Minerva: A Scalable and Highly Efficient Training Platform for Deep Learning

Minjie Wang[1], **Tianjun Xiao**[2], **Jianpeng Li**[3], **Jiaxing Zhang**[3], **Chuntao Hong**[3], **Zheng Zhang**[1]
[1]New York University, [2]Peking University, [3]Microsoft Research

Abstract

The tooling landscape of deep learning is fragmented by a growing gap between the generic and productivity-oriented tools that optimize for algorithm development and the task-specific ones that optimize for speed and scale. This creates an artificial barrier to bring new innovations into real-world applications. Minerva addresses this issue with a layered design that provides language flexibility and execution efficiency simultaneously within one coherent framework. It proposes a matrix-based API, resulting in compact codes and the Matlab-like, imperative and procedural coding style. The code is dynamically translated into an internal dataflow representation, which is then efficiently executed against different hardware. The same user code runs on modern laptop and workstation, high-end multi-core server, or server clusters, with and without GPU acceleration, delivering performance and scalability better than or competitive with existing tools on different platforms.

1 Introduction

This paper examines the tooling of deep learning that has made of its innovations possible. We notice a widening gap between the generic and productivity-oriented tools and the more task-specific ones. Yet, from the perspective of a complete ecosystem, these two classes of tools are intimately related.

To begin our discussion, we note that the core innovations are not by themselves new; going deep, recurrent and convolutional are some of the key ingredients that were proposed many years ago. The success of deep learning and the great amount of remaining puzzles inspire new algorithmic innovations in this space, deriving models that sometimes depart significantly from a straightforward feed-forward network topology [1, 2, 3]. The new models are typically developed in productivity-oriented tools such as Matlab or Octave. Because of their ease of programming coupled with a significant collection of optimization and visualization routines, such tools are ideal in breeding ground of new innovations. However, in order to become relevant, these innovations must be ultimately tested on real-world data set. The key problem of doing so in such tools is their incompetence to handle large volume of data and efficiently exploit advanced hardware that are economically viable due to the trend of big data computing.

Meanwhile, there is a growing number of new task-specific tools. Gravitated towards a few known network architectures, these tools trade-off programmability in favor of speed and scale. For example, many image classification tasks are delivered via variations of the deep convolutional network [4], large-vocabulary speech recognition tasks typically employ multi-layer networks [5, 6], and those in natural language processing use recurrent architectures [7]. These tools are adept to leverage the new breed of hardware, including GPU acceleration for matrix computation (e.g. Cuda-Convnet [8] and Caffe [9]), distributed multi-core CPU-based cluster (e.g. DistBelief [10]), or GPU cluster [11]. Consequently, they optimize for certain kind of networks, with implementations that are tightly coupled with the underlying hardware, and the network models sometimes take shortcuts in order to circumvent hardware limitations. For example, the object classification model in [10] and [11] disallow sharing of parameters across machine boundaries, even when such sharing is known

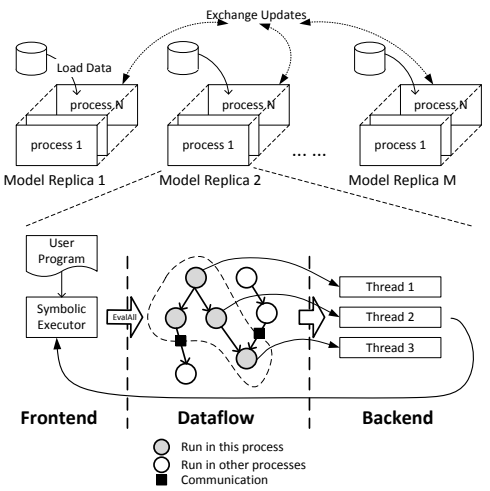

Figure 1: The overall architecture of a Minerva system.

Category	APIs

Building Model	`Layer layer = AddLayer(dim)`
	`AddConnection(layer1, layer2, type)`
	`AddConvConnect(layer1, layer2,...)`

Creating Matrix	`A = Matrix(layer1, layer2, init)`
	`A = Matrix(layer, dim, init)`
	`A = Matrix(dim, layer, init)`
	`vector<Matrix> LoadBatches(layer, dim, batch_size, filename)`

Element-wise OP (unary)	`B = A.Map(func)`
	`B = A op val`
	`B = val op A`

Element-wise OP (multivariate)	`C = A op B` `C = EleWiseProd(A1, A2, ...)`
Matrix Multiplication	`C = A * B`

Aggregation	`B = A.Agg(agg_type)`
	`B = A.AggPerRow(agg_type)`
	`B = A.AggPerCol(agg_type)`

Matrix Layout	`B = A.Transpose()`
Model Paralelism	`SetPartition(layer, part_plan)`

Data Paralelism	`RegisterToParameterServer(para_set)`
	`PushToParameterSever(para_set)`
	`PullFromParameterSever(para_set)`

Table 1: The Minerva API.

to be effective to constrain model capacity to avoid overfitting. While diverse, task-specific tools share the common spirit that the flexibility is limited to network reconfiguration. Their strengths are speed and scale, whereas the long turn-over time and difficulties to debug prohibits them to make dramatic algorithmic changes.

The dilemma is that we are now facing a broken pipeline: innovations produced on one end are increasingly harder to be tested on the other. It is legitimate to argue that this is a result of pragmatic progress, and should be accepted as facts. However, we raise the question as whether this conclusion is superficial, and the apparent gap between productivity and performance is reconcilable. We begin our pursuit with the position that it is impractical to replace tools such as Matlab in the tool chain. Instead, we hypothesize that, with a principled system building approach, it is possible to retain much of the programmability of the productivity-oriented tools at the programming language level, and at the same time exploit modern hardware efficiently underneath.

We have prototyped a system called Minerva to meet this challenge, and this paper reports our progress. Minerva draws inspirations from both classes of tools, and tries to build a bridge in between. Minerva proposes and programs directly against a matrix-based API, preserves the Matlab-like, imperative and procedural coding style. This is unlike tools such as Theano [12]. Consequently, if needed, migrating a Matlab program to Minerva is more straightforward. At runtime, a Minerva program is translated into an internal dataflow representation, which is executed efficiently on different hardware. The same user code runs on modern laptop and workstation, high-end multi-core server, or server clusters, with and without GPU acceleration. This modular and layered design isolates runtime complexity from end users. We have carefully engineered the system, such that it delivers performance and scalability better than or competitive with existing tools on different hardware platforms. For example, Minerva's implementation of the convolutional network is as compact as the Matlab version, and reaches 42.7% top-1 and 29.9% top-5 error rate on ImageNet 1K category classification task, with speed that is 2x faster than ConvNet and close to Caffe on a single GPU. Our implementation of recurrent neural network delivers 28X and 124X performance gain on a 16-core CPU and GPU, respectively, against a widely used hand-tuned single thread implementation.

We acknowledge that programmability is not equal to productivity, but it is a critical and necessary condition. To make Minerva useful, and we are actively working towards releasing it as an open source project. In the remainder of the paper, we will describe the overall architecture (Section 2), the design of programming model (Section 3) and runtime (Section 4). We share our experience and preliminary results in Section 5. Finally, we conclude in Section 6.

2 System Overview

The overall architecture of Minerva system is illustrated in Figure 1. A Minerva process executes every training iteration by alternating between two phases, the first phase symbolically executes the

```
#include "minerva.h"

float Sigmoid(float x) {
  return 1.0 / (1.0 + exp(-x));
}

void MLP_Training() {
  // Building Graph Model
  Model model;
  Layer layer1 = model.AddLayer(500);
  Layer layer2 = model.AddLayer(1000);
  Layer layer3 = model.AddLayer(10);
  model.AddConnection(layer1, layer2, FULL);
  model.AddConnection(layer2, layer3, FULL);
  SetPartition(layer1, 2); SetPartition(layer2, 2);
  model.Finalize();

  // Setting Hyper-parameters
  float rate=0.1; int numEpochs=66, batchSize=100;

  // Initializing matrices
  Matrix W = Matrix(layer2, layer1, RANDOM);
  Matrix V = Matrix(layer3, layer2, RANDOM);
  Matrix b(layer2, 1, RANDOM);
  Matrix c(layer3, 1, RANDOM);
  vector<Matrix> inputs = LoadBatches(layer1,...);
  vector<Matrix> labels = LoadBatches(layer3,...);

  // Data Parallelism: Registering Global Parameters
  ParameterSet pset;
  pset.Add("W", W);  pset.Add("V", V);
  pset.Add("b", b);  pset.Add("c", c);
  RegisterToParameterServer(pset);

  // Learning Procedure
  for(int epoch = 1; epoch <= numEpochs; ++epoch) {
    for(int i = 1; i <= inputs.size(); ++i) {
      Matrix x = inputs[i];
      // Feedforward
      Matrix y = (W * x + b).Map(&Sigmoid);
      Matrix z = (V * y + c).Map(&Sigmoid);
      // Backpropgation
      Matrix ze = EleWiseProd(z, 1-z, z-labels[i]);
      Matrix ye = EleWiseProd(y, 1-y, W.Transpose()*ze);
      // Updating Weight
      W -= rate * ye * x.Transpose() / batchSize;
      V -= rate * ze * y.Transpose() / batchSize;
      b -= rate * ye.AggPerRow(SUM);
      c -= rate * ze.AggPerRow(SUM);
    }
    // Data Parallelism: Sync Parameters
    if (epoch % 3 == 0) PushToParameterSever(pset);
    if (epoch % 6 == 0) PullFromParameterSever(pset);
    EvalAll();
  }
}
```

Figure 2: The Minerva program of a 3-layer neural network

user code to generate the dataflow representation, whereas the second phase evaluates it concretely. The dataflow representation is platform independent, and is identical in each process. This layered design separates the support for language flexibility and execution efficiency. The cost to generate the dataflow graph is negligible, but it enables the rest of the system to focus on performance.

In deep learning, parallelism exists in two mutually complementary forms (DistBelief [10]). For *model parallelism*, multiple processes are grouped into *model replica* to train the same model. The processes partition the dataflow graph deterministically. Each process works on its portion while resolving data dependency among them via asynchronous remote procedure calls, and thus maximumly hides communication overhead. We perform automatic inference to derive near-optimal data placement among processes to preserve locality, which is important when we run on a cluster of machines. Note that by utilizing the specially-designed programming model, our architecture derives data placement in a complete static and de-centralized way, and thus gets rid of the central coordination needed in other computing engines for big data [13, 14, 15].

For *data parallelism*, multiple model replicas work on different partitions of the data sets. Each replica has one local *parameter server* that it updates to. The parameter servers of different replicas asynchronously exchange updates among themselves to keep the models loosely synchronized, possibly via a centralized master.

The system is logically partitioned into two tiers (Figure 1). The upper-level runs data parallelism across model replicas, and the lower-level exploits model parallelism within each model replica. Minerva processes exploit two major hardware advantages: GPU acceleration, and multi-core CPUs. Doing the former is relatively straightforward, the dataflow graph is always evaluated in GPU if it is available. The minimum configuration is one model replica containing only one process coupled with one parameter server process, which can be run on a laptop. Scaling out the system is achieved by adding more replicas, more processes of each replica, or both.

3 Programming Model

A valid design alternative is to express the model using the graph programming paradigm since, after all, the neural network is a graph. Indeed, our first prototype leverages the gather/scatter/apply API in GraphLab [16]. One immediate benefit is that it directly expresses all parallelism inherent in the algorithm. The problem is that the entire training procedure is now broken into isolated vertex programs. Such fragmentation leads to tedious inference of the overall program flow, and debugging becomes a nightmare, which entirely defeats the purpose of being user friendly.

The detour to graph programming is nevertheless beneficial. It reveals the insight that the training programs are a series of matrix operations induced from the graphical representation of the model. Our approach combines the strength of both graphical representation for the model and matrix interface for the computation, which can be summarized as "build as a graph, program with matrices".

3.1 Minerva API

Minerva defines a set of APIs (Table 1) to cover three distinctive stages of a deep-learning algorithm: describing the model architecture, declaring the primary matrices, and finally, specifying the learning procedure. The programming style is imperative and procedural, resembling how one develops algorithm in Matlab. Figure 2 shows the code snippet of a simple 3-layer neural network.

A Minerva *model* has two entities: *layer* and *connection*. User declares a layer by calling `model.AddLayer(dim)`, and `model.AddConnection(l1, l2)` to connect the two layers. In this example, `dim` is a scalar that specifies number of neurons, and the type of connection is fully connected. Line 9-16 builds the model in Figure 2 (partition will be explained shortly). So far, we have found that these two simple APIs are sufficient to describe all the models that we have studied, including recurrent network, and multi-modal network [3]. This explicit graphical structure acts as a scaffold, against which all variables are directly or indirectly pinned. This is critical for Minerva to declare and extract parallelism, as we shall explain shortly.

The basic data type in deep learning algorithm is matrix. For example, the hidden layer's activation is computed with $y = sigmoid(Wx + b)$ (line 40). To carry out this statement, matrix W, x and b must be declared. In Minerva, matrices are either directly derived from the model (e.g. `W`, `b` and `inputs`), or are results of computation (e.g. `y`). In either case, they are attached to the model, via either explicit declaration or program inference. In this example, `x` is attached to `layer1`, `y` and `b` are attached to `layer2`, and `W` is attached to the connection in between (see Figure 3(a)).

Writing training procedure is straightforward. Line 38-49 corresponds to the feed-forward pass, back propagation and the weight update. The style of the program is similar to many of the Matlab implementations of deep learning algorithms we have studied. We incrementally add operators as we implement various deep learning networks. Table 1 summarizes the major ones. The most important types are *element-wise* operator and an extended version of *multiplication* operator. In addition, there are various forms of aggregation operations for error calculation or normalization.

CNN (convolutional neural network) requires some special treatments. Recall that each layer in CNN is a stack of feature maps, and each of which is a 2-D or 3-D data. Thus, the input and output of each layer actually has a internal multi-dimensional structure. Furthermore, instead of a fully connection, a filter is replicated and tiled between the two layers. All these structural information are given by a single interface `AddConvConnect(layer1, layer2,...)`. Then, the `Matrix` class operators such as + and $\star$ automatically take appropriate implementation (see Section 4).

3.2 Expressing parallelism

Minerva runtime executes the user code symbolically to build an internal dataflow graph. The dataflow fragment that corresponds to one statement of the feed-forward pass (line 40) is depicted in Figure 4(a). Fragments of individual statement are stitched together, and when an `EvalAll()` statement is hit (line 54), the entire dataflow representation is evaluated concretely – a form of lazy evaluation. This dataflow representation expresses all the potential parallelism inherent in the training algorithm. To improve performance further, user can elect to employ model parallelism (line 15), data parallelism (line 30-33, 52-53), or both of them. These options are entirely optional, and the system runs correctly without them.

Model parallelism. The algorithm-inherent parallelism is typically too low to be exploited by a multi-core and/or a cluster of servers. Increasing the amount of parallelism can be done by partitioning the models, using the `SetPartition(layer, partplan)` interface. Line 15 instructs that `layer1` and `layer2` be both partitioned into two equal portions, and thereby splits the original model (Figure 3). After partitioning, the equivalent dataflow graph (Figure 4(b)) now expresses more parallelism. Note that all matrices attached to the splitted segments of the model are automatically partitioned via program inference. The split leads to their partitioning scheme depicted in Figure 3(b).

Data parallelism. Minerva can train multiple model replicas in parallel. These replicas execute the same user code but on different set of machines, and synchronize through a *logically centralized* parameter server. To enable data parallelism, we first register to the parameter server the set of parameters to be synchronized (line 30-33). User can control the pace to update and refresh its parameter set (line 52-53). The update sends deltas, whereas the refresh downloads the entire new set. Since weight updates are commutative and associative, the logic in parameter server is simple.

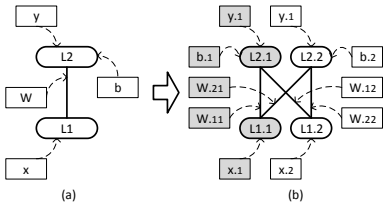

Figure 3: Partitioning of layers and their attached matrices. The result partitions (right) are assigned to two processes (gray and white).

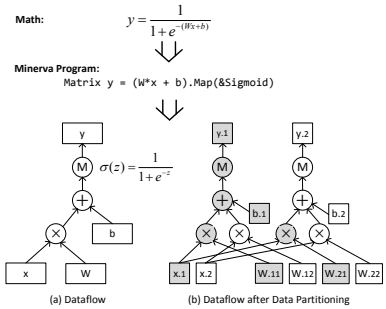

Figure 4: Statement and its dataflow fragment. The data and computing vertexes with different colors reside on different processes.

3.3 User-enabled optimization

The most important optimization to get good performance is to hide various I/O latencies. These optimizations are often scenario and platform dependent. We demonstrate how Minerva achieves them with only a few lines of code change.

For example, when data volume is too large to fit in memory, loading data from disk should proceed in parallel with computation. In Minerva, this can be done by letting a temporary matrix prefetch the next batch of data, and switching to it at the end of the iteration, as illustrated in Figure 5a. Training on the current batch and loading of the next are therefore proceed in parallel.

Another example is to hide network latency in model parallelism. Considering a matrix that represents the neuron activations of one layer. The matrix is partitioned among multiple processes and has number of rows equal to the mini-batch size. A single propagation needs these processes to send its portion of this matrix to all others, incurring bursty and expensive data shuffling across network. Although our runtime already performs partial computation as data arrives, this expense is still visible. We deal with this problem by simply carving up one mini-batch into smaller mini-mini-batch, enabling pipelining across layers (Figure 5b).

```
thisBatch = LoadBatches(..., inputfiles[1]);
for(int n = 1; n < numBatchFiles; ++n) {
 nextBatch = LoadBatches(..., inputfiles[n+1]);
 ...; // Training on thisBatch
 EvalAll();
 thisBatch = nextBatch;
}
```

(a) Batch prefetching

```
for(int i = 0; i < numMiniBatches; ++i) {
 for(int j = 0; j < numMiniMiniBatches; ++j) {
  // Calculate gradient on the j'th mini-mini-batch
  gradients[j] = ...;
 }
 for(int j = 0; j < numMiniMiniBatches; ++j) {
  parameters -= rate * gradients[j]; // Gradient decent
 }
 EvalAll();
}
```

(b) Mini-mini batch pattern

Figure 5: User-enabled optimizations

4 System design

As shown in Figure 1, each Minerva process has identical flow of execution. It first steps through the model-building part of the user code (e.g. line 9-16 of Figure 2), generating an internal graphical representation of the model, which will be referenced to infer data placement decision and parallel execution plan later. The following part (Line 36-55 of Figure 2) is training procedure. Each iteration is decomposed into two phases. The first phase processes statements lazily to form a dataflow graph, whose vertexes are either *computing vertex* or *data vertex*. This phase stops when hitting a `EvalAll()` statement, and the graph is then concretely evaluated. By definition, the inputs of the dataflow graph are data vertexes that are ready at this point (e.g. load training data). The evaluation of a computing vertex fills downstream data vertexes and progressively triggers further evaluation in the dataflow graph. All ready vertexes are queued up and processed by a thread pool. When all of them are processed, the control returns to symbolic execution phase of the next iteration.

Support heterogeneous hardware. Computing vertexes are virtual pointers that carry out predefined operations (see Table 1). These pointers implement polymorphism to call different libraries

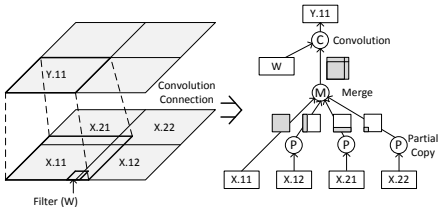

Figure 6: Parallel convolution and its dataflow representation.

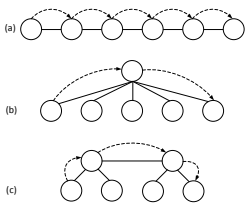

Figure 7: Three possible topologies of parameter servers with their longest update paths.

and engines. We use Intel's MKL library or Nvidia's CUBLAS library to perform matrix computation when running on a multi-core CPU or GPU, respectively. For GPU training, the state of model always live in GPU, and training data is on-demand fetched from CPU memory.

Model parallelism. Processes in one model replica update the *same* model in parallel. Note that theoretically we can arbitrarily increase the amount of parallelism in the dataflow graph by using more partitions to split the layers (using the `SetPartition` API; see Section 3). Since all processes generate identical graphs, parallel execution is then reduced to a coordinated and partitioned execution plan among them.

The important considerations regarding a parallel execution plan are load-balance, coordination overhead and data locality; they are critical especially in a cluster environment. Note that splitting all the layers evenly achieves load-balance in normal cases, and that is the default partitioning strategy. Coordination overhead could thus be removed completely by this statically partitioning policy. During execution, each process executes computing vertexes that it owns and skips others (See Figure 4 for an example). The ownership of a computing vertex is decided by the location of its input and output data vertexes. Therefore, the placement of data vertex uniquely determines the assignment of workload among processes. In theory, one can take any policy to place data vertex, as long as it is deterministic, but it is also tricky to get optimal performance. A good solution should maximizes locality, so that the computing vertexes have inputs local to them most of the time. Our key observation is that data attached to the same layer are also likely to operate together. Our placement policy thus works by enforcing those data to be co-located, and fixing the rest deterministically (see Figure 3). Note that all data are attached to the model, which reveals some clues about what data should be co-located. The insight is that the data that attached to the same model part are likely to be operated together. Consider Line 43 in Figure. 2, both `z` and `labels` are attached to `layer3`, and the $-$ and `EleWiseProd` operations could be perfectly paralleled without incurring any communication, if all these operands are co-located. In fact, these operators comprise majority of operations in a deep learning algorithm. By placing these data together, most of the operations will find their input locally. For other operators like multiplication, they are computationally intense and induce network traffic no matter how smart a placement algorithm is. In practice, this scheme works very well against other more complex alternatives we have considered. For any data that does travel across process boundary, we insert pointers to remote send and receive calls on the edge in the dataflow graph. The transfer is triggered when a data is produced, and thus maximumly hides communication latency.

Handling convolutional network. Convolution is a case that needs special handling owing to its different inherent parallelism. Parallelism in CNN proceeds by dividing up the input and output data into smaller patches. A set of filters local to a process convolves over its partition. The dataflow construction of a convolution operation considers boundary conditions, and fetches data belonging to different partitions (Figure 6). The convolution operator (`C` in Figure 6) follows the same optimization as in [11] by first forming larger matrix from non-consecutive inputs and then performing matrix multiplication. The gain of multiplication outweights the overhead of additional memory copy, and is especially effective for GPU.

Data parallelism. In Minerva, there is one parameter server local to each machine. It receives local updates and asynchronously exchanges them with other parameter servers. For GPU-based training, the update frequency is very high (e.g. $<$ 1s for one mini-batch in ImageNet training), and we have found the master-slave architecture as done in DistBelief non-scalable. We have developed a protocol that enables parameter sharing as long as the parameter servers form a loop-free topology. The

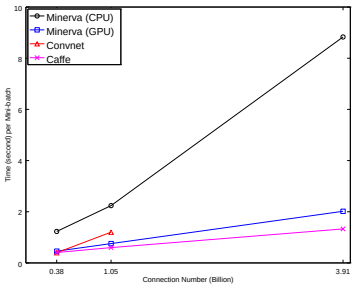

Figure 8: Scale-up results for CNN training.

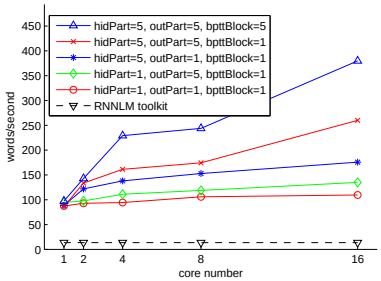

Figure 9: Scale-up results for RNN training.

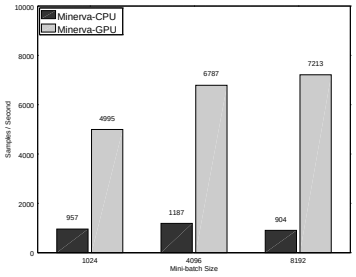

Figure 10: The throughput of speech-net training at different mini-batch size.

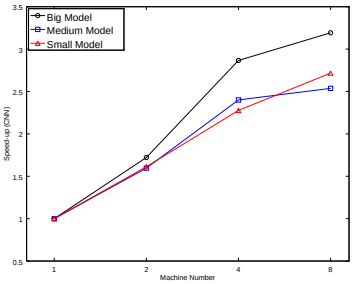

Figure 11: The throughput speed-up with increasing machine numbers for CNN.

protocol resembles the algorithmic structure of belief propagation in Bayes network [17]. Basically, a parameter server merges its own updates with those from all its connected neighbors, and gossips to each of them the missing portion. The protocol is proved to achieve eventual consistency, and tolerates dynamic fail and rejoin of any machine. It gives the flexibility of trading off model divergence with scalability. For example, a topology where all parameter servers form a chain has the best load balance but the slowest propagation, and the master-slave topology is the opposite but performs poorly under high update rate. A twin-master topology strikes a balance in between (Figure 7). We refer user to [18] for the detailed design.

5 Experience and Evaluation

We have implemented the most common deep learning networks and a variety of popular regularizations. We have also built gradient checker, enabling us to catch a number of subtle bugs. The user code is typically as compact as their Matlab versions. We also support scripting a network just as other task-specific tools to speed up common trainings. We have verified model accuracy with various regression tests, against Matlab code as well as other systems. Minerva's implementation of the CNN model [4] gives 42.7% top-1 error rate on the ImageNet 1K classification task. Our study of incremental and hierarchical learning [19] is done entirely through Minerva.

We now report results of system performance (i.e. speed and scale) using three networks; they are chosen because the common baselines are well known. The first is a well-known *CNN* model [4] for ImageNet 1K classification. The *Speech-net* is a conventional deep feed-forward network. Its default configuration contains 1100 input neurons, 8 hidden-layers of 2K sigmoid neurons and a 9K softmax output layer. This is a large-vocabulary speech recognition model our sibling speech research group uses. Finally, *RNN* is a recurrent network with 10K (vocabulary size) input, 1000 hidden and 10K flat output units. Its core training loop has fewer than 60 lines of code.

We have done extensive evaluations across different platforms; due to space limitation we can only present a few results in detail. For desktop productivity, Minerva is 1.5x $\sim$ 2.0x and 15x faster than Matlab for speech-net and a scaled-down CNN, respectively, on a 4-core workstation; we get qualitatively similar results on a Lenovo X1 labtop. In all experiments below that run on CPU, model partitioning is turned on to create more parallelism. The optimal partition scheme is task- and platform-dependent, and can be discovered by simple profiling. For example, the CNN model running on a 16-core server with a 16-way partition enjoys a 4.6x speed up against no partition.

Scaling up. Figure 8 shows the results of CNN on a 16-core Xeon E5-2665 CPU and GTX Titan GPU with mini-batch size of 128, comparing against ConvNet and Caffe. We vary the model size by changing number of feature maps. The small, medium (default) and big model correspond to 0.38/1.05/3.91 billion connections. Focus first on our CPU performance, scaling the model size scales the running time almost linearly. On the GPU side, Minerva runs 2x faster on the medium model than ConvNet, and close to Caffe (0.75 vs. 0.60). For the large model, only Minerva and Caffe can complete the run. We also evaluate the training speed using larger mini-batch size. For the small model, Minerva is about 20% and 24% faster than Caffe with 256 and 512 mini-batch size respectively. For medium and big model, Minerva's and Caffe's performance are close. Further improvements could be achieved by using the unwrapping trick proposed/used in [9, 20].

Figure 9 shows the performance of RNN training with RNNLM and Minerva on a 16-core server, where `hidPart` and `outPart` are the partition number in hidden and output layer, respectively. `bpttBlock` is the model update cycle. Without model parallelism, Minerva performs more than 6 times better than RNNLM on a single core. The reason is that we batch 100 sentences in one training pass and apply matrix-matrix multiplication, unlike RNNLM which only uses hand-tuned matrix-vector multiplication. Without model parallelism, there is limited room for improvement with more cores, and we reach only 110 words/s with 16 cores. Partitioning the model at hidden, the output and at both layers successively unlock more and more parallelism without compromising model accuracy, to around 250 words/s, nearly 18X performance gain over RNNLM. With lower model update frequency (by setting `bpttBlock` = 5), we reach 380 words/s. Finally, the same code trains unmodified on GPU and reaches 1674 words/s. This is another 4.4X speedup over 16-core CPU training and more than two orders of magnitude faster than RNNLM(124X).

On CPU, scaling up the model to 50K, 100K, 500K and 1M input/output units decreases the throughput proportionally, to 94, 47, 9.3 and 4.5 words/s, respectively. This is because of the dominating cost of hidden-output computation, which is proportional to input/output layer size. GPU obtains up to 7.3X speedup over CPU but stops at the 500K configuration, as the model exceeds GPU memory.

The result of speech-net with varying mini-batch size is depicted in Figure 10. The CPU performance varies little around 1K samples per second while larger mini-batch improves the utilization of GPU, and with 8K mini-batch, we reach nearly 7K samples per second. This result matches well with hand-tuned GPU implementation from our sibling speech research group.

Scale-out. The speedup performance of CNN over a 4-core cluster is shown in Figure 11, using user-level network I/O optimization with mini-mini-batch (see Section 3). It achieves nearly 4x speedup with 8 machines. Consistent with the finding in DistBelief [10], speech-net scales relatively poorly, due to the intense cross-machine communication induced by the all-to-all connections. It would seem that the 64-GPU cluster [11] has achieved better scalability of model parallelism. The critical difference is that the benchmark used in that system is a sparse auto-encoder with local receptive field, with *no* weight sharing across machine boundaries. Our CNN is a classifier with a 3-layer fully-connected layers; model parallelizing such a network remains an open challenge.

6 Conclusion and ongoing works

Minerva is a domain-specific instead of task-specific engine. The difference is subtle but important. The challenges we resolve with the modular and layered design is to achieve programmability and performance simultaneously in one coherent framework.

Our ongoing works are multifaceted. We are very close to making GPU data parallelism work. GPU model parallel meets performance problem of using GPU-direct to transfer data among NVidia cards, but we believe this can be solved soon. More problematic is the intense traffic induced by partitioned fully-connected layers. A better alternative maybe one in which some of the fully-connected layers are not split but pipelined [21], Minerva allows such flexibility and we plan to experiment.

We are implementing more advanced deep-learning algorithms and optimizations, including Quasi-Newton methods, Conjugate Gradient Descent and Hessian Free approaches. There is a large body of algorithmic work that Minerva should support if we want to make it useful. We recognize that this has to take community's effort, and are actively working towards releasing Minerva as an open source project.

References

[1] Kihyuk Sohn, Guanyu Zhou, Chansoo Lee, and Honglak Lee. Learning and selecting features jointly with point-wise gated {B} oltzmann machines. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 217–225, 2013.

[2] Yichuan Tang, Ruslan Salakhutdinov, and Geoffrey Hinton. Robust boltzmann machines for recognition and denoising. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 2264–2271. IEEE, 2012.

[3] Nitish Srivastava and Ruslan Salakhutdinov. Multimodal learning with deep boltzmann machines. In *Advances in Neural Information Processing Systems 25*, pages 2231–2239, 2012.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoff Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, pages 1106–1114, 2012.

[5] Abdel-rahman Mohamed, Dong Yu, and Li Deng. Investigation of full-sequence training of deep belief networks for speech recognition. In *INTERSPEECH*, pages 2846–2849, 2010.

[6] Frank Seide, Gang Li, and Dong Yu. Conversational speech transcription using context-dependent deep neural networks. In *INTERSPEECH*, pages 437–440, 2011.

[7] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[8] Alex Krizhevsky. High-performance c++/cuda implementation of convolutional neural networks.

[9] Yangqing Jia. Caffe: An open source convolutional architecture for fast feature embedding. `http://caffe.berkeleyvision.org/`, 2013.

[10] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[11] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1337–1345, 2013.

[12] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, volume 4, 2010.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[14] Zhengping Qian, Xiuwei Chen, Nanxi Kang, Mingcheng Chen, Yuan Yu, Thomas Moscibroda, and Zheng Zhang. Madlinq: large-scale distributed matrix computation for the cloud. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 197–210. ACM, 2012.

[15] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[16] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[17] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[18] Minjie Wang, Hucheng Zhou, Minyi Guo, and Zheng Zhang. A scalable and topology configurable protocol for distributed parameter synchronization. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 13. ACM, 2014.

[19] Xiao Tianjun, Zhang Jiaxing, Yang Kuiyuan, Peng Yuxin, and Zhang Zheng. Error-driven incremental learning in deep convolutional neural network for large-scale image classification. In *ACM Multimedia*, 2014.

[20] Patrice Simard, David Steinkraus, and John C Platt. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 3, pages 958–962, 2003.

[21] Xie Chen, Adam Eversole, Gang Li, Dong Yu, and Frank Seide. Pipelined back-propagation for context-dependent deep neural networks. In *INTERSPEECH*, 2012.