Training Neural Networks in Parallel With ADMM

Nicholas Haltmeyer
hanicho1@umbc.edu

Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County

Abstract

Neural networks are an increasingly popular model for solving classification problems, including optical character recognition. As these networks have become deeper and more rich in output space, alternatives to classical training methods have been proposed to allow for scalable, distributed learning. Herein I describe an application of the alternating direction method of multipliers for training neural networks in a distributed environment. Experiments are then presented for determining appropriate settings and evaluating performance on the MNIST database of handwritten digits.

Key words. ADMM, Bregman iteration, Deep learning, MPI.

AMS subject classifications (2010). 90C56, 68T05, 68Q85

1 Introduction

Artificial neural networks are a class of graphical models that use a connectionist approach to perform classification in a way that is loosely analogous to the human brain. A feedforward network of this type is a composition of directed, bipartite graphs known as a *layers*. The first and final set of vertices belong to the input and output *units*. Each input is multiplied by the value associated with each of its edges, known as *weights*. Following this, all incoming values to each vertex are summed and given to a non-linear *activation function* that produces input for the next layer. The learning problem then is to find the optimal real-valued weight for each edge of the graph. Figure 1.1 is an example of one possible feedforward network configuration.

Conventional methods of learning these weights, such as gradient descent with backpropagation (Rumelhart et al., 1988), are not easily distributed as they rely on a large number of computationally cheap steps. This property makes it that for any sufficiently large factor of parallelization, the cost of communication is dominant. It is then preferable that in a distributed setting, a small number of computationally expensive steps be used instead. The alternating direction method of multipliers (ADMM) can be used to scalably train neural networks in this manner (Taylor et al., 2016).

2 Methodology

Using the notation of Taylor et al., a neural network consists of L layers, each having a linear operator, W_l, and a non-linear activation function, h_l, where $l = 1, 2, \ldots, L$. Given the input

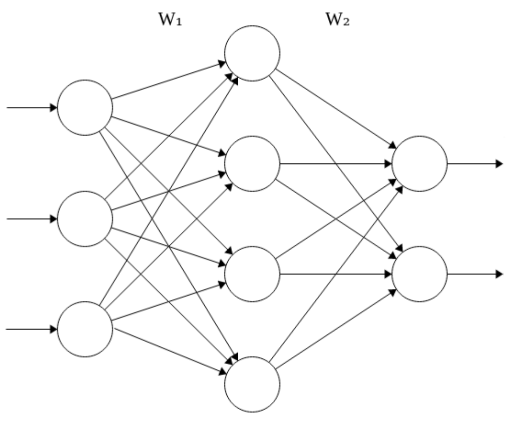

Figure 1.1: Example 2-layer feedforward network with three input units, four hidden units, and two output units.

activations, a_{l-1}, a layer produces the output activations, $a_l = h_l(W_l a_{l-1})$. Final output of the network is the composition of this operation over all layers, as given by

$$f(a_0; W) = W_L h_{L-1}(\dots W_2 h_1(W_1 a_0)), \tag{2.1}$$

where $W = \{W_l\}$ is the set of weight matrices and a_0 is the input for the training samples.

Given some cost function, C, the learning problem then becomes

$$\underset{W}{\text{minimize}} \quad C(f(a_0; W), y), \tag{2.2}$$

where y is the target output for the training samples.

2.1 Backpropagation

The standard serial method for training a neural network is backpropagation with gradient descent. In this method, the gradient of the cost function is computed and propagated backwards from the output layer ($l = L$) to the input layer ($l = 1$). The update rule for each layer is

$$W_l \leftarrow W_l - \alpha \frac{\delta C}{\delta W_l}, \tag{2.3}$$

where α is the learning rate constant.

The problems with this method are twofold: it has issues in not being efficiently distributed and in not finding minima for sufficiently deep networks, where deep here refers to having an arbitrarily large value for L. The latter issue arises from the vanishing gradient problem, where information is lost while propagating errors. This occurs because the gradients of earlier ($l \ll L$) weights are formed from the product of their weight matrices and the gradients of later ($l \simeq L$) weights. When the eigenvalues of earlier weight matrices are small and the gradients of later weight matrices are close to zero, learning is significantly slowed.

2.2 Alternating Direction Method of Multipliers

The alternating direction method of multipliers is a constrained optimization method that works by splitting the variables of the original problem and imposing an equality constraint on the split variables (Lions and Mercier, 1979). This allows the problem

$$\underset{x}{\text{minimize}} \quad f(x) + g(x), \tag{2.4}$$

to instead take the form

$$\begin{aligned} &\underset{x,y}{\text{minimize}} && f(x) + g(y) \\ &\text{subject to} && x = y. \end{aligned} \tag{2.5}$$

The split minimization problem can be solved for x with y constant, then for y with x constant, and so on to convergence.

In order to apply ADMM to learning neural networks, the minimization problem of (2.2) must be reformulated by decoupling the weights from the activation functions. This is done by introducing a term, $z_l = W_l a_{l-1}$, and redefining the activations as $a_l = h_l(z_l)$. The learning problem is then

$$\begin{aligned} &\underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}} && C(z_L, y) \\ &\text{subject to} && z_l = W_l a_{l-1}, \qquad \text{for } l = 1, 2, \dots, L \\ & && a_l = h_l(z_l), \qquad \text{for } l = 1, 2, \dots, L-1. \end{aligned} \tag{2.6}$$

To solve (2.6), the constraints are relaxed by adding an ℓ^2 penalty term, where $||A|| = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{i,j}^2}$, for $A \in \mathbb{R}^{m \times n}$. The problem then becomes

$$\begin{aligned} \underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}} \quad & C(z_L, y) + \beta_L ||z_L - W_L a_{L-1}||^2 + \\ & \sum_{l=1}^{L-1} \big[\gamma_l ||a_l - h_l(z_l)||^2 + \beta_l ||z_l - W_l a_{l-1}||^2\big], \end{aligned} \tag{2.7}$$

where $\{\gamma_l\}$, $\{\beta_l\}$ are constants biasing each constraint. The values of $\{\gamma_l\}$, $\{\beta_l\}$ are analogous to the learning rate in backpropagation. As (2.7) only approximately enforces the constraints, a Lagrange multiplier term is added to give

$$\begin{aligned} \underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}} \quad & C(z_L, y) + \langle z_L, \lambda \rangle + \beta_L ||z_L - W_L a_{L-1}||^2 + \\ & \sum_{l=1}^{L-1} \big[\gamma_l ||a_l - h_l(z_l)||^2 + \beta_l ||z_l - W_l a_{l-1}||^2\big], \end{aligned} \tag{2.8}$$

where λ is the Lagrange multiplier. This formulation, given by Taylor et al., differs from classical ADMM in its use of a Lagrange term for only the cost function, rather than including a separate term for each constraint. This closely resembles Bregman iteration and is chosen for its numerical stability over the alternative. This instability in the classical ADMM formulation is due to the inclusion of multiple non-smooth, non-convex terms.

3 Numerical Method

Given the formulation in (2.8), alternating direction updates are applied to modify a single parameter, while leaving the rest constant. This process is repeated until convergence. The minimization of $\{W_l\}$ and $\{a_l\}$ are least squares problems. The component terms in minimizing $\{z_l\}$ are decoupled, so $\{z_l\}$ can be globally minimized through each one-dimensional component. Appropriate cost and activation functions are chosen so that this can be done easily in closed form. The final algorithm can be seen in Algorithm 1.

3.1 Weights Updates

The update rule for weights is given by

$$W_l \leftarrow \arg\min_W ||z_l - W a_{l-1}||^2. \tag{3.1}$$

This is a least squares problem with a solution in

$$W_l \leftarrow z_l a_{l-1}^{\dagger}. \tag{3.2}$$

Note that the pseudoinverse $a_{l-1}^{\dagger} = a_{l-1}^T (a_{l-1} a_{l-1}^T)^{-1}$.

3.2 Activation Updates

The update rule for activations is given by

$$a_l \leftarrow \arg\min_a \beta_l ||z_{l+1} - W_{l+1} a||^2 + \gamma_l ||a - h_l(z_l)||^2. \tag{3.3}$$

The solution to this least squares problem is found in

$$a_l \leftarrow (\beta_{l+1} W_{l+1}^T W_{l+1} + \gamma_l I)^{-1} (\beta_{l+1} W_{l+1}^T z_{l+1} + \gamma_l h_l(z_l)). \tag{3.4}$$

3.3 Output Updates

The update rule for outputs on layers $l = 1, 2, \ldots, L-1$ is given by

$$z_l \leftarrow \arg\min_z \gamma_l ||a_l - h_l(z)||^2 + \beta_l ||z - W_l a_{l-1}||^2. \tag{3.5}$$

Since the activation function, h_l, is an entry-wise mapping over each component of z, this can be minimized for each one-dimensional component. A piecewise sigmoid activation function is used to facilitate a closed form solution. This activation function is given by

$$h(x) = \begin{cases} 1 & x \geq 1, \\ x & 0 < x < 1, \\ 0 & x \leq 0. \end{cases}$$

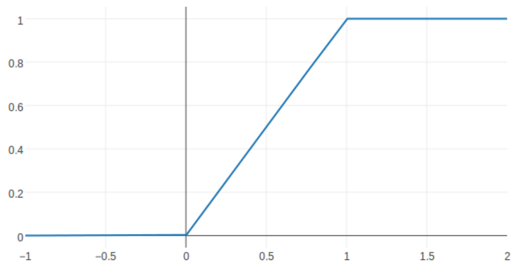

Figure 3.1: Discrete sigmoid, $h(x)$.

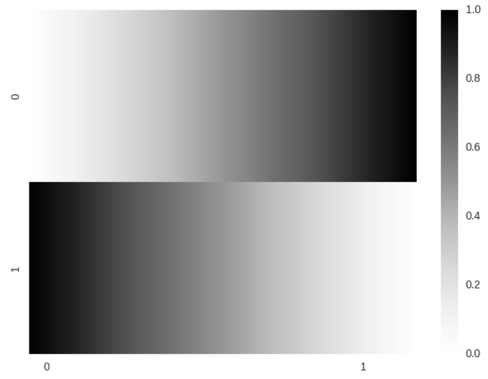

Figure 3.2: Hinge, $C(x, y)$.

A plot of this activation function can be seen in Figure 3.1.

The update rule for outputs on layer $l = L$ is given by

$$z_L \leftarrow \arg\min_z C(z, y) + \langle z, \lambda \rangle + \beta_L ||z - W_L a_{L-1}||^2. \tag{3.6}$$

Similar to minimizing earlier layers, a cost function must be chosen so that this can be solved easily in closed form. An appropriate function for this is hinge cost, given by

$$C(x, y) = \begin{cases} \max(0, 1 - x) & y = 1, \\ \max(0, x) & y = 0. \end{cases}$$

A heatmap of this cost function can be seen in Figure 3.2.

3.4 Lagrange Multiplier Updates

After all parameters have been updated, the Lagrange term is updated by

$$\lambda \leftarrow \lambda + \beta_L (z_L - W_L a_{L-1}). \tag{3.7}$$

In practice, a small number of iterations are completed before the Lagrange terms are first updated. This is known as a warm start, and its use speeds up learning, as the immediate values for the parameters used in initializing λ can be noisy.

3.5 Parallelization

Learning is done in parallel by distributing the training samples across N nodes. The terms $\{a_l\}$, $\{z_l\}$, λ are broken apart by column, with a factor of division equal to the number of nodes. Each node maintains a local copy of the complete set of weight matrices, $\{W_l\}$. By this scheme, the only update that differs in a parallel setting is the weight update. All other

Algorithm 1 ADMM for ANNs

```
1: procedure ADMM-Learn(features $a_0$, labels $y$)
2:     repeat
3:         for $l = 1, \cdots, L-1$ do
4:             $W_l \leftarrow z_l a_{l-1}^{\dagger}$
5:             $a_l \leftarrow (\beta_l W_{l+1}^T W_{l+1} + \gamma_l I)^{-1}(\beta_l W_{l+1}^T z_{l+1} + \gamma_l h_l(z_l))$
6:             $z_l \leftarrow \arg\min_z \gamma_l ||a_l - h_l(z)||^2 + \beta_l ||z - W_l a_{l-1}||^2$
7:         $W_L \leftarrow z_L a_{L-1}^{\dagger}$
8:         $z_L \leftarrow \arg\min_z C(z,y) + \langle z, \lambda \rangle + \beta_L ||z - W_L a_{L-1}||^2$
9:         $\lambda \leftarrow \lambda + \beta_L (z_L - W_L a_{L-1})$
10:    until converged
```

updates are performed exactly as in the serial method, operating instead on local subsets of the data.

As weights are stored in full on every node, updates must be communicated. Using the strategy of transpose reduction (Goldstein et al., 2015), the size of this communication is reduced by computing the update subterms on each node, then taking their sum before computing the final step. The update $W_l \leftarrow z_l a_{l-1}^T (a_{l-1} a_{l-1}^T)^{-1}$ parallelizes as

$$W_l \leftarrow \left(\sum_{n=1}^{N} z_l^n (a_{l-1}^n)^T \right) \left(\sum_{n=1}^{N} a_{l-1}^n (a_{l-1}^n)^T \right)^{-1}. \tag{3.8}$$

This is done by first computing each parenthesized term on every node. Following, `MPI_Allreduce` is called on both terms with the `MPI_SUM` operator. The inverse of the second term is computed on all nodes, followed by its product with the first term.

4 Experiments

For evaluating this method, the MNIST (Lecun et al., 1998) dataset of handwritten digits is used for training and validation. This dataset consists of grayscale pixel data (0 to 255) for 28×28 pixel images of digits and their corresponding labels (0 to 9). From this, networks used in evaluation have 784 input units and 10 output units. Before being fed to a network, input values are normalized to lie between 0 and 1.

The dataset consists of 60,000 training samples and 10,000 validation samples. The error of a network is determined by first learning the model with the 60,000 training samples, then evaluating the top selected categorization for each of the 10,000 validation samples. Final error is given as a value between 0 and 1, denoting the fraction of validation samples incorrectly categorized.

4.1 Settings

Experiments were written in the C programming language, utilizing the GNU Scientific Library and MPI[1]. Code was run using Intel MPI, version 5.1.3 build 20160120, and compiled with `icc`, version 16.0.3 build 20160415. The `icc` compiler directive `-O3` was used to perform optimization.

Experiments were run on the maya compute cluster[2], utilizing a portion consisting of 72 nodes, each with 64GB memory and two 8 core Intel E5-2650v2 Ivy Bridge CPUs (2.6 GHz, 20 MB cache).

Network communication on the cluster is performed using low latency, wide bandwidth InfiniBand interconnect systems. This gives, ideally, a latency of 1.2 µs in transferring a message between two nodes, with bandwidth of up to 3.5 GB/s.

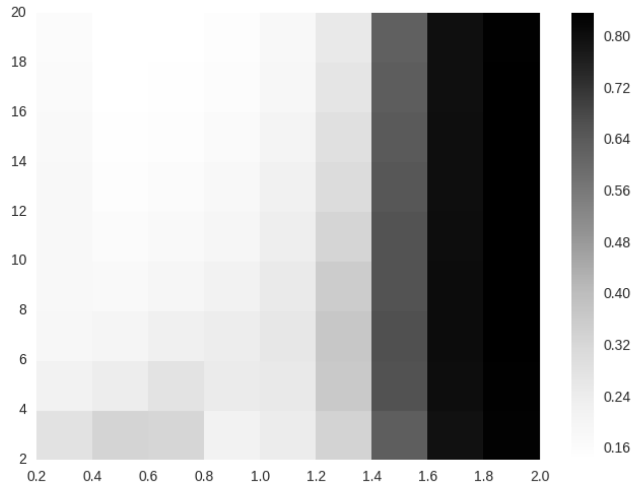

Figure 4.1: Error for values of β, γ.

4.2 Hyperparameter Tuning

Weight matrices are first set through a normalized initialization scheme (Glorot and Bengio, 2010), given by

$$W_l \sim U[-\frac{\sqrt{6}}{\sqrt{n_l + m_l}}, \frac{\sqrt{6}}{\sqrt{n_l + m_l}}], \tag{4.1}$$

where n_l, m_l are the number of input and output units for layer l. This initialization scheme helps prevent any particular weight from being over-saturated and skewing results when the other weights of that layer are appropriately set.

[1]https://github.com/hanicho/admm-nn
[2]http://hpcf.umbc.edu/system-description/

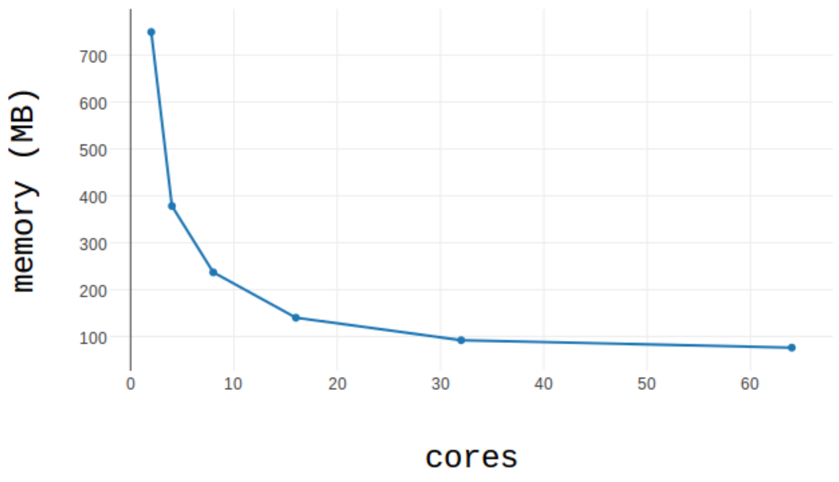

Figure 4.2: Mean resident memory usage for each node (excluding the root).

The values of $\{\beta_l\}$, $\{\gamma_l\}$ were chosen to be constant for all layers. These global constants were then evaluated for a 3-layer network configuration ($784 \times 500 \times 500 \times 10$) and results can be seen in the heatmap of Figure 4.1. From this, the values $\beta = 0.2$ and $\gamma = 20$ were chosen for testing.

The number of iterations taken before updating λ is not nearly as sensitive as the choices of β, γ, as few iterations need to be taken before z_L, W_L, and a_{L-1} can initialize λ to relative stability. For all tests, 10 iterations are taken before updating λ.

Table 4.1: Memory usage per node (MB).

N	Predicted	Observed
1	979.58	1914.98
2	492.38	749.27
4	248.78	378.30
8	126.98	236.91
16	66.08	140.17
32	35.63	92.18
64	20.40	76.35

Table 4.2: Memory usage over all nodes (MB).

N	Predicted	Observed
1	979.58	1914.98
2	984.75	1930.69
4	995.10	1948.09
8	1015.81	2332.35
16	1057.22	2679.99
32	1140.03	3386.55
64	1305.66	5315.53

4.3 Memory Usage

For parallel execution of ADMM, local copies of the full set of weight matrices are stored on each node. Each node stores a portion of the other variables, $\{a_l\}$, $\{z_l\}$, λ. In all tests using ADMM, a 3-layer network configuration ($784 \times 500 \times 500 \times 10$) is used.

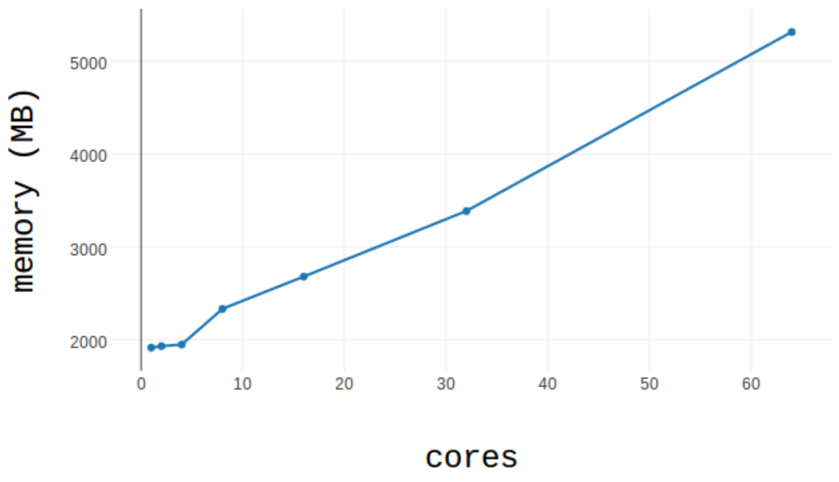

Figure 4.3: Mean resident memory usage over all nodes (including the root).

Each node has $784 \times 500 + 500 \times 500 + 500 \times 10 = 647,000$ values in the local set of weight matrices. As each value is stored as a C-language `double float`, the memory used on each node for the weight matrices is $8 \times 647,000 = 5,176,000$ bytes.

For $\{a_l\}$, $\{z_l\}$, and λ, each node stores a portion of each according to the number of nodes used. Given 60,000 samples used over N nodes, local memory usage of activations is

$$8 \times \left(\frac{60,000}{N} \times 500 + \frac{60,000}{N} \times 500 + \frac{60,000}{N} \times 10\right) = \frac{484,800,000}{N} \tag{4.2}$$

bytes. The local memory usage of the outputs is equal to that used for activations. For λ, memory usage is the number of local samples multiplied by the number of outputs in the final layer,

$$8 \times \left(\frac{60,000}{N} \times 10\right) = \frac{4,800,000}{N} \tag{4.3}$$

bytes. This makes the total memory usage for any particular node

$$5,176,000 + 2 \times \left(\frac{484,800,000}{N}\right) + \frac{4,800,000}{N} = +5,176,000 + \frac{974,400,000}{N} \tag{4.4}$$

bytes ($5.176 + \frac{974.4}{N}$ MB).

In practice, more memory is used to store ancillary data during computation. In particular, the root node maintains a complete copy of the training and validation samples for evaluating fitness of the model. The average case memory usage on each node (excluding the root) can be seen in Figure 4.2. The average case memory usage for all nodes (including the root) can be seen in Figure 4.3. Table 4.1 contains the predicted and observed memory usage per node. Table 4.2 contains the predicted and observed memory usage over all nodes.

These memory results reflect expected behavior and demonstrate how memory use per node decreases as the number of nodes increases. For sufficiently large datasets, it would be necessary to only maintain the entire dataset when initializing the method, so that the root node does not include the excess seen here.

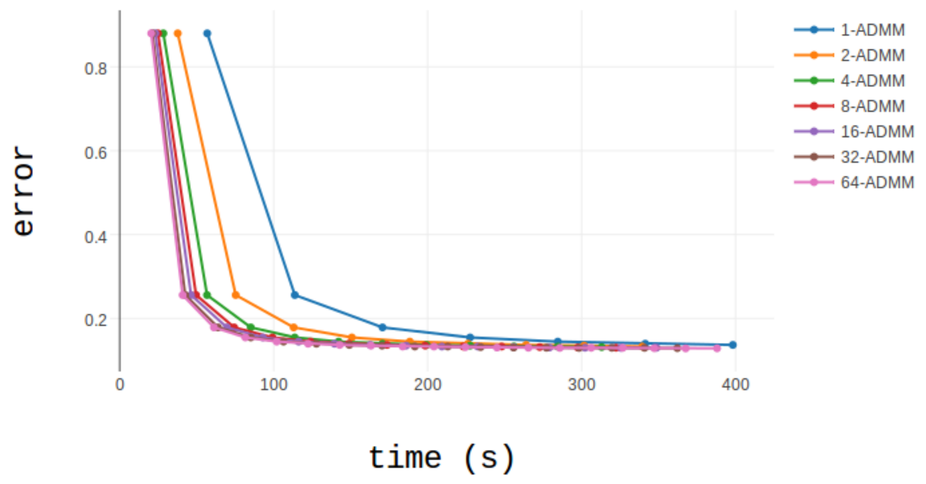

Figure 4.4: Error with respect to time for different values of N.

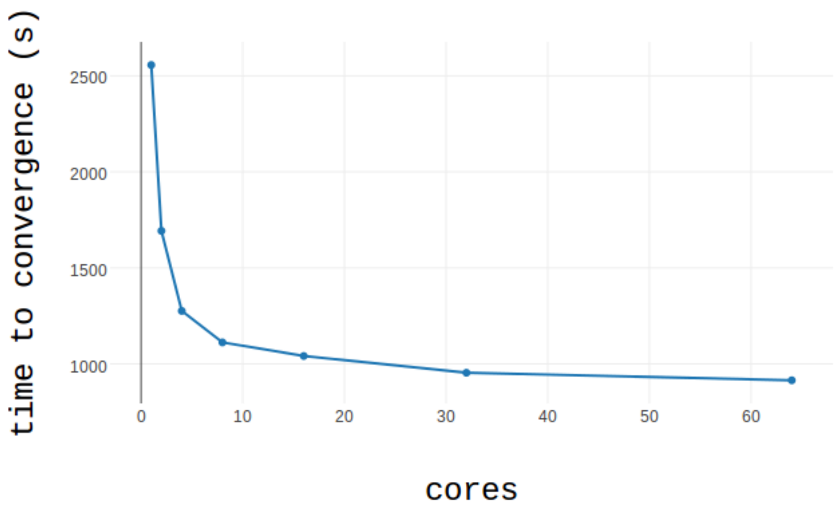

Figure 4.5: Time to convergence with respect to cores used.

4.4 Parallel Results

Using the configuration determined by previous tests, performance of parallel ADMM is evaluated. Given the 3-layer network configuration, a reconstruction error of 0.12 over the validation data is considered to have met convergence. ADMM was run for values of $N = 1, 2, 4, 8, 16, 32, 64$. A plot comparing the error with respect to time for each value of N can be seen in Figure 4.4. A plot comparing the number of cores used with respect to time until convergence can be seen in Figure 4.5. Speedup can be seen in Table 4.3.

The computation of

$$\left(\sum_{n=1}^{N} a_{l-1}^{n}(a_{l-1}^{n})^{T}\right)^{-1}, \tag{4.5}$$

can act as a bottleneck whereby in adding more nodes one may not improve the speed of learning. This bottleneck can occur because the time taken to compute this inverse is replicated for all nodes. This limits speedup when the number of inputs for each layer is greater than the number of local samples.

Table 4.3: Time to convergence (s)

N	Time	Speedup
1	2556.77	1.00
2	1692.83	1.51
4	1275.98	2.00
8	1111.96	2.30
16	1041.19	2.46
32	953.96	2.68
64	914.73	2.80

5 Looking Forward

Future direction for applying this method exists in introducing hyperparameters comparable to those in standard neural network training algorithms. One of these is the momentum term, which accelerates learning based on the previous change in weights, multiplied by a scalar constant (Sutskever et al., 2013). A more comprehensive analysis of layer-dependent values of $\{\beta_l\}$, $\{\gamma_l\}$ may also improve performance.

Evaluating this method on a dataset with more training samples would allow for greater speedup as more nodes are introduced, since the baseline time spent in computing (4.5) would be dominated by other computations that enjoy parallel speedup.

5.1 Conclusions

I have described and evaluated the use of ADMM in training neural networks. In examining this method, I determined appropriate hyperparameters for the MNIST application domain and presented experimental results that demonstrate the properties of memory use and convergence.

Acknowledgments

I would like to thank my advisor, Ting Zhu, for his guidance in this project and my academic career.

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See hpcf.umbc.edu for more information on HPCF and the projects using its resources.

References

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256.

Goldstein, T., Taylor, G., Barabin, K., and Sayre, K. (2015). Unwrapping ADMM: efficient distributed computing via transpose reduction. *CoRR*, abs/1504.02147.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.

Lions, P.-L. and Mercier, B. (1979). Splitting algorithms for the sum of two nonlinear operators. *SIAM Journal on Numerical Analysis*, 16(6):964–979.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Neurocomputing: Foundations of research. pages 696–699. MIT Press, Cambridge, MA, USA.

Sutskever, I., Martens, J., Dahl, G. E., and Hinton, G. E. (2013). On the importance of initialization and momentum in deep learning. *ICML (3)*, 28:1139–1147.

Taylor, G., Burmeister, R., Xu, Z., Singh, B., Patel, A., and Goldstein, T. (2016). Training neural networks without gradients: A scalable ADMM approach. *CoRR*, abs/1605.02026.