MagmaDNN: Accelerated Deep Learning Using MAGMA

Daniel Nichols
dnicho22@vols.utk.edu
University of Tennessee, Knoxville

Kwai Wong
kwong@utk.edu
University of Tennessee, Knoxville

Stan Tomov
tomov@icl.utk.edu
University of Tennessee, Knoxville

Lucien Ng
lokm13@gmail.com
The Chinese University of Hong Kong

Sihan Chen
sihanchen@link.cuhk.edu.hk
The Chinese University of Hong Kong

Alex Gessinger
alexgessinger1@gmail.com
Slippery Rock University

ABSTRACT

MagmaDNN [17] is a deep learning framework driven using the highly optimized MAGMA dense linear algebra package. The library offers comparable performance to other popular frameworks, such as TensorFlow, PyTorch, and Theano. C++ is used to implement the framework providing fast memory operations, direct cuda access, and compile time errors. Common neural network layers such as Fully Connected, Convolutional, Pooling, Flatten, and Dropout are included. Hyperparameter tuning is performed with a parallel grid search engine.

MagmaDNN uses several techniques to accelerate network training. For instance, convolutions are performed using the Winograd algorithm and FFTs. Other techniques include MagmaDNNs custom memory manager, which is used to reduce expensive memory transfers, and accelerated training by distributing batches across GPU nodes.

This paper provides an overview of the MagmaDNN framework and how it leverages the MAGMA library to attain speed increases. This paper also addresses how deep networks are accelerated by training in parallel and further challenges with parallelization.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; *Parallel computing methodologies.*

KEYWORDS

neural networks, deep learning, parallel computing

ACM Reference Format:
Daniel Nichols, Kwai Wong, Stan Tomov, Lucien Ng, Sihan Chen, and Alex Gessinger. 2019. MagmaDNN: Accelerated Deep Learning Using MAGMA. In *Practice and Experience in Advanced Research Computing (PEARC '19), July 28-August 1, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3332186.3333047

1 INTRODUCTION

Machine Learning is becoming an increasingly vital aspect of today's technology, yet traditional techniques prove insufficient for complex problems [9]. Thus, Deep Learning (DL) is introduced, which is a methodology for learning data with multiple levels of abstraction [15]. DL is driven by Deep Neural Networks (DNN), which are Artificial Neural Networks (ANN) comprised of multiple layers and often convolutions and recurrence. DNNs are used to model complex data in numerous fields such as autonomous driving [5], handwriting recognition [16], image classification [14], speech-to-text algorithms [11], and playing computationally difficult games [18].

Non-trivial DL tasks often require large weight matrices and therefore an abundance of training samples. This leads to long computation time and large memory footprints. Popular data-sets, such as ImageNet, contain more than 14 million training images and over 20 thousand classes[2]. Such models can take several days to weeks to train. Long computation times combined with the abundance of data in the modern era increases the need for fast, scalable DL implementations.

Modern advances in GPU technology have greatly accelerated DNN implementations and allowed them to scale with sufficient proportion [8]. Despite the success in scaling DNNs the technology is not easily accessible to researchers outside of DL and is only slowly making its way into popular frameworks such as Tensorflow.

2 MAGMADNN FRAMEWORK

The MagmaDNN framework consists of four major components: **MemoryManager** (2.1), **Tensor** (2.2), **Layer** (2.3), and **Model** (2.4). Each of these wrap around the prior and provides increasing levels of abstraction for the framework's user. Three of these, all but the MemoryManager, are used in the typical **Workflow** (2.5) of a MagmaDNN program.

2.1 MemoryManager

When accelerating computation on a GPU, handling memory can become a disconnect between researcher and results. MagmaDNN removes this obstacle with its MemoryManager, which is responsible for abstracting memory operations into a single class. It defines and can control four different memory types: *HOST*, *DEVICE*, *MANAGED*, and *CUDA_MANAGED*. The difference between the latter two being that *CUDA_MANAGED* uses CUDA's unified memory, while *MANAGED* is MagmaDNN's own implementation of host-device synchronized memory.

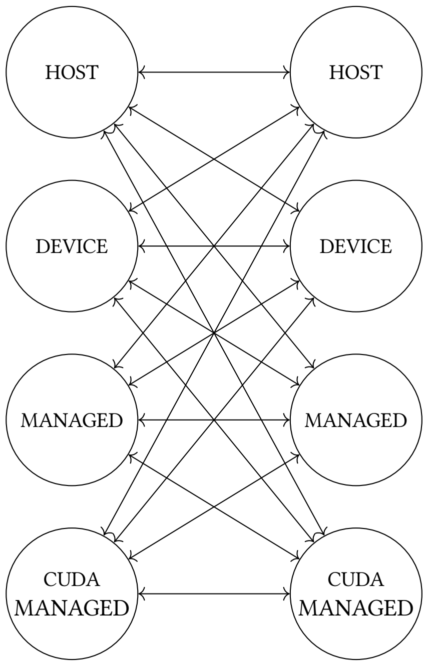

Figure 1: MemoryManager Copying Support

Memory bugs in GPU intensive code typically arise from confusion as to where data is stored and how to access it across devices. The manager keeps track of its memory's device location, which it uses to handle inter-device data communication.

Creating MemoryManagers is simple and only requires four pieces of information: *data type*, *size*, *memory type*, and *device id*.

```
MemoryManager<type> m (size, memory_type, device);
```

Type can be `float`, `double` or `magmaHalf` dictating whether to use either single, double, or half precision, respectively. *Size* provides the number of entries that the MemoryManager will store, and will require that `size * sizeof(type)` bytes be allocated. *Memory type* can be either *HOST*, *DEVICE*, *MANAGED*, or *CUDA_MANAGED*, which will determine how the data is stored and how it is used during training. The *device id* specifies which device to use for storage. For instance, a memory type of *DEVICE* and device id of 0 will store the data on GPU 0.

Copying is wrapped into the `copy_from` function, which handles all cases of MemoryManager copying (see figure 2.1).

```
m.copy_from(const MemoryManager<T>& m);
```

In addition to simplifying memory operations MagmaDNN aims to optimize them. Typically, host-device copying introduces a bottleneck, especially in the case of device $\rightarrow$ host memory operations. For this reason, MagmaDNN includes its own *MANAGED* type memory, which only synchronizes when necessary in order to minimize copying. Memory can also be prefetched asynchronously avoiding the need to wait for data to copy during network training.

2.2 Tensor

Fundamental to deep learning techniques is the tensor. Wrapped around a MemoryManager, the Tensor class provides additional representational functionality and math operations. The data structure interprets its linear memory as a multi-dimensional array. Tensors can be indexed python style, with negative indices, and support reshaping and squeezing. By using the MemoryManager, Tensors abstract the method in which their data is stored. The typical workflow (see Section 2.5) often involves computations on a single tensor on both the host and device. For this reason, Tensors use MagmaDNN's *MANAGED* type memory by default.

Like the MemoryManager, tensors are straightforward to create and only require a shape.

```
Tensor<type> t ({axis1,axis2,...});
```

Here `{axis1,axis2...}` is a vector of integers that define the size of each tensor axis. Optionally, a memory type and device id can be specified, however, they are defaulted to be *MANAGED* and 0.

Using this structure, MagmaDNN implements tensor multiplication and addition with MAGMA to accelerate the operations. MAGMA utilizes both the multi-core CPU and GPU device to accelerate these operations [20]. Other element-wise operations such as the Hadamard product and activation functions are implemented using optimized CUDA kernels.

In recent years, convolutional and recurrent networks have become the focal point of deep learning. Convolutional layers are crucial in models training on image or other spatial data. MagmaDNN uses the Winograd algorithm to compute convolutions, but also has the support to use the CuDNN [7] framework.

2.3 Layer

DNNs are comprised of layers each defined by a weight and bias tensor and some activation function. Each Layer class is capable of forward and backward propagation, updating the weights based on the values of neighboring layers. MagmaDNN implements several different layer types: *Input*, *Fully Connected*, *Activation*, *Conv2D*, *Pooling2D*, *Dropout*, and *Output*. Training routines make use of the abstract *Layer* class allowing the use of custom layer implementations.

Each layer provides various parameters. For instance, the activation layer accepts several activation functions: *sigmoid*, *relu*, and *tanh*. Each of these is supported on the host and device.

2.4 Model

Typical DNN users are not always knowledgeable of or willing to create implementations of training routines. Implementations of back-propagation and various optimizers are non-essential to applied models and can present a development bottleneck to researchers. For this reason, MagmaDNN employs the Model class.

The Model class abstracts the network and its training functionality. It creates a simple interface for creating DNN layers and running a training routine. Given a testing set and a list of layers, a Model can train a DNN with the given layers and testing data and predict any future samples using the DNN. For researchers utilizing DNNs in various applications, the Model class allows for a faster development time without concern for the implementation of network optimization.

Creating models only requires two structures: a `Param` struct, which stores model hyperparameters, and a vector of network layers.

```
Model m (params, layers);
m.fit(x_train, y_train, verbose, &out);
```

Training the networks is then a simple function call to `fit`, which uses the model parameters and *x_train/y_train* tensors to run a training routine.

2.5 Workflow

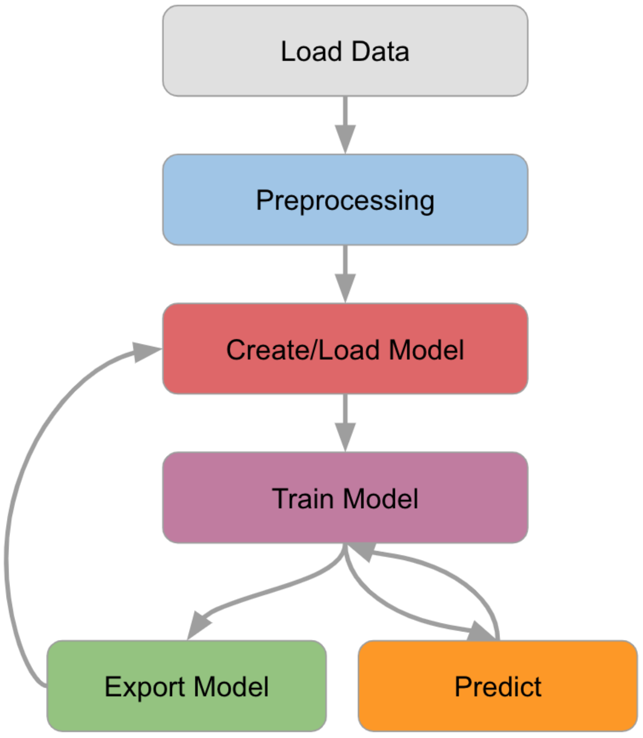

Figure 2: MagmaDNN Workflow

Like many other frameworks, MagmaDNN is built around the workflow *Load/Pre-Process Data* $\longrightarrow$ *Create Model* $\longrightarrow$ *Train* $\longrightarrow$ *Predict* $\longrightarrow$ *Train* $\longrightarrow$ $\cdots$ (see figure 2). Functionality is offered for data I/O and manipulation, however, the core is in the Model training step. Due to separating individual tasks into standalone classes, for example the Tensor class, different workflows are viable. However, MagmaDNN is optimized towards that presented in figure 2.

3 PERFORMANCE

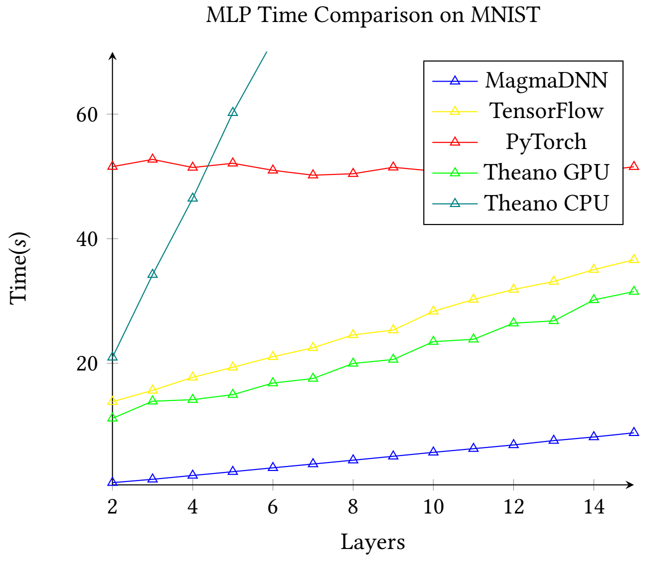

Figure 3: Time Comparisons

3.1 MNIST Test

The Modified National Institute of Standards and Technology (MNIST) data set is a collection of approximately 60 thousand images of handwritten digits along with their proper labels [1]. Each image is reduced to 28x28 pixels and anti-aliased to grayscale. Following its first use by Lecun et. al. [16], the data set has become a standard test for neural networks and other machine learning techniques.

MNIST was used to compare MagmaDNN to other popular frameworks such as Tensorflow, PyTorch, and Theano (see figure 3). All models were formed using sequential fully connected layers, each with 528 hidden units, a learning rate of $\eta = 0.05$, a weight decay of $\alpha = 0.001$, and activation function sigmoid. All libraries were tested with 100 samples per batch for 5 epochs on an Intel Xeon X5650 (2.67GHz×12) processor accompanied by an Nvidia 1050Ti GPU. The graphics card was equipped with 4 GB of memory and 768 cores. Each test was ran using the GPU, in addition to a CPU only Theano test, which is included to give an additional frame of reference to how GPUs accelerate DNN training.

MagmaDNN was the fastest in each test, finishing five epochs on the four layer test in 2.00477 seconds. On the four layer test MagmaDNN was $\approx$ 6.8 times faster than TensorFlow and $\approx$ 17.8 times faster than the Theano CPU only run.

3.2 Scaling

As shown in figure 3 MagmaDNN not only trained the fastest, but scaled at an optimal rate compared to other popular frameworks (see table 1 and figure 3). PyTorch scaled better than MagmaDNN, but performed poorly on the small MNIST model due to excessive *HOST* $\leftrightarrow$ *DEVICE* copy operations.

Framework	**Δ time / Δ layer**
MagmaDNN	0.6197
TensorFlow	1.7524
Theano (GPU)	1.5271
Theano (CPU)	12.5071
PyTorch	-0.08

Table 1: Change in Training Time with Number of Layers

4 PARALLELISM

As with many of computationally intensive tasks DNN training can be accelerated by distributed computing. Despite this fact there are several challenges to training networks in parallel.

4.1 Previous Work

Most modern DNN frameworks provide support for the use of GPU acceleration. GPUs drastically improve the training time of larger networks by allowing for faster computation of matrix operations. This large difference is evinced by the run-times in figure 3 and run-time increase rates in table 1.

Despite the accelerating capacity of GPUs they are often bottle-necked by memory transferring. To combat the bottle-neck, memory transfers are reduced by using minibatches. As suggested by the name, minibatches work by training on sets or "batches" of samples at a time rather than a single sample. However, another hyperparameter, the *batch size*, has been added to the model design process increasing the overall complexity of the problem.

Typically, increasing the batch size increases computational efficiency, while decreasing the accuracy of the model. Larger batch sizes also impede the convergence of stochastic gradient descent (SGD), when optimizing the network. In practice various tricks such as warm-up [10], gradually increasing batch size [19], or layer-wise adaptive rate scaling (LARS) [21] are required to guarantee convergence. However, in general they do not eliminate the batch size limit, but only raise it [4]. Using combinations of these techniques You et. al. were able to train the AlexNet model in 11 minutes on the ImageNet-1k data set [22].

GPUs and minibatches are currently present in most DNN workflows and provide significant speedups over their absence. However, they only apply to training in a single node. Parallelization techniques can be added on top of them to extend training to multiple nodes and improve the training time, while maintaining validation accuracy.

The most common of these techniques is *Data-Parallelism* (see figure 4). In data parallelism weights are sent from a master node to N worker nodes. Let w^j be the weights of the j-th worker node. Each node computes the gradient ∇w^j and sends it back to the master node. Once the master node has received the gradients from each worker it calculates $\overline{w} \leftarrow \overline{w} - \eta/N \sum_{j=1}^{N} \nabla w^j$, the average weight, and broadcasts $\overline{w}$ back to each worker. Blocking data parallelism is the typical method used in scaling deep learning and it has shown promising results [12][6].

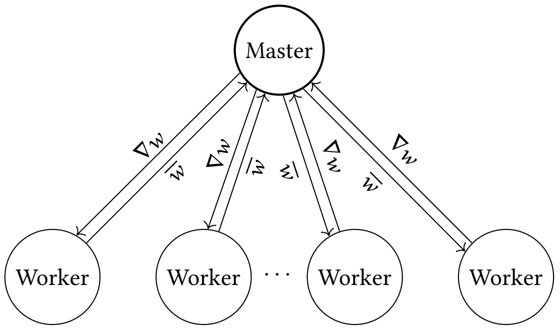

Figure 4: Data Parallelism

Despite its success data parallelism has drawbacks. Each node is required to be given a necessary amount of work such that it is not mostly idle. Additionally, a blocking call has been introduced to the parameter server creating "lulls" where processors sit idle. To deal with these issues some systems train using *Model-Parallelism*. In model parallelism network parameters are split evenly across available devices. Due to device memory constraints model parallelism does not scale well. For instance, when training using large images model parallelism forces the use of small batch sizes in order to fit the data in device memory.

Another approach is *Layer-Parallelism*. In a similar scheme to CPU pipelining, layer parallelism computes layers in parallel as data becomes available. Training is accelerated here by minimizing CPU idle time. Layer parallelism is used in practice due to its performance benefits [13][3]. However, it suffers from irregular transfer rates between processors [4].

Typically *Hybrid-Parallelism* is the most optimal way to train a network in parallel [4]. Hybrid parallelism combines various aspects of data, model, and layer parallelism in order to train. Determining how to mix methods into hybrid parallelism introduces more parameters and is model dependent. Thus, hybrid parallelism does not extend well to general training tasks.

4.2 MagmaDNN Parallelism

Accelerating fine-grained parallelism is currently not within the scope of MagmaDNN, so other frameworks are used to accelerate these areas. One instance being the use of MAGMA to implement linear algebra routines. By using MAGMA, tensor operations are computed with state-of-the-art parellelization techniques. Specifically, the highly parallel SIMD architecture on CUDA devices are utilized to accelerate matrix computations.

Course-grained parallelism is accomplished in multiple ways in the MagmaDNN framework. Models can make use of several GPUs on a single node to train larger, memory-consuming networks. Additionally, multiple GPUs can be employed to train with *Data-Parallelism* (as described in section 4.1).

Data parallelism is implemented using a combination of CUDA's asynchronous capabilities and CUDA aware MPI. The use of MPI gives MagmaDNN the capability to train across several nodes in a network. Training does not suffer greatly from the lack of fault-tolerance in popular MPI implementations due to the large amount of samples typically trained on.

4.3 Hyperparameter Tuning

Data parallelism succeeds in not only optimizing weights, but also model hyperparameters. MagmaDNN uses a Random and/or Exhaustive Grid Search technique to optimize hyperparameters. Those currently being optimized are learning rate, weight decay rate, batch size, and number of training epochs. However, the routine is modular and able to add new dimensions to the search space.

In grid search, as with data parallelism, a parameter server sends a parameter set to each node, where the model is trained according to its received parameters. The parameter server, or master, in turn receives the training time, accuracy, and loss associate with each parameter set. Using some objective function, typically a combination of training duration and accuracy, the optimization routine gives the optimal training parameters.

Grid search can be run to exhaustively search a range of parameters (with a given step size), however, this is often too large to be feasible. For these reasons grid search can be ran using random sampling until some accuracy threshold is met.

5 DISCUSSION AND FUTURE OF MAGMADNN

Parallel hardware continues to increase in capability as the field of DL continues to grow. Even as exaflop computing approaches on the horizon DNNs fail to utilize the entirety of this immense computing power. Due to the complexity of modern DL tasks it is becoming increasingly essential to fully exploit parallelism in DNN training.

Given MagmaDNN's fast training and C++ interface it can serve as a valuable tool in the DL atmosphere. Being a young framework it lacks many features that current mature frameworks possess. The project aims to provide a modular framework for researchers to rapidly implement DNNs and train them quickly. Beyond a simple interface, MagmaDNN will provide techniques for distributed training, half-precision, and hyperparameter optimization.

Currently MagmaDNN is structured as a standard C++ project with a `Make` build system. The code has been tested on Ubuntu 16 and greater and Mac OS, but is likely to run on any *nix style operating system with proper CUDA driver support. MagmaDNN aims to provide more tests and examples on distributed systems going forward.

6 AVAILABILITY

MagmaDNN is currently developed and supported by the Innovative Computing Laboratory (ICL) and Joint Institute for Computer Science (JICS) at the University of Tennessee, Knoxville and Oak Ridge National Laboratory. Source code, documentation, tutorials, and licensing can all be found on the project's homepage[1].

ACKNOWLEDGMENTS

This work was conducted at the Joint Institute for Computational Sciences (JICS), sponsored by the National Science Foundation (NSF), through NSF REU Award #1659502, with additional Support from the University of Tennessee, Knoxville (UTK), and the National Institute for Computational Sciences (NICS). This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Computational Resources are available through a XSEDE education allocation award TG-ASC170031. In addition, the computing work was also performed on technical workstations donated by the BP High Performance Computing Team.

[1]https://bitbucket.org/icl/magmadnn/

REFERENCES

[1] 1998. MNIST. http://yann.lecun.com/exdb/mnist/. http://yann.lecun.com/exdb/mnist/ [Online; accessed 11-March-2019].

[2] 2011. ImageNet. http://image-net.org/index. http://image-net.org/index [Online; accessed 11-March-2019].

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). arXiv:1603.04467 http://arxiv.org/abs/1603.04467

[4] Tal Ben-Nun and Torsten Hoefler. 2018. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *CoRR* abs/1802.09941 (2018). arXiv:1802.09941 http://arxiv.org/abs/1802.09941

[5] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, and et al. 2016. End to End Learning for Self-Driving Cars. *arXiv:1604.07316 [cs]* (Apr 2016). http://arxiv.org/abs/1604.07316 arXiv: 1604.07316.

[6] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. 2016. Revisiting Distributed Synchronous SGD. *CoRR* abs/1604.00981 (2016). arXiv:1604.00981 http://arxiv.org/abs/1604.00981

[7] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[8] Adam Coates, Brody Huval, Tao Wang, David J Wu, Andrew Y Ng, and Bryan Catanzaro. [n. d.]. Deep learning with COTS HPC systems. ([n. d.]), 9.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[10] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR* abs/1706.02677 (2017). arXiv:1706.02677 http://arxiv.org/abs/1706.02677

[11] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and et al. 2012. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine* 29, 6 (Nov 2012), 82âĂŞ97. https://doi.org/10.1109/MSP.2012.2205597

[12] Forrest N. Iandola, Khalid Ashraf, Matthew W. Moskewicz, and Kurt Keutzer. 2015. FireCaffe: near-linear acceleration of deep neural network training on compute clusters. *CoRR* abs/1511.00175 (2015). arXiv:1511.00175 http://arxiv.org/abs/1511.00175

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR* abs/1408.5093 (2014). arXiv:1408.5093 http://arxiv.org/abs/1408.5093

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. *ImageNet Classification with Deep Convolutional Neural Networks*. Curran Associates, Inc., 1097âĂŞ1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521 (May 2015), 436. Citation Key: lecunDeepLearning2015a.

[16] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov 1998), 2278–2324. https://doi.org/10.1109/5.726791

[17] MagmaDNN 2019. MagmaDNN Site. https://bitbucket.org/icl/magmadnn.

[18] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, and et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (Jan 2016), 484âĂŞ489. https://doi.org/10.1038/nature16961

[19] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. 2017. Don't Decay the Learning Rate, Increase the Batch Size. *CoRR* abs/1711.00489 (2017).

arXiv:1711.00489 http://arxiv.org/abs/1711.00489

[20] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Comput.* 36, 5-6 (June 2010), 232–240. https://doi.org/10.1016/j.parco.2009.12.005

[21] Yang You, Igor Gitman, and Boris Ginsburg. 2017. Scaling SGD Batch Size to 32K for ImageNet Training. *CoRR* abs/1708.03888 (2017). arXiv:1708.03888 http://arxiv.org/abs/1708.03888

[22] Yang You, Zhao Zhang, Cho-Jui Hsieh, and James Demmel. 2017. 100-epoch ImageNet Training with AlexNet in 24 Minutes. *CoRR* abs/1709.05011 (2017). arXiv:1709.05011 http://arxiv.org/abs/1709.05011