MagmaDNN: Towards High-Performance Deep Learning Using Magma

Nichols, Daniel
dnicho22@vols.utk.edu

Keh, Sedrick
sedrickkeh@gmail.com

Chan, Kam Fai
kfchan1998@gmail.com

July 2019

Abstract

Deep Learning (DL) has become an increasingly attractive tool with the advent of GPUs and distributed computing. Models that would have taken years to train can now be trained in several hours or less. Progress in training techniques has led to the introduction of larger and more complicated models, which only increases the resources and tools necessary to train. Most DL solutions aim to provide a modular and user-friendly interface, typically in Python, often at the cost of performance. In this paper we discuss the development of MagmaDNN: a framework aimed at providing a simple interface with performance benefits from the use of Magma and c++.

Contents

1 Introduction **2**
- 1.1 Deep Learning Foundations . . . 2
  - 1.1.1 Mathematical Foundations . . . 2
  - 1.1.2 Programming Foundations . . . 4

2 MagmaDNN **6**
- 2.1 Related Work . . . 6
- 2.2 Design Principles . . . 7
- 2.3 MagmaDNN v1.0 Update . . . 7
- 2.4 Interface . . . 7
- 2.5 Modularity . . . 8

3 Profiling **9**
- 3.1 Fine-Grain . . . 9
- 3.2 Course-Grain . . . 9
  - 3.2.1 MNIST and CIFAR . . . 9

3.2.2 Comparisons . . . 10
3.3 Compute Graph Optimization . . . 10

4 Distributed Training **11**
4.1 Techniques . . . 11
4.1.1 Data Parallelism . . . 11
4.1.2 Model Parallelism . . . 11
4.1.3 Hybrid Parallelism . . . 14
4.2 Implementations . . . 14
4.2.1 Parameter Server . . . 14
4.2.2 AllReduce . . . 14
4.2.3 Ring-Reduce . . . 14

5 Applications **14**
5.1 Hyper-Parameters . . . 14

6 Conclusions **16**

7 Future Work **16**

8 Availability **16**

Acknowledgement **17**

1 Introduction

1.1 Deep Learning Foundations

Neural Networks are a parametric learning technique. The networks are comprised of *layers*, which each store a vector of weights and define some transformation function. Forward propagation (see Algorithm 1) is used to predict classes and back-propagation (see Algorithm 2) is used to calculate gradients of the weight vectors at each layer based on error. In section 1.1.1 a more formal approach is presented.

1.1.1 Mathematical Foundations

Let $\mathcal{D}$ be our training data set with each sample having dimension d and n output classes. Let $f : \mathbb{R}^d \to \mathbb{R}^n$, given by $f(\boldsymbol{x}; \boldsymbol{w})$, be some objective function that predicts the output class of $\boldsymbol{x} \in \mathcal{D}$ parameterized by $\boldsymbol{w}$. f is the *network* and represents some manifold $\mathcal{M}$. Now consider some loss function $\mathcal{L} : \mathbb{R}^n \times \mathcal{D} \to \mathbb{R}$, which takes the predicted class and correct class and returns the error in prediction.

The task in training comes down to minimizing the expected error.

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \; \mathbb{E}_{\boldsymbol{x}\sim\mathcal{D}} \left[\mathcal{L} \left(f^*(\boldsymbol{x}), f(\boldsymbol{x}; \boldsymbol{w}) \right) \right]. \tag{1}$$

Two problems arise from this statement. First, the choice of f. Second, the means in minimizing the loss. f is distinctive in the case of neural networks. Since networks aim to learn highly non-linear, abstracted data, then we use a tetration of activation functions. For example,

$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x}; \boldsymbol{w}^{(1)}); \boldsymbol{w}^{(2)}); \boldsymbol{w}^{(3)}).$$

This is an example of a 3 layer neural network. However, in practice some arbitrary number of $f^{(i)}(\cdot;\cdot)$ are used. Each $f^{(i)}$ semantically corresponds to a *layer* of the network.

To avoid the use of bias addition assume that $\boldsymbol{w}, \boldsymbol{x} \in \mathbb{R}^{d+1}$, such that the last value $\boldsymbol{x}_{d+1} = 1$. This correctly incorporates the bias into the transformation $\boldsymbol{w}^T\boldsymbol{x}$.

A typical choice for $f^{(i)}$ is $g(\boldsymbol{w}^T\boldsymbol{x})$, where g is called the *activation function*. This is also called a *dense layer*. After transforming, the values are typically scaled into the range $[-1, 1]$ or $[0, \infty)$ using an *activation function* g. A standard choice of activation function g is $\sigma(x) = \frac{1}{1+\exp(-x)}$. Other popular choices include softmax, RELU, tanh, and RBF.

$$\text{softmax}(\boldsymbol{x}_i) = \frac{\exp(\boldsymbol{x}^T\boldsymbol{w}_i)}{\sum_k \exp(\boldsymbol{x}^T\boldsymbol{w}_k)} \tag{2}$$

$$\text{RELU}(\boldsymbol{x}_i) = \begin{cases} \boldsymbol{x}_i, & \boldsymbol{x}_i > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

$$\tanh(\boldsymbol{x}_i) = \frac{\exp(2\boldsymbol{x}_i) - 1}{\exp(2\boldsymbol{x}_i) + 1} \tag{4}$$

$$\text{RBF}_{\text{gaussian}}(\boldsymbol{x}_i) = \exp(-(\varepsilon\boldsymbol{x}_i)^2) \tag{5}$$

Algorithm 1: Forward-Propagation. Here $\lambda\Omega(\boldsymbol{w})$ is included as a regularization term.

Result: Forward-Propagation
$\boldsymbol{h}^{(0)} \leftarrow \boldsymbol{x}$ where $\boldsymbol{x} \in \mathcal{D}$

foreach *layer* $\in 1 \ldots \ell$ **do**
  $\boldsymbol{a} \leftarrow \left(\boldsymbol{w}^{(layer)}\right)^T \boldsymbol{h}^{(layer-1)}$
  $\boldsymbol{h}^{(layer)} \leftarrow f^{(layer)}(\boldsymbol{a})$
end
$\hat{\boldsymbol{y}} \leftarrow \boldsymbol{h}^{(\ell)}$
error $\leftarrow \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \lambda\Omega(\boldsymbol{w})$

Now that f is correctly defined it must be minimized. Since f is non-convex we cannot be guaranteed to find a global minima, however, typical convex optimization techniques will find local minima. The most common minimization algorithm used in NN training is gradient descent or one of its many adaptations. Let $\boldsymbol{y}, \hat{\boldsymbol{y}}$ be the actual and predicted value from our network, respectively. Gradient descent gives us an update rule for $\boldsymbol{w}$.

$$\boldsymbol{w}_j := \boldsymbol{w}_j - \eta\nabla_{\boldsymbol{w}_j}\mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}}). \tag{6}$$

Where η is the learning rate. By continually repeating this update rule for each weight $\boldsymbol{w}_j$ in the network, the weights will approach a local minima, $\boldsymbol{w}_j^*$, assuming η is within

an appropriate range and the optimization remains numerically stable. To update each individual $\boldsymbol{w}_j$, since each layer has its own weight vector, a special algorithm, called back-propagation is used (see Algorithm 2). Back-propagation makes use of computing gradients backwards to get an appropriate gradient for each internal function of f.

Algorithm 2: Back-Propagation. Here $\lambda\Omega(\boldsymbol{w})$ is included as a regularization term.

Result: Back-Propagation
First calculate $f^{(i)}(\boldsymbol{x};\boldsymbol{w})\,\forall i$ and some $\boldsymbol{x} \in \mathcal{D}$ using Algorithm 1.
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} \mathcal{L}(\boldsymbol{y}, \hat{\boldsymbol{y}})$
foreach *layer* $\in \ell \dots 1$ **do**
  $\boldsymbol{a} \leftarrow f(\boldsymbol{x}; \boldsymbol{w}^{(layer)})$
  $\boldsymbol{g} \leftarrow \boldsymbol{g} \odot f^{(layer)'}(\boldsymbol{a})$
  $\nabla_{\boldsymbol{w}^{(layer)}} J \leftarrow \boldsymbol{g} \left(\boldsymbol{h}^{(layer-1)}\right)^T + \lambda \nabla_{\boldsymbol{w}^{(layer)}} \Omega(\boldsymbol{w})$
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(layer-1)}} J = \boldsymbol{w}^T \boldsymbol{g}$
end

1.1.2 Programming Foundations

Designing and implementing DL frameworks is a non-trivial task. Often providing a flexible, clean interface comes at the cost of computation speed and vice versa. Python has been a natural choice for data science applications, while linking with fast c/c++ frameworks for the backend. This hybrid attempt at providing a dynamic interface with great efficiency comes at the cost of convoluted code bodies in several languages and complicated build processes.

DL, at its core, is an implementation of Algorithms 1 and 2 to solve the optimization problem in Equation 1. This often involves optimization techniques such as SGD (Equation 6) or Adam.

A crucial data structure in deep learning libraries is the *compute graph* (see figure 1). Mathematical functions are represented as graphs, similar to Abstract Syntax Trees in compilers, where each node is some generic operation. Graphs are a natural data structure choice due to the dependence hierarchy in mathematical expressions.

DL offers many benefits towards the use of a compute graph. For instance, most DL workflows use lazy execution. The compute graph is formed for the specific model and then it is executed during the training phase. By waiting to execute the graph, statistics can be obtained on the entire structure. More importantly, the graph can be optimized before the expensive computation. For more examples of compute graph optimization see section 3.3.

Compute graph optimization assists in reducing training time. Other techniques help reduce development time and/or complexity. One of these is *automatic differentiation*, which is a common technique for computing derivatives in software. In MagmaDNN the back-propagation algorithm uses a modified reverse-mode automatic differentiation. Numeric differentiation is too slow for deep learning applications and symbolic differentiation comes with a large development overhead. Automatic differentiation is a technique that makes use of the chain rule to compute derivatives or, in this case, gradients. Consider the function:

$$y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3 \tag{7}$$

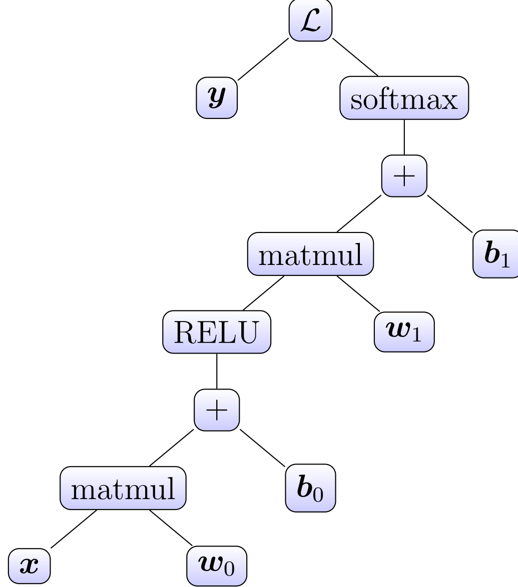

Figure 1: Computational Graph Example: A Simple 2-Layer Dense Network

Given a fixed weight w_j we would like to compute $\frac{\partial y}{\partial w_j}$. By defining the following recursive rule:

$$\frac{\partial y}{\partial w_j} = \frac{\partial y}{\partial w_{j+1}} \frac{\partial w_{j+1}}{\partial w_j} \text{ with } w_0 = x \tag{8}$$

This rule computes the gradients from the outside function inwards or, in the case of our compute graph, from the top node to w_j. Contrary to forward-mode automatic differentiation, reverse-mode passes over each function just once. The backpropagation algorithm (see Algorithm 2) is a modified version of reverse-mode differentiation.

Another serious computational consideration in DL is numerical stability. Modern DL workflows typically make use of 32 and 16-bit floating point precision for training (add citation). This low precision works well for fast training, but introduces problems with roundoff and underflow, which are exacerbated by the vanishing gradient issue in neural networks.

For example, log cross entropy, a typical loss function in neural networks, takes special care to avoid underflow and NaN values. The loss function is computed as

$$\text{cross entropy}(\boldsymbol{x}) = -\frac{1}{N} \sum_i \boldsymbol{y}_i \log \hat{\boldsymbol{y}}_i \,, \tag{9}$$

where $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are the ground truth and predicted values for $\boldsymbol{x}$. Cross entropy loss is based on the assumption that $\boldsymbol{y}$ and $\hat{\boldsymbol{y}}$ are probabilistic. Thus, the output of the network, $\hat{\boldsymbol{y}}$, is typically the softmax function (see equation 2). Now $\hat{\boldsymbol{y}} \in [0,1]^n$ and as our network gets more accurate these values will tend towards 0 and 1. Using a trained network an incorrect classification of class i will produce cross entropy $-\frac{1}{N} \log \hat{\boldsymbol{y}}_i$, where $\hat{\boldsymbol{y}}_i \approx 0^+$. Training examples like these cause numerical instability as 16 and 32-bit log functions will return

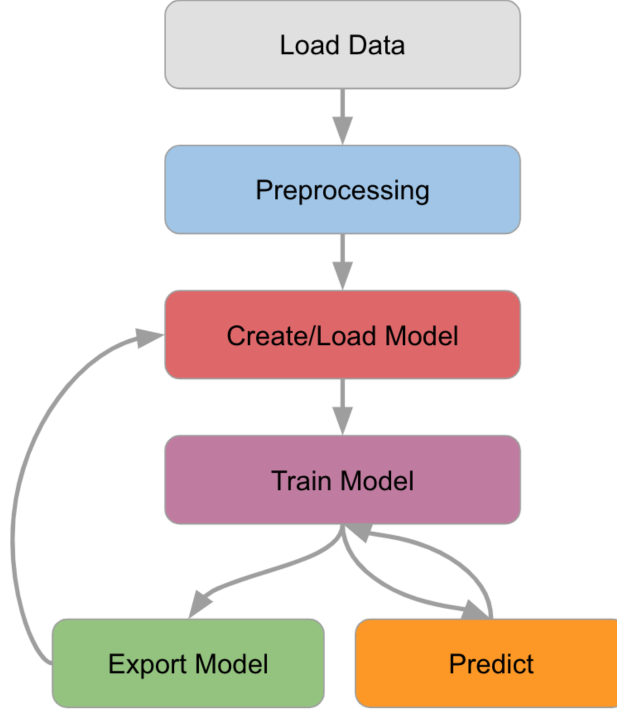

Figure 2: MagmaDNN Workflow

`-Inf` for small input close to zero. To alleviate this we define a small $\epsilon > 0$ and calculate the cross entropy as

$$\text{cross entropy}_\epsilon(\boldsymbol{x}) = -\frac{1}{N}\sum_i \boldsymbol{y}_i \log\left(\hat{\boldsymbol{y}}_i + \epsilon\right). \tag{10}$$

2 MagmaDNN

MagmaDNN is an DL framework implementation with the goal to provide out of the box high performance training on heterogeneous multicore and distributed architectures. It is built around the Magma [1] library, which powers its linear algebra routines on the GPU. For some DNN specific GPU routines, such as convolutions, CuDNN [2] is utilized to accelerate computation.

2.1 Related Work

The past several years has seen the release of numerous machine learning frameworks with strong DL support such as Tensorflow [3], PyTorch, Theano, Caffe [4], and MxNet. Each framework boasts unique design elements, speedups, and interfaces.

2.2 Design Principles

2.3 MagmaDNN v1.0 Update

MagmaDNN was created in 2017 initially to showcase the speed of Magma and viability for use in high performance machine learning workloads. Features were slowly added, but it was apparent that the initial code base could not scale into a mature deep learning framework. For this reason, MagmaDNN v1.0 was released, which is a complete rewrite of the old v0.2. It includes CPU only functionality, dynamic memorymanagers, a compute graph tensor core, and several other features.

Along with v1.0 came the functionality to compile and run MagmaDNN code on a CPU only machine. The intention here was to provide a simple testing ground for writing MagmaDNN code before they're deployed on GPU enabled machines.

2.4 Interface

Without cost in performance MagmaDNN aims to implement a modular, modern c++ interface for deep learning. C++ code, while faster and more explicit than Python, often exposes data scientists to a more difficult development environment. For instance, Python abstracts any semblance of memory management. In c++ it is near impossible to hide the use of references and pointers; especially when resources are being managed. To alleviate this programming burden MagmaDNN bundles most memory management operations into well-defined, modular classes. Release (v1.2 – unreleased at the time of writing) will fully utilize STL's smart pointers.

Tensors, the core of the DL workflow, are simple to create and dynamic in their interpretation of their underlying memory. Constructing Tensors only requires a shape, however, an optional initializer, memory type, and device id can be specified.

```
::magmadnn::Tensor<T> (
    const std::vector<unsigned int> shape&, /* tensor shape */
    const ::magmadnn::tensor_filler_t<T>& initializer,  /* (optional) */
    ::magmadnn::memory_t mem_type,  /* memory type (optional) */
    ::magmadnn::device_t device_id  /* device id (optional) */
)
```

And for a simple example:

```
/* construct a (100, 32) shaped tensor filled with zeroes */
Tensor<float> x ({100, 32}, {ZERO,{}});
```

Beyond Tensors there are many other library components with similarly simple constructions. At the core of DL is the neural network and training. Even with powerful tensor and

compute graph tools/interfaces it is difficult to write a training routine without DL knowledge. Non-DL researchers should not have to bother with the nuances of training loops. MagmaDNN solves this by offering the `Model` class which puts together every component of the library into a clean training interface.

`NeuralNetwork` is a sublass of `Model`, which specifically outlines the workflow of training a neural network.

```
::magmadnn::model::NeuralNetwork<T> (
    /* network layers */
    const std::vector::layers<::magmadnn::layer::Layer<T>*>& layers,
    ::magmadnn::optimizer::loss_t loss,    /* loss function */
    ::magmadnn::optimizer::optimizer_t optim, /* optimizer */
    ::magmadnn::model::nn_params_t params /* network hyperparameters */
)
```

`NeuralNetwork` in turn implements the `fit` function, which will train our network for us. An example might look like below.

```
/* create a neural network model */
model::NeuralNetwork<float> model (layers, optimizer::CROSS_ENTROPY,
                                   optimizer::SGD, params);

model::metrics_t metrics_out;
/* train the neural network */
model.fit(x_data, y_data, metrics_out, is_verbose);
```

Assuming the layers and data set (`x_data` and `y_data`) have been created appropriately this simple interface will train the neural network.

2.5 Modularity

A large issue with the MagmaDNN v0.2 framework was its lack of modularity. Custom workflows, optimizers, and loss functions could not be interchanged for the default behavior encoded in the package. For this reason MagmaDNN v1.0 provides a much more modular framework.

For example, the Neural Network class in MagmaDNN uses the `Optimizer` class to minimize the network. `Optimizer<T>` is an abstract class, which allows users of the framework to define a custom optimizer. Creating custom optimizers is as simple as defining the class and implementing its `update` function.

3 Profiling

Since speed is the main focus of MagmaDNN's development it is essential to optimize the framework at each development step. MAGMA optimizes all linear algebra routines, while CuDNN provides optimal performance for several DNN GPU based tasks. However, MagmaDNN still presents many optimization decisions. The following section will discuss several techniques and practices used in the library's optimization.

3.1 Fine-Grain

Most of MagmaDNN's fine grained optimization is dealt with by linear algebra packages and CuDNN. However, the choice of which package and function to use is essential to optimized training speed.

MagmaDNN's tensor reductions are an essential part of the library as they are used in most reductions. However, tensor reductions can be slow if too small a tensor is being reduced on the GPU. MagmaDNN uses several tricks to speed up redutions all without implementing its own reduction code.

By default MagmaDNN uses CuDNN for GPU tensor reductions. CuDNN, however, does not have optimal performance for small and low-dimensional tensors. Here we explore the case of reducing a matrix. For instance, sum-reducing a 2-dimensional tensor (i.e. a matrix) into a vector is trivial using matrix vector products.

$$\textit{row sum: } \boldsymbol{X}\mathbf{1}$$
$$\textit{col sum: } \boldsymbol{X}^T\mathbf{1}$$

By using the highly optimized `gemv` routines in Magma and CuBlas we can replace the call to `cudnnReduceTensor` with gemv calls. Only more optimization questions are raised by this trick. Is Magma or CuBlas faster? Are smaller reductions faster on the CPU or GPU? By running fine-grained tests (see figure 3) we were able to find that `cuBlas::gemv` performed the best for small reductions.

Magma's `gemv` seemed to perform the worst. Its performance increased for larger matrix sizes, however here we only observed small matrix vector products.

Given the above data MagmaDNN uses `cuBlas::gemv` for all its 2-dimensional tensor reductions and CuDNN for any other reductions.

3.2 Course-Grain

Techniques such as data parallelism and distributed training with CUDA-aware MPI are included in MagmaDNN to employ course-grained parallelism.

3.2.1 MNIST and CIFAR

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 [5]. Each image is reduced to 28x28 pixels and anti-aliased to grayscale. Following its

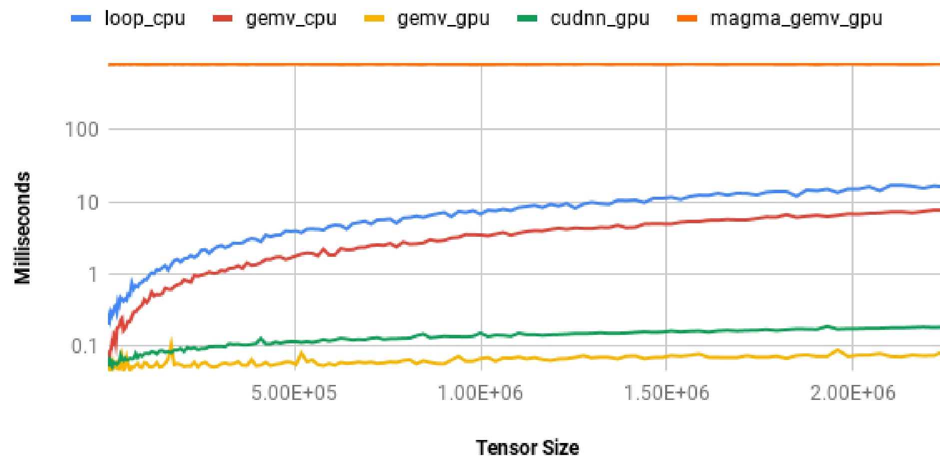

Figure 3: 2-Dimensional Tensor Reduction Comparisons

first use by Lecun et. al. [6], 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. 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$\times$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.

3.2.2 Comparisons

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 ≈ 6.8 times faster than TensorFlow and ≈ 17.8 times faster than the Theano CPU only run.

3.3 Compute Graph Optimization

Once the compute graph has been constructed, we can optimize it by reducing passes over the data and swapping out expensive routines for equal/similar inexpensive ones.

One example of the latter is swapping single precision matrix multiplication with half precision matrix multiplication (see figure 5). By utilizing the tensor cores in newer V100 cards we can see 1.2-2x speedup in training times.

Another optimization that can be performed is operation fusion (see figure 6). Here operations that can be performed in a single pass over the data are fused before being executed.

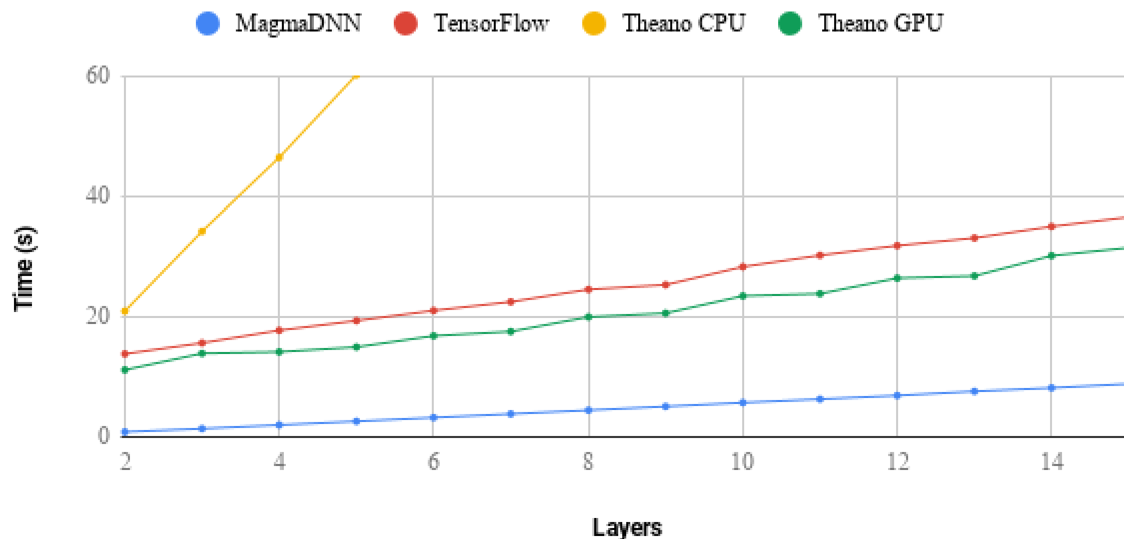

Figure 4: Training Time Comparisons

This minimizes the number of instructions during execution and speeds up the training process. In figure 6 the graph could actually be executed in a single pass, however, MagmaDNN does not change `gemm` calls as they are highly optimized and in separate dependencies.

Finally, the best approach to compute graph optimization is to use some combination of the previous mentioned techniques.

4 Distributed Training

4.1 Techniques

4.1.1 Data Parallelism

In data parallelism the same model is copied across nodes and each model is trained with separate data in parallel. At the end of each iteration, gradients are averaged across each node and training continues (see figure 8).

Data parallelism scales well as there is very little communication. However, in trying to fully utilize each node with large mini-batches, models converge much slower.

4.1.2 Model Parallelism

In model parallelism different chunks of the model are copied across nodes and training is done in a pipelined fashion (see figure 8).

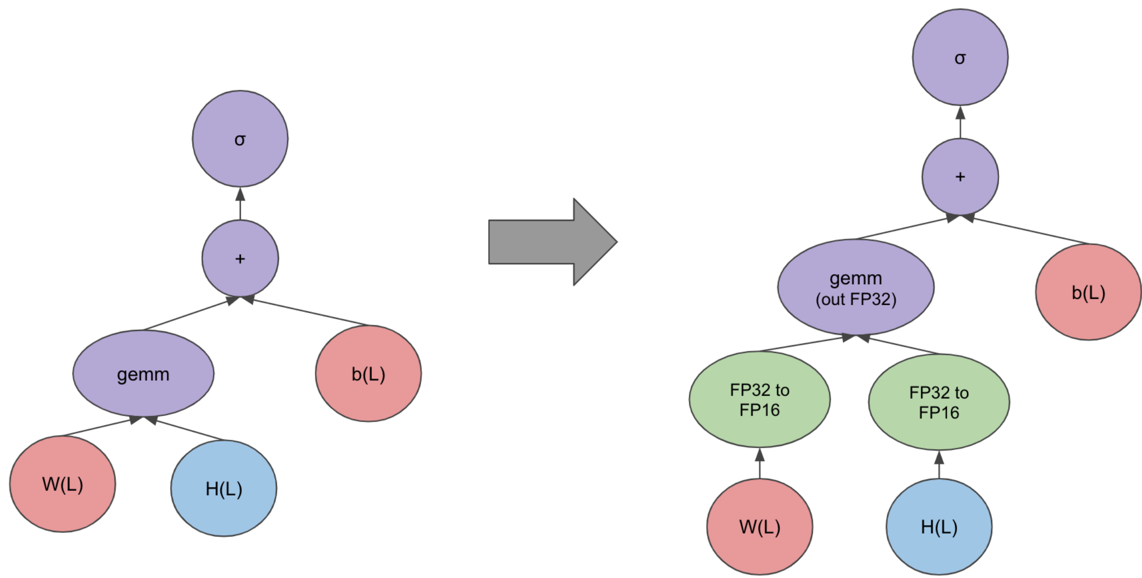

Figure 5: Mixed Precision Graph optimization

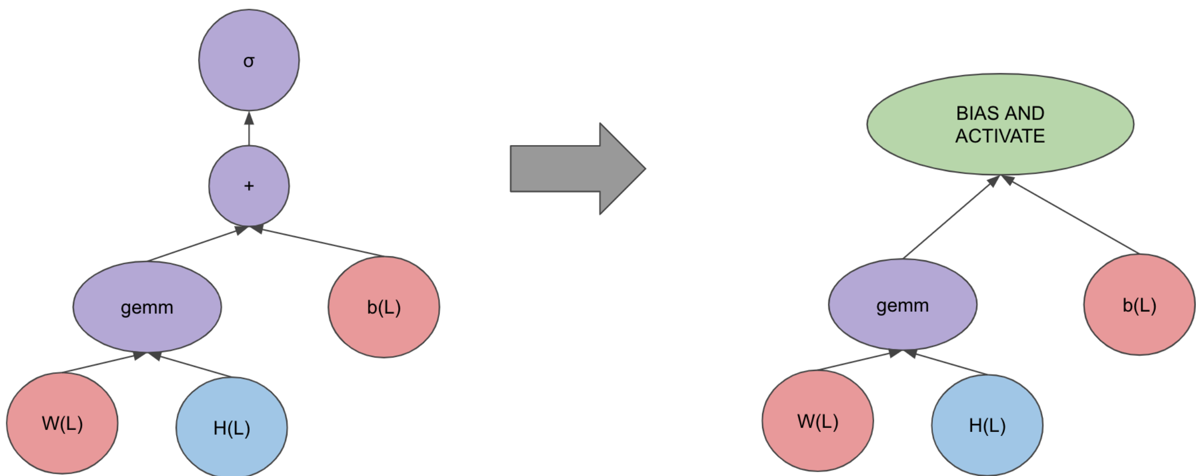

Figure 6: Fused Operation Compute Graph Optimization

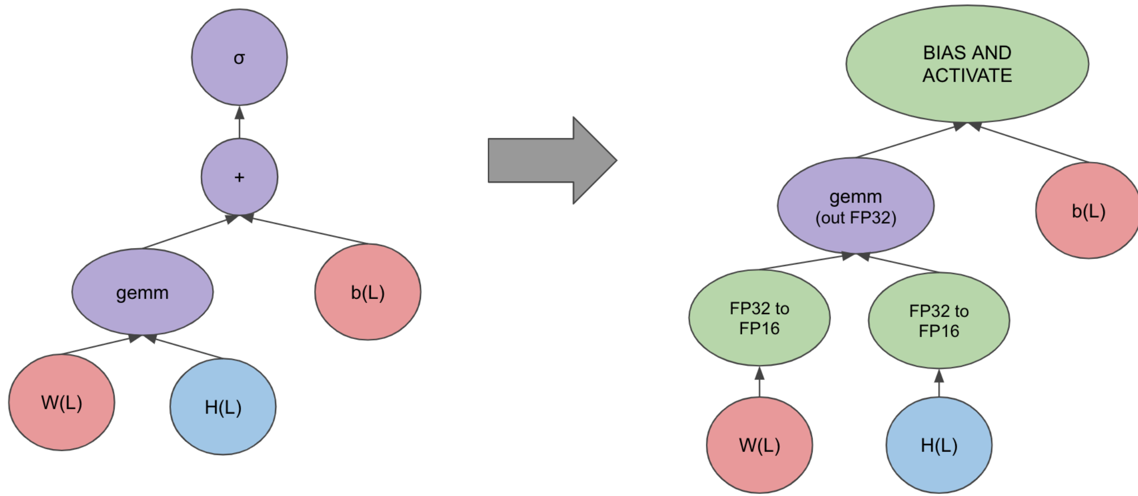

Figure 7: Hybrid Compute Graph Optimization

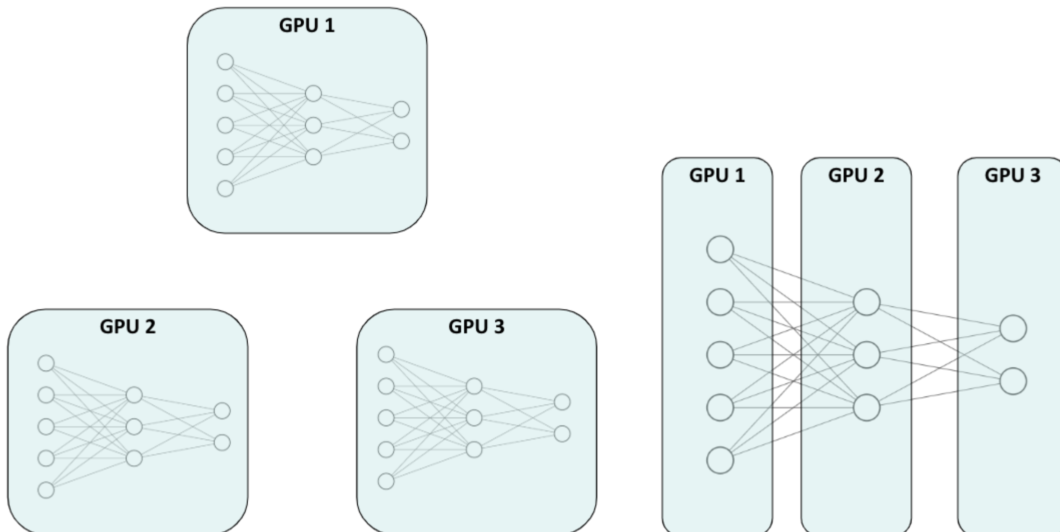

Figure 8: Data Parallelism (left) and Model Parallelism (right)

4.1.3 Hybrid Parallelism

As each distribution strategy offers unique solutions and drawbacks the best strategy is *Hybrid-Parallelism*, which combines each of the previous in some custom manner to exploit the parallel nature of a specific model. However, this makes hybrid parallelism model specific and non generalizable.

4.2 Implementations

4.2.1 Parameter Server

In this model weights are sent from a master node to N worker nodes (see figure 9). 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.

4.2.2 AllReduce

This method gets its name from the `MPI_Allreduce` call used to implement the synchronization. The Allreduce method is fairly straightforward and incredibly parallelizable. In fact allreduce gets around 90% scaling on most training problems.

Here the entire model is copied to each node/GPU. While training on each GPU, after the backpropagation, gradients are averaged across all nodes. This is easily implemented using `MPI_Allreduce` or `ncclAllReduce`.

4.2.3 Ring-Reduce

In the Ring-Reduce paradigm, we pass gradients around in a circle, similar to a token, and average them as we go along. This can also be extended to a toroidal communication pattern. Ring-Reduce acheives better performance than a simple AllReduce, but by a small margin and is much harder to implement.

5 Applications

5.1 Hyper-Parameters

Hyperparameters are the non-trained parameters in DL such as learning rate, batch size, epochs, momentum, *etc.* Finding the optimal hyperparameters is another optimization problem and far from simple. Typically, models use hand tested hyperparameters to find optimal values. However, there are many algorithms designed to address the hyperparameter problem. Both *GridSearch* and *RandomSearch* are search algorithms that aim to cover the parameter space.

GridSearch is an exhaustive technique that searches over an entire parameter space and trains the model at each point. The *optimal* model is chosen based on some training metric (i.e. accuracy, time, loss, *etc.*).

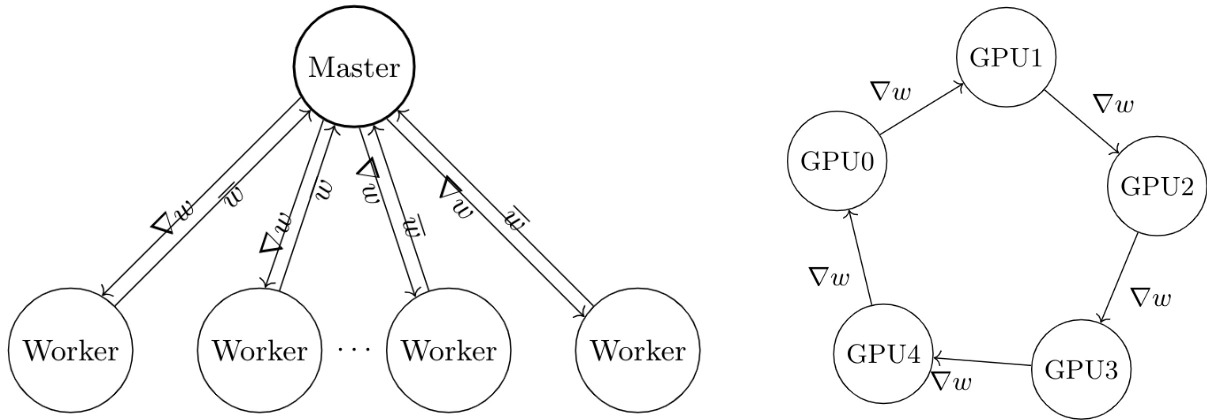

Figure 9: Master-Worker Reduce (Left) and Ring AllReduce (Right).

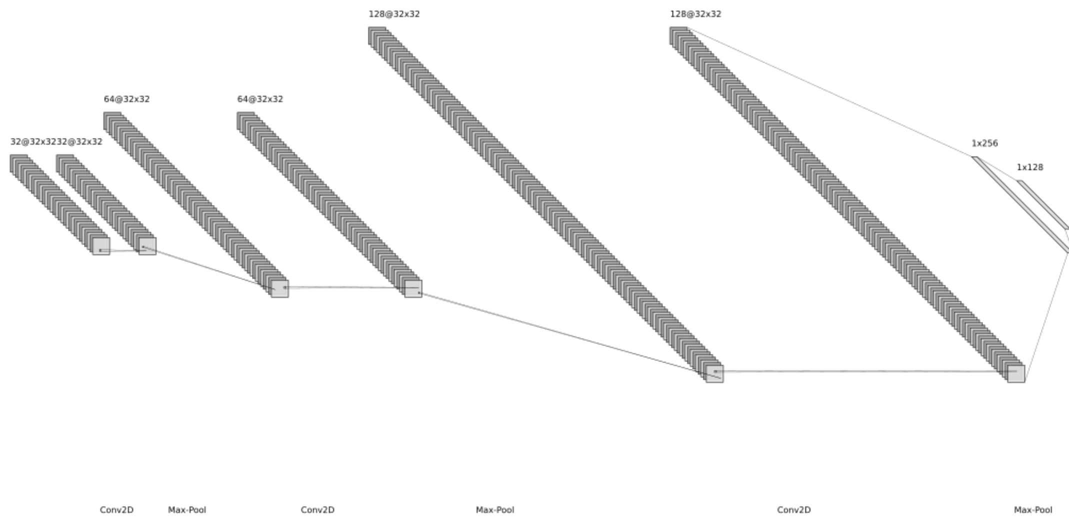

Figure 10: Convolutional Architecture

RandomSearch is similar to grid search, however, it chooses parameters randomly according to some distribution p. Smarter implementations update p as more information is known about the metric's distribution w.r.t. p.

Both *GridSearch* and *RandomSearch* often find optimal parameters, however, they come at great computational cost. The computational burden can be alleviate by using training tricks.

One such trick is to distribute the search using *data parallelism* (see section 4.1.1). By sending each sample to a separate worker node, which will train and report metrics, the computation can be distributed to reduce the total search time. However, this does not reduce the computation required and adds the complexity of distributed programming. MagmaDNN simplifies the difficulties of the distributed programming aspect, by running the model in parallel using OpenDIEL (add citation). OpenDIEL is a workflow package which provides a modular framework for designing and running parallel workflows.

Another technique is to implement early stopping. If a model is performing poorly, then it can be ignored training stopped early. This allows the search algorithms to not train in entirety. When combined with *data parallelism* and smart *RandomSearch*, this approach often offers the best method for determining hyperparameters.

6 Conclusions

MagmaDNN has been successful in its task to provide performance benefits over other frameworks and to give a c++ interface. It can connect well with other scientific codes due to the c++ implementation. There are still many additions needed going forward (see section 7), however, as is MagmaDNN performs competitively with other major packages.

7 Future Work

MagmaDNN has 2 main development goals: provide a high-performance framework for distributed DL and maintaining a usable interface. By MagmaDNN v2.0 we aim to push forward in each of these categories with two major tasks. MagmaDNN should have a competitive ResNet50 training on large scale systems, such as Summit. By v2.0 it will also fully utilize a modern c++ interface and c++2x STL features.

A more competitive feature set is also planned for the package. A downfall of MagmaDNN is its lack of features when compared to larger packages. This is in part due to the small number of data science oriented c++ packages compared to Python.

8 Availability

MagmaDNN releases are hosted on BitBucket[1]. Major releases and downloads are kept here while development takes place on the project's github[2]. The framework is written in c++ and has been tested on the GCC and Clang toolchains. MagmaDNN has been tested on

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

[2] `https://github.com/MagmaDNN/magmadnn`

Ubuntu, Aarch, and MacOS and is likely to work on any *nix based system. No system calls are used, so it is likely to work in a Windows environment with little modification.

Acknowledgement

References

[1] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.

[2] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[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. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[4] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014.

[5] Mnist. `http://yann.lecun.com/exdb/mnist/`, 1998. [Online; accessed 11-March-2019].

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.