Unwrapping admm: Efficient distributed computing via transpose reduction. In *19th International Conference on Artificial Intelligence and Statistics(AISTATS)*, 2016.

He, Bingsheng and Yuan, Xiaoming. On non-ergodic convergence rate of douglas–rachford alternating direction method of multipliers. *Numerische Mathematik*, 130(3): 567–577, 2015.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Jaeger, Herbert. *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*. GMD-Forschungszentrum Informationstechnik, 2002.

Lukoševičius, Mantas. A practical guide to applying echo state networks. In *Neural Networks: Tricks of the Trade*, pp. 659–686. Springer, 2012.

Martens, James and Sutskever, Ilya. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1033–1040, 2011.

Møller, Martin Fodslette. A scaled conjugate gradient algorithm for fast supervised learning. *Neural networks*, 6 (4):525–533, 1993.

Nair, Vinod and Hinton, Geoffrey E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 807–814, 2010.

Netzer, Yuval, Wang, Tao, Coates, Adam, Bissacco, Alessandro, Wu, Bo, and Ng, Andrew Y. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, pp. 4. Granada, Spain, 2011.

Ngiam, Jiquan, Coates, Adam, Lahiri, Ahbik, Prochnow, Bobby, Le, Quoc V, and Ng, Andrew Y. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 265–272, 2011.

Nocedal, Jorge and Wright, Stephen. *Numerical optimization*. Springer Science & Business Media, 2006.

Patel, Ankit B, Nguyen, Tan, and Baraniuk, Richard. A probabilistic theory of deep learning. *arXiv preprint arXiv:1504.00641*, 2015.

Riedmiller, Martin and Braun, Heinrich. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pp. 586–591. IEEE, 1993.

Sainath, Tara N, Horesh, Lior, Kingsbury, Brian, Aravkin, Aleksandr Y, and Ramabhadran, Bhuvana. Accelerating hessian-free optimization for deep neural networks by implicit preconditioning and sampling. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*, pp. 303–308. IEEE, 2013.

Sutskever, Ilya, Martens, James, Dahl, George, and Hinton, Geoffrey. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pp. 1139–1147, 2013.

Towsey, Michael, Alpsan, Dogan, and Sztriha, Laszlo. Training a neural network with conjugate gradient methods. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 1, pp. 373–378. IEEE, 1995.

Yin, Wotao, Osher, Stanley, Goldfarb, Donald, and Darbon, Jerome. Bregman iterative algorithms for \ell_1-minimization with applications to compressed sensing. *SIAM Journal on Imaging Sciences*, 1(1):143–168, 2008.

Zhang, Sixin, Choromanska, Anna E, and LeCun, Yann. Deep learning with elastic averaging sgd. In *Advances in Neural Information Processing Systems*, pp. 685–693, 2015.