Fast and Scalable Deep Learning on the Minerva System

Hongyu Zhu

Master of Science

School of Computer Science

McGill University

Montreal, Quebec

2015-04-15

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science

Copyright ©2015 Hongyu Zhu

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank all persons who have ever helped me on this exciting project:

- The leader of the team, Prof. Zheng Zhang from New York University - Shanghai, for giving me a position to work with many excellent researchers, giving me many helpful advices, and granting me to use this project on my Master thesis [1] .
- My Master and thesis supervisor at McGill University, Prof. Xue Liu, for allowing and supporting me to join this team, and being nice to me on this thesis.
- My teammates Minjie Wang from New York University, Chuntao Hong, Yutian Li from Microsoft Research Asia and Jie Shao for their hard works, and Tianjun Xiao, Jiaxing Zhang from Microsoft Research Asia, for guiding me on my AlexNet and LSTM works.
- My family in China for their support all the time
- Mr. Mingyuan Xia for helping me review this thesis
- My friend Naomi Serfaty for helping me on the French abstract

[1] There is a previous publication [39] on NIPS 2014 regarding this project. This paper presents some latest progress.

ABSTRACT

The development of deep learning has been driven by both algorithm innovations and high performance computing supports for big data processing in real world. The gap between productivity-oriented tools targeting algorithm innovations and task-specific tools optimized for performance and scalability is growing. This created a barrier to bring new algorithm advancements into real-world applications. To bridge this gap, our team presents a coherent framework solution, called Minerva, which renders programming flexibility and execution efficiency with a layered design. It provides matrix-based APIs, leading to user programs as compact as Matlab ones. The user code is dynamically translated into a dataflow representation which enables underlying executions against various hardware environments. Based on Minerva, we have implemented some modern deep learning examples, such as CNN and LSTM. For many modern deep learning architectures, Minerva is able to deliver performance and scalability competitive with or better than existing task-specific tools.

ABRÉGÉ

Le dveloppement de lapprentissage profond est conduit la fois par linnovation algorithmique et par la haute performance dans le support technique du traitement de grosses donnes dans le monde daujourdhui. Lcart entre les outils pour linnovation algorithmique et les outils pour des taches spcifiques optimisant la performance et la flexibilit est croissante. Ceci a crer une barrire lapport des nouvelles avancs algorithmiques dans les applications du monde daujourd'hui. Pour crer un lien entre cet cart, on prsente une solution sous forme de structure logicielle, appel Minerva. Elle rend la flexibilit de la programmation et lefficacit de lexecution avec des couches de design. Elle offre des API bass sur des matrices, qui mne a des programmes dutilisateurs aussi compact que Matlab. Le code dutilisateur est dynamiquement traduit en une reprsentation de flux de donnes qui permet lexecution contre des diffrents environnement materiel. Bass sur Minerva, nous avons mis en place quelques exemples d'apprentissage en profondeur modernes, comme CNN et LSTM. Pour plusieurs architectures moderne de lapprentissage profonds, Minerva est capable de dlivrer une performance et une flexibilit comptitive ou meilleure que celles outils ddis a des taches spcifiques qui existent dj.

CHAPTER 1
Introduction

1.1 Motivation

In the family of machine learning approaches, deep learning has now emerged as one of the most promising members. During past several years, it has produced groundbreaking results across various domains of applications such as image recognition, speech recognition, computer vision and natural language processing [14, 26, 32]. On common databases such as ImageNet, deep learning architectures such as deep neural networks, convolutional neural networks also claim the state-of-the-art performances [20].

Unlike traditional shallow neural networks, deep learning assumes a much deeper and cascading hierarchy of non-linear neurons. These neurons are trained to capture complex feature representations from the data. Higher-level features are represented in terms of lower-level features. The ideas of going deep, convolutional and recurrent themselves were proposed many years ago, but they were not the mainstream when they were just proposed. Three reasons lead to the popularity of deep learning today: the increasing processing abilities of hardware, the increasing size of data which comes from real machine learning applications, and the innovations of training techniques (such as dropout and restricted boltzman machine pre-training).

General Purpose Graphic Processing Units (GPGPUs) are a recent breakthrough in parallel computing field. There are typically about one thousand cores in a modern GPU with multiple streaming processors. This many-core architecture of a GPU has demonstrated its powerful potential in high performance computing. This hardware innovation provides strong supports for

the idea of going deep and wide. Since NVidia's CUDA programming model for multi-GPU becomes more and more popular, libraries such as CUBLAS for matrix computations are developed for advanced algorithm design. These progresses enable many further advancements in deep learning.

To fulfill more complex classification tasks, more and more feature detectors are requested for improving the learning ability. The size of data grows much larger as well, so that the model can fit these parameters with sufficient training data. This leads to complex models and lets the training of a deep neural networks become both computationally intensive and I/O intensive. To see this challenge, take Alexs model [20] for the ImageNet dataset for example. The model contains 7 hidden layers, 650 thousand neurons and 60 million parameters. To train this model with 2 GPUs (GTX580) will take one week. The model [10] that won 2012 ImageNet 1K category competition is three times larger than this model in terms of the number of parameters, and for the large-vocabulary speech recognition problem, which is the first successful application of deep learning, the model contains 7 hidden layers, with 429 input neurons, 32 thousand output neurons, and 3 thousand neurons per hidden layer. Such a network contains 156 million parameters. To train 675 million input samples (98GB in total) with a single Tesla-class GPU, it takes 75 days to converge to a good results. On high-end servers it would have to take months to train. The need for a scalable system which can train fast with the help of general purpose hardware is imminent.

Systems like GraphLab [22] and Spark [41] have shown good scalability and efficiency when dealing with large scale applications. However limited by their programming interfaces and especially their abilities to express matrix operations, only a few deep neural networks were implemented using these

systems. People in machine learning community usually use productivity-oriented tools such as Matlab and Octave. Their matrix-based API enables the programming coupled with optimizations and visualizations of the model, which are crucial routines for deep learning research works. The innovations of training techniques are typically developed using these tools. The drawbacks of these tools are also obvious. Their abilities to handle large amount of data from real world are poor. The trend of big data computing requires the tools that exploit hardware computability efficiently.

To implement given neural network models and process real data, people often uses task-specific tools. Regarding different tasks there are typical types of topologies (e.g. deep convolutional neural network for large-scale image recognition problems, and recurrent neural network for natural language processing). These tools target the trade-off between programmability and scalability. They are typically optimized for certain kinds of network topologies by utilizing the benefit of GPU acceleration of matrix computation (e.g. Cuda-ConvNet [19] and Caffe [17]), GPU cluster or distributed multi-CPUs cluster (e.g. DistBelief [8]). This often leads to tight couples between the implementation and the underlying hardware environment. For example, Caffe aims at optimization on a single CPU-GPU environment, which makes multi-GPU training on Caffe impractical to implement. The hardware limitations could sometimes become an obstacle for the innovations of algorithms.

1.2 Our Contributions

So the dilemma is right affront: innovations of algorithms produced on one platform are hard to be tested on the other. The gap between performance and productivity is widening. We presented a prototype system solution to meet this challenge. By this approach we can retain the programmability

of the productivity-oriented tools, and meanwhile take advantages of modern hardware environment.

Our prototype system, called Minerva, examines the advantages of these classes of tools and tries to build a bridge over the gap. The programming interface of Minerva is directly against a Matlab-like API, preserving compact and matrix-based coding style. To migrate a Matlab or NumPy program to Minerva is straight forward if necessary. The user program is then transferred to a data flow representation, which expresses the internal parallelism maximally. This enables the system to exploit the heterogeneous hardware efficiency underneath. (This part is a teamwork by my teammates Minjie, Yutian and Chuntao.)

Based on Minerva's underlying data flow engine, we explore the details on efficient implementation of modern deep learning architectures. The APIs are carefully designed so that good performance can be achieved without losing programmability. Especially, Minerva enables users to exploit the computation power of multi-GPU on the training procedure, delivering good scalability. (This part is conducted by Tianjun and me, with helps from our teammates on some optimization details.)

1.3 Thesis Organization

The remainder of this thesis is organized as following. In Chapter 2 we introduce the general background of deep learning research. We will also introduce an overview of current programming platforms for deep learning applications in this Chapter. We will then describe the overall architecture (Chapter 3), the design of the programming model (Chapter 4), and the runtime implementation (Chapter 5) of our system. In Chapter 6 and Chapter 7, we share our implementation techniques and current experiment results on

some modern deep learning architectures. Finally, we summarize in Chapter 8.

CHAPTER 2
Background

2.1 Related Work

Although deep learning research has a history of decades, large-scale deep learning is a relatively new field. Before this direction becomes hot and attractive for many researchers in the machine learning community, people in the system community have built a fair number of platforms that aim to speed up Big Data computing. The pioneer in this field is the most ubiquitous MapReduce [9] framework, as in the Hadoop [33] software library. After that we have many distributed systems that are designed for various certain kinds of applications. DryadLINQ [40] and SCOPE [3] are designed for relational algebra. Pregel [23] and GraphLab [22] are presented for large-scale graph algorithms and analysis. MadLINQ [29] and Presto [37] aim at big matrix computation. For machine learning applications, we have Spark [41] which targets at iterative in-memory algorithms, and Dandelion [31] which is designed for machine learning algorithms across heterogeneous hardware clusters. Despite their success for their target applications, it is somehow disappointing that, these systems are not widely welcome among researchers in the community of deep learning.

A typical deep learning algorithm iterates over a series of matrix computations. Therefore the most natural way to express this logic is to use tools like Matlab or Octave. Due to their expressiveness and popularity, most advances regarding deep learning algorithms are accompanied with release of Matlab code. Packages like deep learning tools are not only for beginners, but also easily modified and manipulated by adept researchers.

Platforms such as python-based Theano [2] and Torch [7] sacrifice a certain amount of generality in order to gain specialized deep learning optimizations. These optimizations largely integrates with the underlying matrix computation libraries such as MKL or CUBLAS. The user programs are compiled into a series of matrix operations, calling into these libraries on CPU and GPU respectively.

The training tasks of speech, vision and natural language processing require greater speed-up and scale-out, since they need to deal with large amount of data from real world. In these cases, the model-oriented platforms like Caffe [17], ConvNet [19] (for vision tasks with convolutional neural network models), Word2Vec [24] or RNNLM [25] (for natural language processing with recurrent neural network models) are more welcome. Such platforms are often hand-tuned and deeply optimized in order to cooperate with the underlying hardware features such as cache hierarchies, multi-core CPUs and many-core GPUs. Despite that they are derived from general deep learning architectures, the major flexibility of their programs is mostly reconfigurability (e.g. the number of layers, the size of kernels, activation functions etc.). As task-specific tools, they are not expected to work well on other domains, or accommodate new algorithm innovations smoothly.

Googles DistBelief [8] is a system built from scratch to handle large-scale deep learning tasks. It runs over 16 thousand cores in a large CPU cluster. Both model parallelism (parallelism within one model) and data parallelism (parallelism across multiple coordinated models) can be expressed with the framework of DistBelief. However, details on the programming model, system design and implementation of DistBelief are not revealed to public. This becomes one of the key drives for the project of Minerva.

2.2 Deep Learning Background

In this section, we examine several modern deep neural network models. These models have been widely used by deep learning communities and have been proven to deliver good performance in applications such as speech/vision recognition and natural language processing.

2.2.1 Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) is the most common training procedure in practice. The spirit of SGD is simple. Assuming θ to be our hypothesis, D to be our training dataset, the SGD algorithm can be expressed as following:

```
Choose a initial hypothesis weights w and learning rate a.
Repeat until converge:
   For all data point d{i} in D:
      g = CalculateGradient(d{i}, w)
      w = w - ag
```

The principle of SGD is to evaluate the loss function of our hypothesis at each data point. In practice the training dataset D is usually partitioned into multiple mini-batches $\{D_1, D_2, \}$. The update procedure is applied after the gradient of each mini-batch is calculated. The number of data points that are included in one mini-batch is usually treated as a parameter in the configuration of a model. Such mini-batch mechanism provides compact matrix computation when calculating the gradient, leading to faster convergence.

The following code is an example of a 3-layer feedforward neural network which adopts SGD method in Matlab code. In this case, our hypothesis consists of two matrices W and V, along with their corresponding bias vector b and c. A mini-batch $input\{idx\}$ is a matrix which consists of a series of data samples column. The computation of the activation y at hidden layer $layer2$

is shown in line 16. The output z of layer $layer3$ takes $layer2$ as input, and produce the output for the whole model similarly. The loss function is the typical mean squared error between output z and the ground truth $label\{idx\}$. When backpropagating, ze and ye are the error derivatives with respect to $layer3$ and $layer2$ before applying sigmoid function. Their products with the input of $layer2$ and $layer1$ yield the weight update matrices, modulated by the learning rate factor.

```
% Setting Hyper-parameters
learning_rate = 0.1; numEpochs = 66; batchSize = 128;
% Initializing model parameters
W = 0.1 * randn(layer2, layer1)
V = 0.1 * randn(layer3, layer2)
b = 0.1 * randn(layer2, 1)
c = 0.1 * randn(layer3, 1)
%Loading training data
[inputs, labels] = LoadBatches(batchsize, ...);

%Learning Procedure
for epoch = 1 : numEpochs
    for idx = 1 : length(inputs)
        x = inputs{idx};
        % feedforward pass
        y = sigm(W * x + repmat(b, layer2, 1));
        z = sigm(W * y + repmat(c, layer3, 1));
        % backward pass
        ze = z .* (1 - z) .* (z - labels{idx});
        ye = y .* (1 - y) .* (W' * ze);
        % updating weight
```

```
        W = W - learning_rate * ye * x' / batchsize;
        V = V - learning_rate * ze * y' / batchsize;
        b = b - learning_rate * sum(ye, 2) / batchsize;
        c = c - learning_rate * sum(ze, 2) / batchsize;
    end
end
```

Most of the deep learning models follow this basic feedforward-backpropagation structure, but with much deeper and wider configurations. As we can see, to program this multi-layer neural network architecture with matrix-based APIs is simple. However, to scale it up is not so easy. One direction is to use highly optimized underlying libraries for massive matrix computations. These libraries are usually external to the productivity-oriented tools. Another way to achieve the scale-up is by using the idea of Asynchronous Stochastic Gradient Descent (ASGD), such as HOGWILD [30] algorithm which parallelizes the traditional SGD algorithm without any locking. The idea of a ASGD algorithm typically involves partitioning of the data, identical execution of each individual process, and maintaining of a parameter server.

2.2.2 Convolutional Neural Networks (CNN)

CNN, inroduced by Kunihiko Fukushima [12] and improved by Yann LeCun [21], is the leading neural network architecture for image recognition tasks. Apart from the traditional image recognition field, CNN has recently demonstrated its power in other fields such as artificial intelligence [6]. Unlike multi-layer feedforward neural network, CNN adopts a different style when doing the forward pass, as shown in Figure 2-1. One convolutional layer uses a set of input images, each with a filter (sometimes called kernel). Each filter looks at every small portion of its corresponding input image, called its receptive field, in order to detect good representations of the input. The

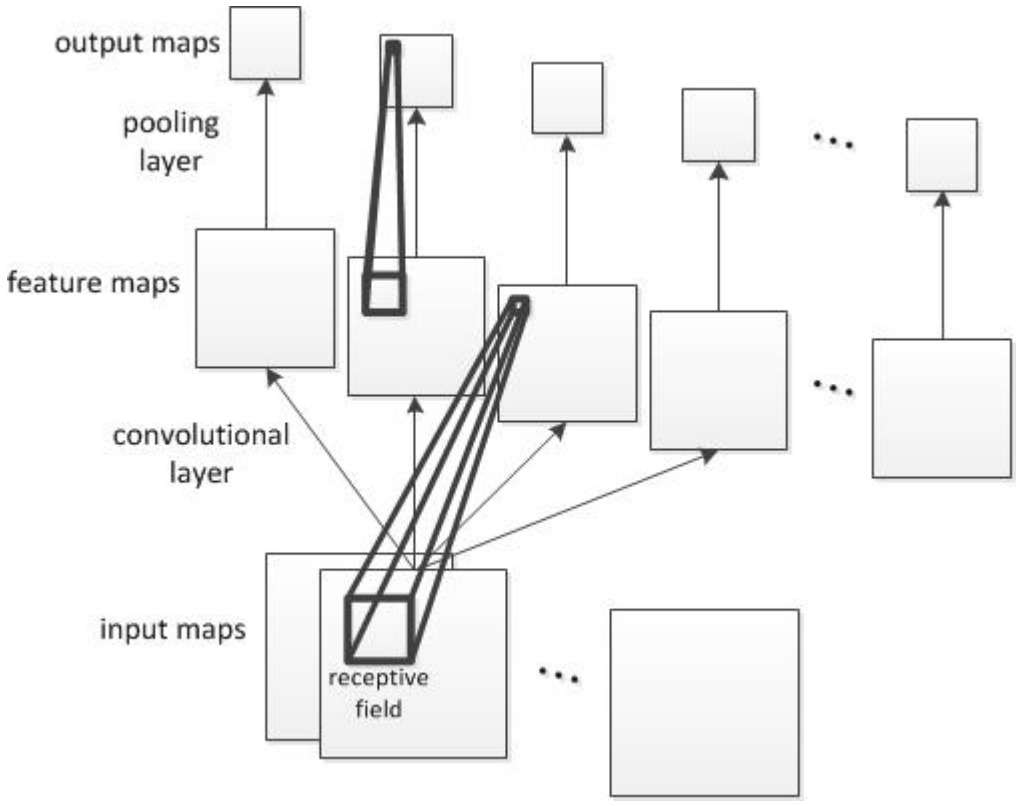

Figure 2–1: Convolutional Neural Network (CNN)

filters convolve against every input image respectively, generating a stack of feature maps. This procedure is usually followed by non-linear activations and poolings. Relu activation is used for most of CNN architectures. The pooling layers compute the maximum or average of feature maps over a small region, combine the output and reduce the variance. These three typical operations replace the typical straightforward matrix multiplications in feedforward neural networks.

The number of parameters, which is the size of filters, is much smaller than a typical fully-connected weight matrix. For example, ConvNet [20] has 96 11×11 filters at the first layer. This doesn't mean that training CNN can be much faster. The input has 3 RGB channels of raw image with 224×224 pixels each. With a 4×4 stride, the output of this convolutional layer contains 96 feature maps with 56×56 pixels in each of it. Since the size of feature maps

decreases with the depth of layers, layers close to the output layer tend to have more feature maps to preserve the information carried along from the input images, and balance the computation load. Therefore the number of pixels in hidden layers can be large. Another reason for the computation to be slow is that the addresses of these pixels in memory is not continuous for matrix computation. Implementation of Caffe [17] includes one-to-one memory copy on pixel level, which is a non-neglectable cost.

This structure of CNN is very effective for learning patterns in natural images. However implementing CNN correctly is difficult. Therefore people in deep learning community tend to use existing tools such as Caffe [17] or cudnn [4] instead of implementing one on their own.

2.2.3 Recurrent Neural Network (RNN)

RNN is another important variation of deep learning models. The major distinction between RNN and classical neural networks is that RNN has internal cycles. Figure 2-2 shows a typical RNN example, where hidden units have self-loop connections. This topology brings the concept of time series into the training of this model. The output of the hidden layer at time step t will be the input for the same hidden layer at time step $t+1$. As such, RNN incorporates temporal knowledge and resolves long-range dependencies. The following equations describe the forward pass of training of RNN:

$h_t = g(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$

$y_t = g(W_{xh}h_t + b_z)$

g is an element-wise non-linear activation function (e.g. a sigmoid function), x_t is the input, h_t is the hidden state. y_t is the output at time step t, produced by the same way as multi-layer feedforward network.

Though the structure looks shallow, with temporal knowledge the learning procedure is deep. One way to see this topology is that we unroll the network

over time, as shown in Figure 2-2(a)(b). The training procedure is then similar to classical feedforward network. This procedure is called back-propagation through time (BPTT). BPTT typically goes back in a limited time steps.

2.2.4 Long Short Term Memory (LSTM)

LSTM is a variation of RNN architecture. It was proposed by Sepp Hochreiter and Jurgen Schmidhuber [15] in 1997. Recent breakthrough on image description task [11, 38] and translation task [5, 35] make this architecture impressive to the public.

There are several weaknesses of the traditional RNN. The major one is that the error gradients vanish or explode exponentially with the number of time steps when doing BPTT, typically referred as an "error carousal". There are two typical ways to address this issue within the old model. The first one is to limit the time lags of the BPTT procedure, which sacrifices the ability of the network to learn long term data. The second one is to force the error flow to be constant by setting parameters. This approach enables a LSTM network to cooperate with infinite time lags, but introduces many weight conflicts which make learning difficult. The major difference between LSTM and traditional RNN is that LSTM provides a gate mechanism that allows the network to achieve the following properties:

1. ability to avoid the exponentially error gradients vanishing and exploding problem;
2. ability to learn when to forget the previous hidden states and when to update the current hidden state with new gradients.

Figure 2-3 shows the architecture of a LSTM memory block. The input at time step t is composed by two parts, the data at time step t and the output at the previous time step $t-1$, just like the unrolling procedure of the traditional RNN. There are three modulating gates that control one LSTM memory block,

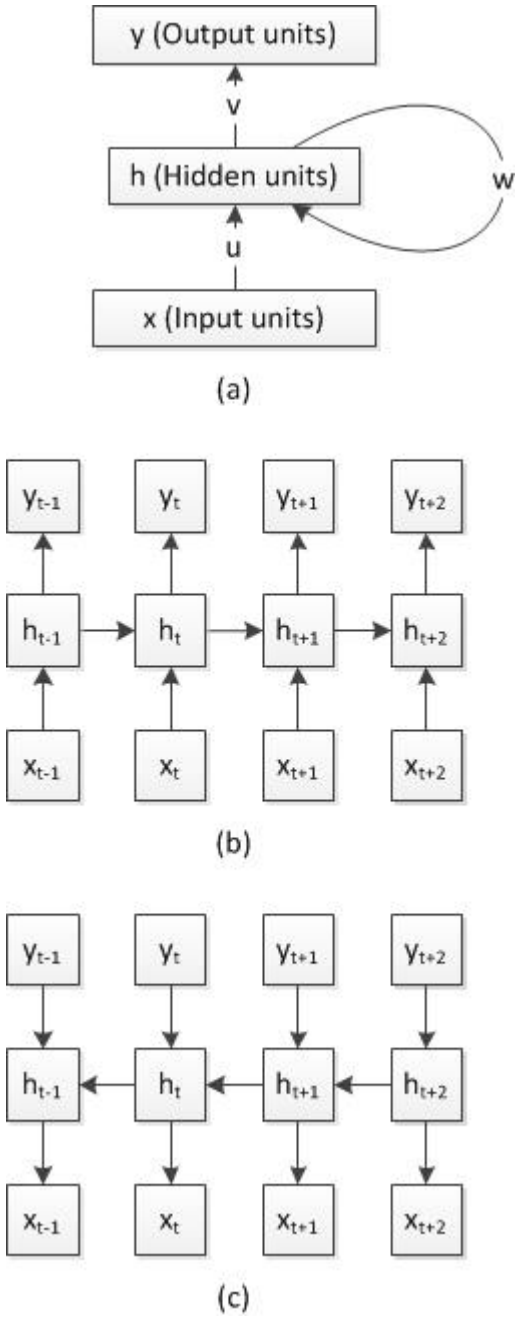

Figure 2–2: Recurrent Neural Network (RNN). (a) the simplest RNN topology; (b) the forward pass after unrolling the network; (c) the backward pass after unrolling the network

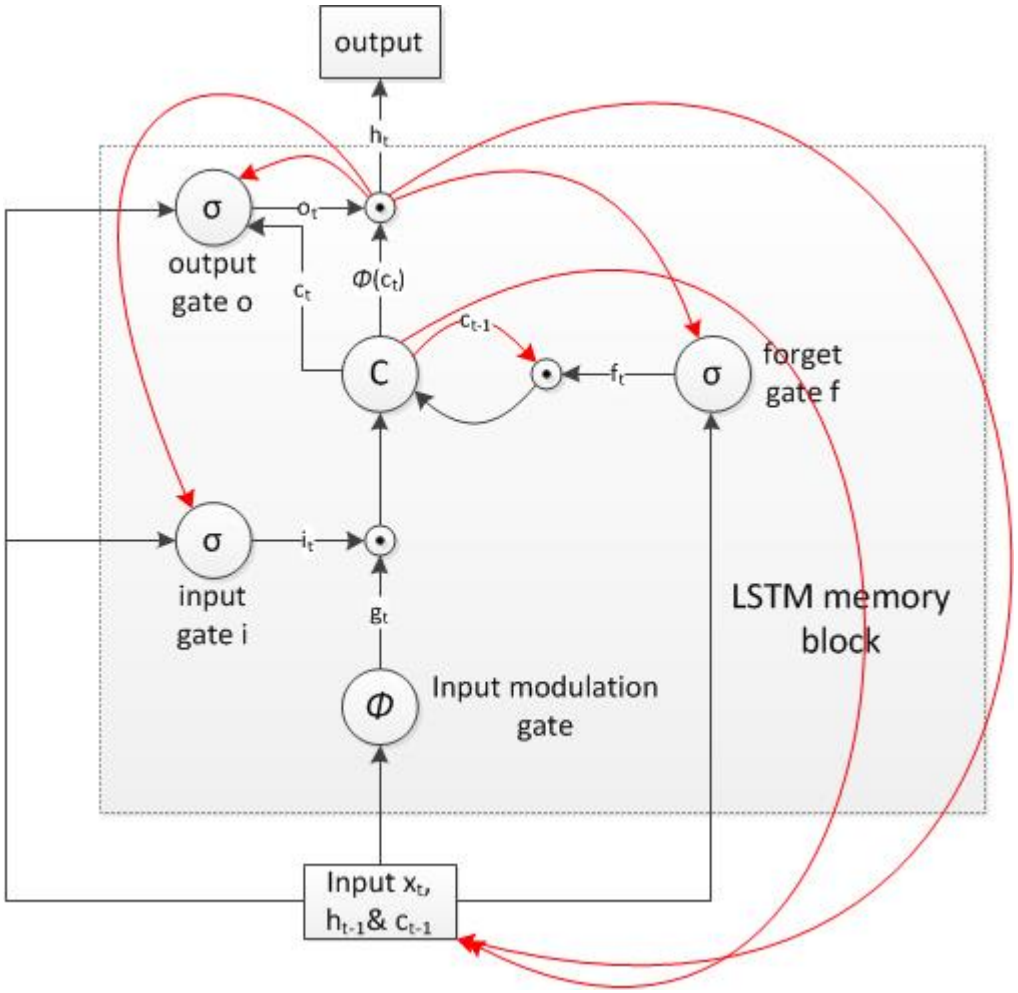

Figure 2–3: An Long Short Term Memory (LSTM) memory cell

the input gate i_t, the output gate o_t and the forget gate f_t, each with its own weight matrix. The modulating factor of each gate is generated by the input and its own weight matrix, normalized to $(0, 1)$ by the sigmoid function. At time step t, the memory cell c_t at time step t is composed by its previous state c_{t-1} at time step $t-1$, modulated by the forget gate f_t, and a function g_t of the current input and previous hidden state, modulated by the input gate i_t. The output of this block is modulated by the output gate o_t. The equations that express the forward procedure of the memory cell are shown as following:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + c_{t-1} \odot W_{xc} + b_i)$$

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + c_{t-1} \odot W_{xf} + b_f)$$

$$g_t = \phi(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + c_t \odot W_{xo} + b_o)$$

$$h_t = o_t \odot \phi(c_t)$$

Here $\phi(x) = 2\sigma(2x) - 1$. Each gate has its own weight matrix (W_{xi}, W_{xf}, W_{xo}). The input gate enables a LSTM network to learn whether it should accept the receiving input. The output gate enables a LSTM network to learn whether it should output its result. When doing BPTT, the errors may be cut off by these gates so that the property 1 can be achieved by regular back-propagation. Therefore the disadvantages of the naive approach using by traditional RNN are avoided. These additional gates enable the LSTM to learn long-term dynamics, achieving learning ability that traditional RNN is not capable of.

CHAPTER 3
System overview

(The main body of Minerva system is a group work by my teammates Minjie, Yutian and Chuntao. I am presenting details about the underlying system from Chapter 3 to Chapter 5, because they are needed to understand how to optimize user programs and where potential bottlenecks could be.)

3.1 Finding Parallelism

A deep learning system takes advantages of parallelism on two levels: model parallelism, and data parallelism (DistBlief [8]). For model parallelism, a single model can be trained by multiple cooperating processes simultaneously, with each taking care of a fraction of the entire model. The output of the parallel training procedure must be the same as if the entire model is trained by just one process. Our approach to achieve model parallelism is by transforming the user-defined model into a dataflow representation, a directed acyclic graph. This graph is then partitioned by the runtime. A process will be called to execute one of the partitions of the graph, while resolving data dependencies and communications in a distributed way. The execution is asynchronous, which means that there is no central coordination operation among all processes. Each process is called via asynchronous remote procedure call, which overlaps the communication overhead maximally.

Data parallelism is a coarse level of parallelism comparing to model parallelism. In Minerva, users may define several parameter servers, each associated to one machine. The input data set is then partitioned into several data shards, and distributed across all parameter servers, achieving parallel executions. Each parameter server updates its local weights respectively

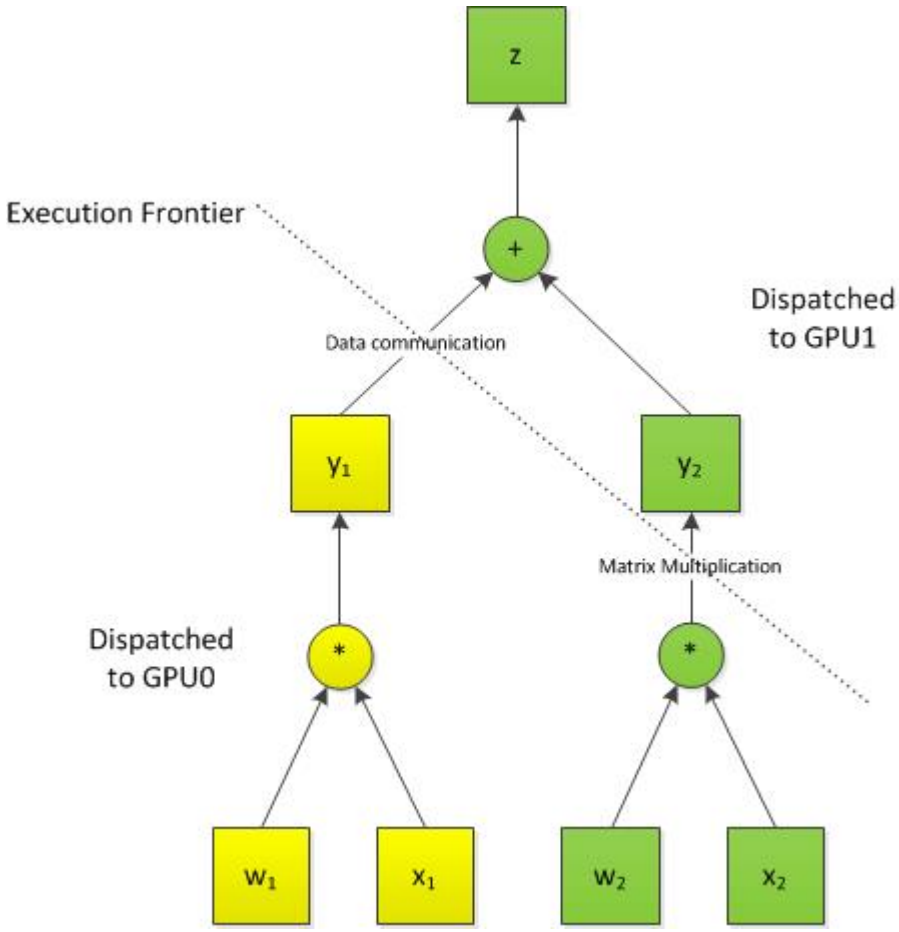

Figure 3–1: Consider a DAG as above. The yellow vertices are associated to GPU0 and the green ones are associated to GPU1. Assume the execution frontier at some moment is the dash line. The data communication and the matrix multiplication on the frontier can be proceeded simultaneously. This can be done by the scheduling of a CPU-GPU architecture.

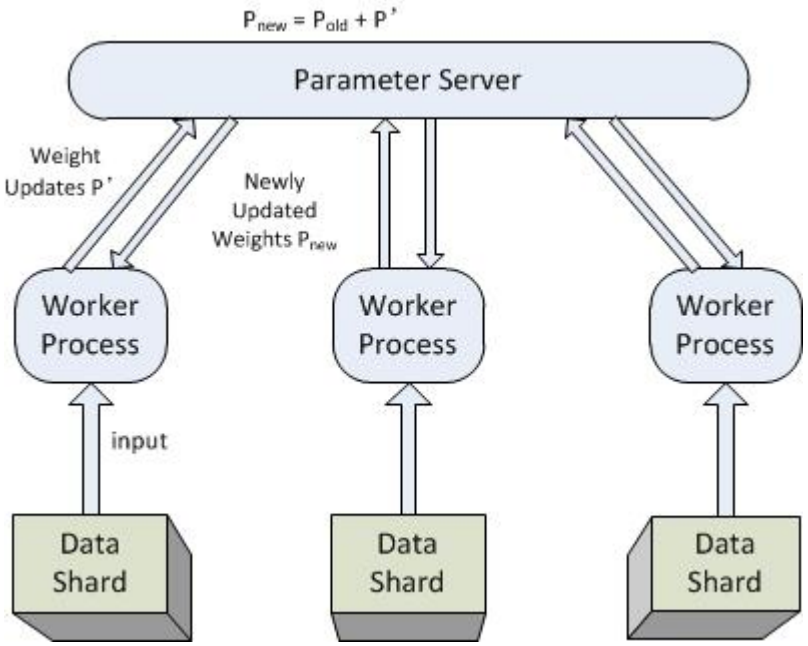

Figure 3–2: Asynchronous Stochastic Gradient Descent (ASGD)

and asynchronously. The weights are exchanged among all parameter servers periodically. This scheme can be regarded as a decentralized variation of Asynchronous Stochastic Gradient Descent (ASGD) algorithm. Users may also implement the exact ASGD scheme (Figure 3-2), which has a centralized parameter server. In ASGD, The training procedure is executed among several worker machines with the same data partition strategy. Each worker process exchanges its own weight updates with the parameter server and gets the latest updated weights respectively. Essentially, Minerva has no domain knowledge of these user-defined data, giving flexibility for the users to design their own scheme.

3.2 Overall Architecture

Figure 3-3 illustrates the overall architecture of Minerva system. The system execution is logically divided into two phases. The first phase converts the frontend user program into a dataflow representation, which is a logical Directed Acyclic Graph (DAG). The graph generated will be referenced for data

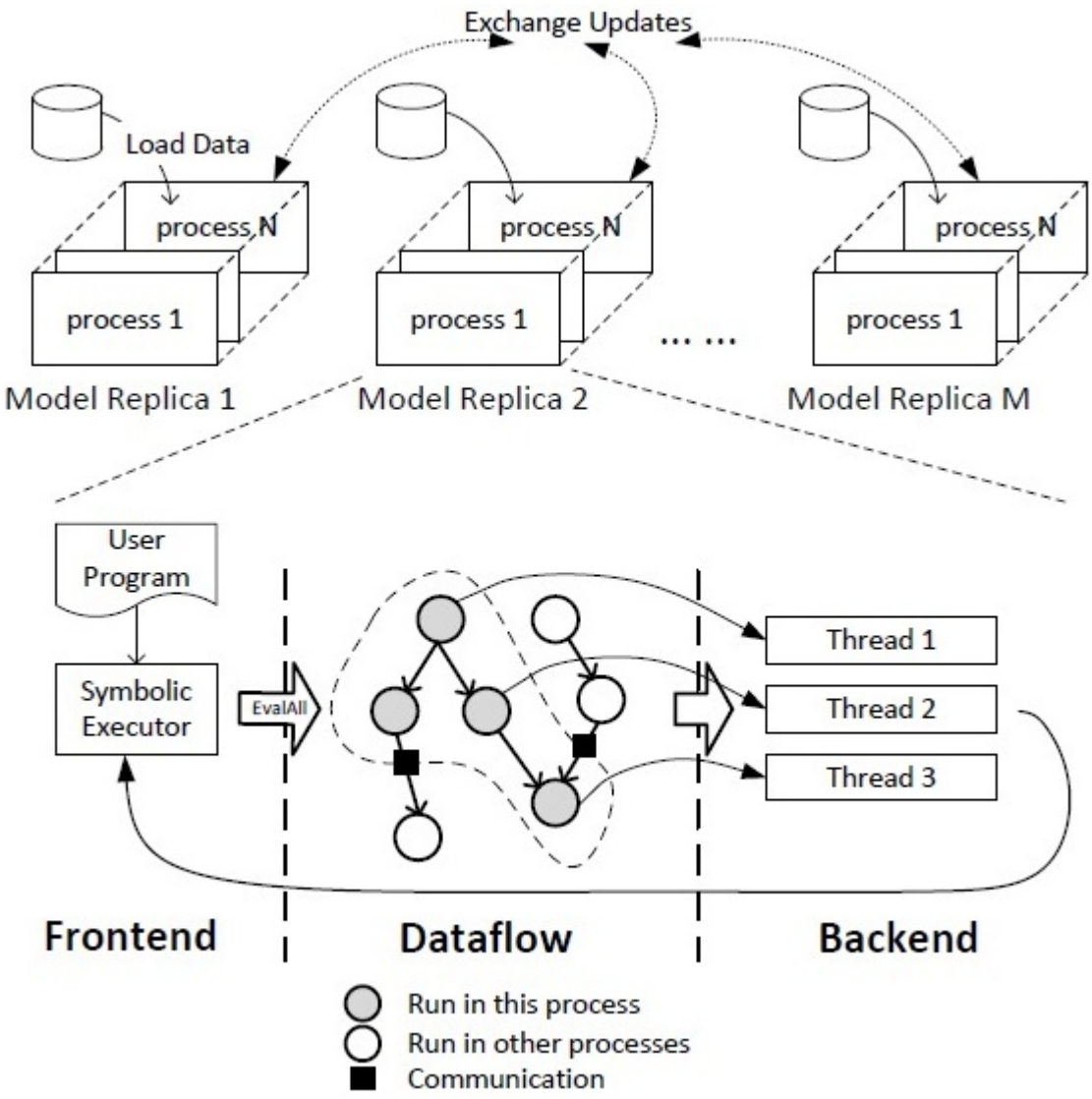

Figure 3–3: The overall architecture of Minerva system [39]

placement decision and further parallel execution plan. This phase is hardware independent. Therefore it separates the flexibility of Minerva's programming APIs and the underlying support of heterogeneous hardware environments. The overhead to generate this dataflow graphical representation is negligible because its scale is far more incommensurate to the actual data execution.

The second phase evaluates the logical DAG generated by the first phase concretely. A calling for the actual data to be printed or by other use external to Minerva system will trigger this phase. Each process takes a portion of

the graph, which is a induced graph generated by a subset of vertices. With data parallelism is handled by the first phase, the second phase focuses on the performance improvements on model parallelism. Multi-core CPUs and GPU acceleration are the major two advantages provided by hardware environments. To exploit GPU acceleration is straightforward, the process just executes the op vertex if the corresponding GPU implementation is available and there is sufficient resource for this execution. The details of exploiting the resources of Multi-core CPUs will be discussed in following chapters.

CHAPTER 4
Programming Model

A fair alternative to express the model is by adopting a graph-style representation. After all a neural network is essentially a graph model, and we actually tried this direction before. Our first attempt used the underlying gather/scatter/apply API in GraphLab [22] to build a deep learning architecture. The benefit of this approach is that the parallelism inherent inside this graph-style model is directly handled by the underlying system. However the lack of the programmability becomes a severe drawback. We have to break the entire training procedure into isolated vertex programs. Simply programming and debugging a compact matrix computation takes a great deal of efforts. This drawback becomes the major reason for us to give up this approach.

One can achieve more fined-grained model parallelism by partitioning the matrix. As it turns out, on a multi-core CPU machine, Intel's MKL already leverages multi-threaded implementation on the multiple cores. This provides another opportunity to exploit the model parallelism and boost the performance.

Despite the hardness of its programmability when dealing with neural networks, the graph-based approach is still valuable for its strength in expressing the natural parallelism. Minerva system combines the common matrix-based programming interfaces with the underlying graph representation. This can be summarized as "program with matrices, build as a graph".

4.1 Minerva API

(The python-API layer of Minerva is implemented by Yutian and me. Yutian implemented the CNN APIs and the Select-operations for LSTM programming.)

Minerva provides matrix-based APIs (Table 4-1). Users can easily adopt matlab coding style to define the model and training procedure. The user program is coded in python, and the underlying runtime system is implemented using c++. We use Boost Python library [1] to make the frontend user code in python cooperate with the underlying system code in c++.

The basic data type in deep learning is defined in matrix. A matrix in deep learning algorithm may be a hyper-matrix, which means that the dimension of a matrix may exceed 2 (e.g. the basic data type *blob* in Caffe [17] has dimension of 4). Users can define a matrix with arbitrary dimension by initializing the values of its entries with constant or random variables as following:

```
W = owl.zeros([10, 10], 0.0, 0.1)
B = owl.bias([10, 1])
```

where W is a 10 by 10 matrix filled with entries by Gaussian distribution with mean 0.0 and standard deviation 0.1, B is a 10 by 1 matrix with all zero values. *owl* is the name of Boost.Python module created by Minerva system. The Minerva APIs are all defined in this module, which needs to be imported on the head of a Minerva user program.

Minerva doesn't provide specific I/O interfaces. Instead, we provide interfaces that enable transformation from a NumPy [28] array to the basic array type in Minerva system and vice versa. We encourage users to directly use SciPy [18] packages to do I/O operations, since SciPy packages provide convenient interfaces for users to read data from a mat-file, which is one of the

Table 4–1: The Minerva APIs

Catagory	APIs
Matrix Creators	A = zeros(scale)
	A = ones(scale)
	A = randn(scale, mean, var)
	A = randb(scale, p)
Matrix Multiplication	A = B * C
	A *= B
Matrix Layout	A = B.trans()
	A = B.reshape(scale)
Element-wise Operators (multi-variate)	A = B op C
	A += B
	A -= B
	A /= B
Element-wise Operators (unary)	A = B op v
	A = v op B
	A += v
	A -= v
	A /= v
Reductions (unary)	A = B.sum()
	A = B.sum(dim)
	A = B.sum(scale)
	A = B.max()
	A = B.max(dim)
	A = B.max(scale)
	A = B.argmax(dim)
	A = B.count_zero()
Element-wise Functions	A = mult(B, C)
	A = sigmoid(B)
	A = exp(B)
	A = ln(B)
	A = relu(B)
	A = tanh(B)
Interface with NumPy	NArray = from_numpy(ndarray)
	ndarray = to_numpy(NArray)

most common file formats in machine learning, defined according to Matlab convention.

In Minerva, a matrix either has to be directly initialized from the model, or has to be a result of computations. Such computations could be matrix multiplications or Element-wise operations. For example, the activation of a hidden layer is usually given by $y = owl.sigmoid(W*x+b)$, where $owl.sigmoid$ is an element-wise function that produces sigmoid function for every entry in the input matrix.

Notice that the second dimension of matrix x is the size of one mini-batch, but the corresponding dimension of bias matrix b is just 1 because b is actually a vector. We need to replicate the bias vector b so that the resulting matrix has the same dimension of $W * x$. In Matlab this is handled by a function, called *repmat*, which returns tiling of a matrix. However we notice that this is a very common type of operations. Therefore we don't create equivalent function to *repmat* in Matlab. When doing element-wise adding between two matrices, Minerva runtime system automatically detects the dimension inequality between the two operands, and replicate the smaller matrix so that their dimensions are equal to each other.

Convolutional neural networks introduce some other operations which are not by matrix convention. There are four basic types of operations: convolution, pooling, activation and normalization. Only activation operations among the four are element-wise so that we can adopt typical matrix operations. However the key operations belong to first two types. People feel reluctant to implement these two types of operations due to the difficulties in programming and debugging. Minerva system addresses all three convolutional operations by using similar interfaces as cudnn [4] (shown in Table 4-2).

Table 4–2: The Minerva APIs for CNN operations

Catagory	APIs
Convolution	top = conv_forward(bottom, filter, bias, conv_info) error = conv_backward_data(diff, bottom, filter, conv_info) filter_update = conv_backward_filter(diff, bottom, filter, conv_info) bias_update = conv_backward_bias(diff)
Activation	top = activation_forward(bottom, algo) top = softmax_forward(bottom, algo) error = activation_backward(diff, top, bottom, algo) error = softmax_backward(diff, top, algo)
Pooling	top = pooling_forward(bottom, pool_info) error = pooling_backward(diff, top, bottom, pool_info)
Normalization	top = lrn_forward(bottom, scale, local_size, alpha, beta) error = lrn_backward(bottom, top, scale, diff, local_size, alpha, beta)

The *bottom* and *top* variables are the matrix parameters for the corresponding convolutional functions. They inherit from Minerva's basic matrix data type. The only constraint is that the matrices involved in convolutional computations are required to be 4-dimension. The $conv\_info$ and $pool\_info$ are sets of configuration information (e.g. the size of kernel, padding and stride) that is required for convolutional and pooling functions.

To support batch training of Long Short Term Memory (see section 6.2.2), we add matrix indexing APIs as well. For instance, $A = B.Select(vectorIdx)$ can be easily used to fetch a batch from the embedding table. The matrix A is composed of columns from matrix B, indexed by vector Idx. To support convenient updating of the embedding table, we designed $SelectiveSub$, which is somewhat an inverse operation of $Select$ plus a matrix substraction.

4.2 Expressing Parallelism

The first phase of Minerva system execution, mentioned in Section 3.2, takes the user program as its input and transform it into a dataflow representation. For example, consider the following clause $y = owl.sigmoid(W * x + b)$. The corresponding DAG is shown in Figure 4-1.

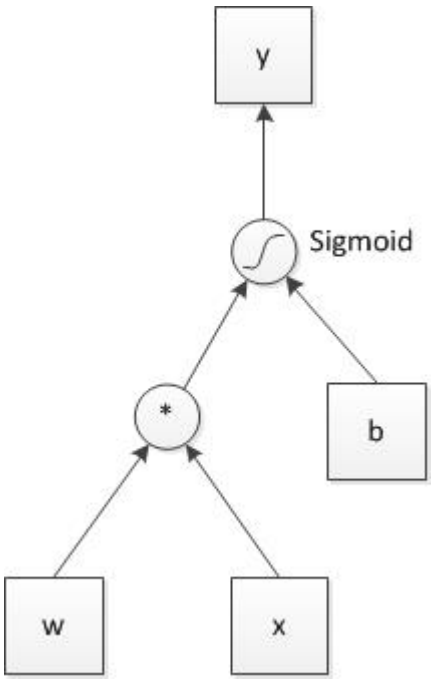

Figure 4–1: The DAG representation for $y = owl.sigmoid(W * x + b)$

The vertices of the dataflow DAG are distributed across several hardware environments to run the actual execution according to user configurations. Such information is associated to each vertex in DAG. Minerva system has portable APIs for users to define such configurations. Users may manipulate the distribution via a concept which we called "virtual device". A "virtual device" is typically corresponding to a single GPU or a CPU, which can be defined using following clauses:

$cpu\_device = owl.create\_cpu\_device()$

$gpu\_device = owl.creat\_gpu\_device(gpu\_id)$

The first clause defines a "virtual device" that consists of a GPU. The second clause defines a "virtual device" that consists of a GPU with device id gpu_id (the same device id when calling $cudaSetDevice(gpu\_id)$). The distribution of the user code is then simply accomplished by calling the following clauses:

$owl.set\_device(cpu\_device)$ or $owl.set\_device(gpu\_device)$

These clauses indicate that the user program coming after these clauses will be eventually executed in hardware associated to them until another *owl.set_device* is called or the end of the user program. The device info set by *owl.set_device* will be accompanied to each vertex in the dataflow DAG.

CHAPTER 5
System Design

The two-phase execution, as shown in Figure 3-2, is identical in every Minerva process. Minerva system first builds an internal logical DAG representation of the model defined by symbolic execution of the frontend program. This phase is single-threaded. It stops when a $Eval()$ statement is hit. User may call this statement explicitly, or Minerva system will eventually call this statement automatically when the data is required immediately for external use (e.g. I/O use). The backend execution procedure of the DAG starts from the frontier of the graph and goes along with a downstream way. The frontier data points are assumed to be ready at the beginning of this phase. When all of the vertices are processed, this second phase is then terminated. When the second phase is thoroughly processed, Minerva system will return to symbolic processing of the user code.

5.1 The Dataflow Representation

The dataflow graph has two types of vertices, computing vertex and data vertex. A computing vertex represents an operation, typically a matrix operation (e.g. matrix multiplication, element-wise addition, etc.). A data vertex represents an operand. Their dependency relations are represented by the connections between them. The ingoing connections to a computing vertex indicate the inputs of the corresponding operation. The result of the a computing vertex is associated to the data vertex at the opposite side of an outgoing connection.

5.2 Runtime Design and Implementation

When the second phase is triggered, which means when the data vertex is actually called to be evaluated, the statuses of all vertices related to the resulting data vertex are marked as ready. The frontier of the DAG is a set of vertices that has no incoming edges. The data associated to these vertices is fed by initialization or external I/O interfaces. Minerva system adopts a push style to schedule the execution of all ready vertices, starting from the frontier.

Each vertex is associate to corresponding hardware information, the "virtual device" which it will be dispatched. Each device has its own scheduler to handle multi-thread or multi-stream execution. The corresponding task of each computing vertex will be pushed to its designated device. The ownerships of data vertices are determined by the location of its neighbouring computing vertices. The system automatically handles data communication among CPU and GPU in a distributed and lazy way. Data transmission is only performed when the scheduler finds required data missing locally. In this case, there will be a blocking call waiting for the corresponding data. The scheduler then takes all computing vertices that have their inputs ready, allocates hardware resources such as memory and threads for the tasks. A thread pool will assign threads that will actually carry the computation. On the other hand, the output data generated by a computing task stays locally (the main memory or GPU memory). If the data is demanded by other devices it will be appended with an RPC call which will send the data. In order to overlap the communications and the computations as much as possible, the computing vertices which have remote posterior vertices have higher priorities than others. When all the posterior computing vertices of a data vertex are called, the associated data is then garbage collected. As we can see, this entire procedure needs no central coordination and is totally distributed.

Minerva system supports heterogeneous hardware. When the input data is ready, the actual computing function is called according to the respective underlying hardware environment. All of Minerva APIs regarding matrix computations have available GPU implementations. For most of matrix computations, Minerva system uses Intel's MKL Library [16] and Nvidia's CUBLAS Library [27] for running on multi-core CPUs and GPUs respectively. For CNN operations, Minerva system uses cudnn [4] Libraries which implement complex CNN operations efficiently on GPU.

5.3 Optimizations

An important optimization strategy of Minerva system is to hide the I/O latency and the computing time. Minerva system adopts a lazy style to schedule the tasks. Most of clauses in the user program merely define the topology. For these clauses, the actual computations still wait for scheduling and evaluating when they are called by the runtime. On the other hand, Minerva interfaces integrate with the external I/O tools such as SciPy, therefore the I/O execution is actually carried on when they are called. The computation time and I/O cost can be overlapped if they are data independent. Minerva achieves the overlapping by defining asynchronous evaluation primitives:

$m.start\_eval()$: a non-blocking call to start the evaluation phase of matrix m.

$m.wait\_for\_eval()$: a blocking call to get the actual value of matrix m. The value of m is computed when this call returns.

When the $start\_eval()$ method is called, Minerva system will finish the first phase of the DAG representation, then start a new thread to handle the second phase in runtime concurrently. The main process will move on and handle the coming I/O operations. The system will automatically block the execution when the corresponding data is called by other tasks.

Convolution operations deserve some special treatments due to its special inherent parallelism. Notice that the forward computations among all feature maps in a same layer are independent to each other. Parallelism of forward convolutional operations can proceed by partitioning feature maps and convolve them concurrently. The exact grouping trick is also used in Caffe [17], where the numbers of groups are set by the user in the configuration file. Such mechanism leverages load-balance issue when performing convolutional computations.

CHAPTER 6
Application Implementation

We have implemented several common deep learning networks based on a variety of deep learning research tasks with Minerva system. Our experiences of implementing these deep learning networks also provide us an additional guide of some critical utilities that Minerva system should support. For example, we add the script ability to run a Minerva system out of a configuration file. This is a useful tool when running experiments with many different parameter configurations. The support for this utility is also suggested by some advanced machine learning researchers. Deep learning programs can be difficult to implement correctly. Therefore we add a built-in gradient checker which helps detect the potential bugs.

The matrix-based Minerva programs can succinctly express various deep learning algorithms ranging from speech, natural language processing to computer vision, leading to programs as compact as Matlab versions. The most prominent advantage is that Minerva system is adept at developing algorithm innovations and advanced optimizations. The programmability doesn't sacrifice the performance. As we will show in the following, Minerva system renders scalable, comparable or better performance than task-specific implementations.

6.1 Convolutional Neural Network

Recent development suggests that CNN can also be in arsenals for application fields other than computer vision. The performance of CNN can sometimes be improved by simply increasing the scale of the CNN topology [6]. The obstacles of doing so come from both hardware limitations and

software constraints. Training a network which exhausts the power of CNN by increasing the scale of the topology could take days or even weeks. The computation-intensive property of CNN tasks makes it highly suitable for Minerva system to execute.

6.1.1 Implementation

We present the details of programming Alex Krizhevsky's model [20], a very successful model which won prize of ImageNet 1K classification competition, using Minerva. There are 5 convolutional layers, each with a convolution, local-normalization sub-layers. The input layer contains 3 channels of 227-by-227 RGB images, randomly cropped from a original 256-by-256 raw image. It is then followed by a convolution sub-layer, a local normalization layer and a max-pooling sub-layer with a stride of 2, generating 96 56-by-56 feature maps. The second convolutional layer has a convolution layer with stride of 1 and padding of 2, and a pooling layer which has the same configuration as the previous one. This layer generates 256 feature maps. The third layer and the fourth layer doesn't contain pooling layers. The numbers of output feature maps are both 384 for these two layers. The last convolutional layer contains a max-pooling layer with the same configuration as the first two layers, generating 256 feature maps, each with size of 6-by-6. The filter size varies from 11-by-11 (the first layer) to 3-by-3 (the last three layers). With the dimension of weights defined ahead, we can define the feed-forward pass of this part with only 9 lines of Minerva code.

```
acts[0] = data # input
acts[1] = ele.relu(conv_forward(acts[0], model.weights[0],
    model.bias[0], model.conv_infos[0])) # conv1
acts[2] = pooling_forward(acts[1], model.pooling_infos[0]) #
    pool1
```

```
acts[3] = ele.relu(conv_forward(acts[2], model.weights[1],
    model.bias[1], model.conv_infos[1])) # conv2
acts[4] = pooling_forward(acts[3], model.pooling_infos[1]) #
    pool2
acts[5] = ele.relu(conv_forward(acts[4], model.weights[2],
    model.bias[2], model.conv_infos[2])) # conv3
acts[6] = ele.relu(conv_forward(acts[5], model.weights[3],
    model.bias[3], model.conv_infos[3])) # conv4
acts[7] = ele.relu(conv_forward(acts[6], model.weights[4],
    model.bias[4], model.conv_infos[4])) # conv5
acts[8] = pooling_forward(acts[7], model.pooling_infos[2]) #
    pool5
```

The last convolutional layer is immediately followed by three fully-connected layers, each with a weight size of 4096-by-9216, 4096-by-4096 and 1000-by-4096, respectively. This structure has 60 million free parameters in total.

```
re_acts8 = acts[8].reshape([np.prod(acts[8].shape[0:3]),
    num_samples]) # reshape the conv layer into a 1−dimension layer
acts[9] = ele.relu(model.weights[5] * re_acts8 +
    model.bias[5]) # fc6
mask6 = owl.randb(acts[9].shape, dropout_rate)
acts[9] = ele.mult(acts[9], mask6) # drop6
acts[10] = ele.relu(model.weights[6] * acts[9] +
    model.bias[6]) # fc7
mask7 = owl.randb(acts[10].shape, dropout_rate)
acts[10] = ele.mult(acts[10], mask7) # drop7
acts[11] = model.weights[7] * acts[10] + model.bias[7] # fc8
```

```
out = softmax_forward(acts[11], soft_op.instance) # prob
```

6.1.2 Multi-GPU Training

To distribute the computation, we divide each min-batch into several portions of equal size. These portions are fed to each device so that each device has the same amount of workload. This can be easily manipulated by using the device APIs at each training iteration:

```
owl.set_device(gpu0)
out1 = train_one_mb(model, data1, label1, weightsgrad1,
   biasgrad1, dropout_rate)
owl.set_device(gpu1)
out2 = train_one_mb(model, data2, label2, weightsgrad2,
   biasgrad2, dropout_rate)
```

The *train_one_mb* method contains both forward and backward pass. As we can see, the two *train_one_mb* functions are executed against different GPUs. They use the same *model* class and *dropout_rate* so that the configurations of both executions are the same. Each GPU functions as a parameter server. The data passed to the two functions are totally separated so that the dependencies during execution are guaranteed. The weight updates come right after the second *train_one_mb* call, as well as momentum. In this case the *gpu*1 device will carry on the updating and the system will automatically handle the data copy.

One trick to improve the parallelism is to increase the scale of the DAG between two evaluations. This trick provides more freedom for the scheduler to exploit the model parallelism. This can be done by calling the evaluation method periodically after a certain number of iterations. For example one can

write the following code so that the backend evaluation is carried on after each ten iterations.

```
for (samples, labels) in get_train_mb(minibatch_size):
    count = count + 1
    # begin of training & updating
    ...
    # end of training & updating
    if count % 10 == 0:
        print_training_accuracy(out, label)
```

The *print_training_accuracy* method will get the actual value of the matrix *out* by calling its evaluation function. The DAG evaluation is triggered only when this statement is hit. The caveat is that the GPU memory limit could be exceeded due to large DAG.

6.2 Long Short Term Memory (LSTM)

Long Short Term Memory is an important variation of recurrent neural networks. Recent breakthrough on image "show and tell" tasks [11, 38] have demonstrated the learning power of this structure. The implementations for this particular architecture have come out one after another. The based environment varies from productivity-oriented tools such as python-numpy to task-specific tools such as theano and RNNlib. In this section we introduce the implementation of LSTM using Minerva, and our current progress in optimizing the LSTM training based on this implementation.

6.2.1 Implementation

The coding style of Minerva is as succinct as a python-numpy version. The model of our implementation contains one input layer, one hidden layer and one output layer, as every other existing implementation does. Although the

gate matrices are separate, the behaviours of these gates are similar. Current implementations often merge these matrices into one large weight matrix. The following c++ code shows one feed-forward pass for one training sample.

```
for (int t = 1; t < sent.size(); ++ t)
{
    data[t] = emb_weight[sent[t - 1]];

    // compute input gate
    act_ig[t] = ig_weight_data * data[t] + ig_weight_prev *
       Hout[t - 1] + Elewise::Mult(C[t - 1], ig_weight_cell) +
       ig_weight_bias;
    act_ig[t] = Elewise::SigmoidForward(act_ig[t]);

    // compute forget gate
    act_fg[t] = fg_weight_data * data[t] + fg_weight_prev *
       Hout[t - 1] + Elewise::Mult(C[t - 1], fg_weight_cell) +
       fg_weight_bias;
    act_fg[t] = Elewise::SigmoidForward(act_fg[t]);

    // compute feed forward activation
    act_ff[t] = ff_weight_data * data[t] + ff_weight_prev *
       Hout[t - 1] + ff_weight_bias;
    act_ff[t] = Elewise::TanhForward(act_ff[t]);

    // compute cell
    C[t] = Elewise::Mult(act_ig[t], act_ff[t]) +
       Elewise::Mult(C[t - 1], act_fg[t]);
```

```
    // compute output gate
    act_og[t] = og_weight_data * data[t] + og_weight_prev *
       Hout[t - 1] + Elewise::Mult(C[t], og_weight_cell) +
       og_weight_bias;
    act_og[t] = Elewise::SigmoidForward(act_og[t]);

    Hout[t] = Elewise::Mult(Elewise::TanhForward(C[t]),
       act_og[t]);

    NArray Y = Softmax(decoder_weights * Hout[t] +
       decoder_bias);
}
```

The time step t goes along with the length of a training sample. As we can see, the DAG generated at one time step of LSTM is more complicated than the DAG of one feed-forward pass of AlexNet (Section 6.1.1). This unrolling procedure results in a very deep chain, while the granularity of matrix operations is relatively light. The intermediate activations, cells and hidden outputs are stored to be used in back-propagation.

6.2.2 Batch Training

One way to accelerate the training procedure is to batch the training samples. This procedure, usually adopted in CNN training, groups separated per-sample training procedures into a large compact training procedure. It greatly increases the granularity of a single operation, providing potentials to exploit the GPU computation power and boost the performance.

There is a tricky issue when we group a bunch of samples into one batch. The lengths of samples are not equal. We simply fill short samples with empty symbols in tail, making all samples in one training batch equally long. This trick may force the model to do lots of inferences from an empty symbol to itself when doing feed-forward process. When doing back-propagation process, the error vectors are timed with a mask matrix, preventing the useless empty-to-empty inference errors from going back. A more clever but arbitrary way is to concatenate short samples into long samples before grouping them.

To migrate to a batch version of LSTM from a single-sample version is not hard. Making the batches and updating the embedding weights are the major gaps between these two versions. To make a batch, the user can use the Select primitive, fetching columns from the vocabulary embedding table with only one extra line of code. To update embedding of all samples in one batch, the SelectiveSub primitive can be used without additional for-loops. The following codes show the difference of updating between these two versions.

single-sample version:

```
for (int t = 1; t < dEmb.size(); ++ t)
  emb_weight[sent[t - 1]] -= rate * dEmb[t];
```

multi-sample(batch) version:

```
for (int t = 1; t < dEmb.size(); ++ t)
  emb_weight.SelectiveSub(rate * dEmb[t], index_batch[t - 1]);
```

where $index_batch$ is a vector of sample indices in this batch.

The boost brought by this batch method is prominent. A batch size of 10 would make one training epoch 4× faster. A batch size of 128 would raise this number to approximately 25×, which is much faster than the equivalent numpy

implementation. The batch size cannot go arbitrary large. There are two ceilings for simply increasing it. The first one is GPU memory. Having large batch size may overflow the GPU memory due to the intermediate matrices generated during training. The second one is the accuracy. Increasing the batch size usually damages the accuracy for one training epoch. However, it could lead to a better result by enabling us to train much more epochs. The tuning of batch size is task-related and beyond the scope of this paper.

To implement a multi-GPU LSTM model is both tricky and interesting. We will present details of our exploration in Section 7.2.1.

CHAPTER 7
Experiments and Evaluations

In this chapter we present the initial experimental results of several successful models implemented based on Minerva platform. Among these models, AlexNet [20] for 1K ImageNet classification, the VGG net [34] and the GoogleNet model [36] belong to the convolutional neural network category. The tests are executed against a cluster of GeForce Titan Black GPUs. We also present the Long Short Term Memory model, an advanced recurrent neural network model, implemented using Minerva.

7.1 Convolutional Neural Network

Table 6-1 shows the scalability of this model on a 4-core GPU cluster, as well as the VGG model and the GoogleNet model (These two models are implemented on Minerva system by Tianjun). The training procedure is distributed to GPUs by data parallelism strategy. Each GPU functions as one parameter server. The most simple way to do the parameter exchange is to give one server special master position. Each of other parameter servers sends their updates to the master and receives the same newly updated weights. The synchronous execution doesn't introduce overhead since the computation procedure on each server is identical. But due to the underlying CPU-GPU architecture, the costs of transmitting the data are not balanced among different servers. For example, data transmission from one GPU core to another one may include expensive CPU-CPU communication. We carefully designed the data communication paths across distributed GPUs to save the cost. This trick is shown in Figure 7-1.

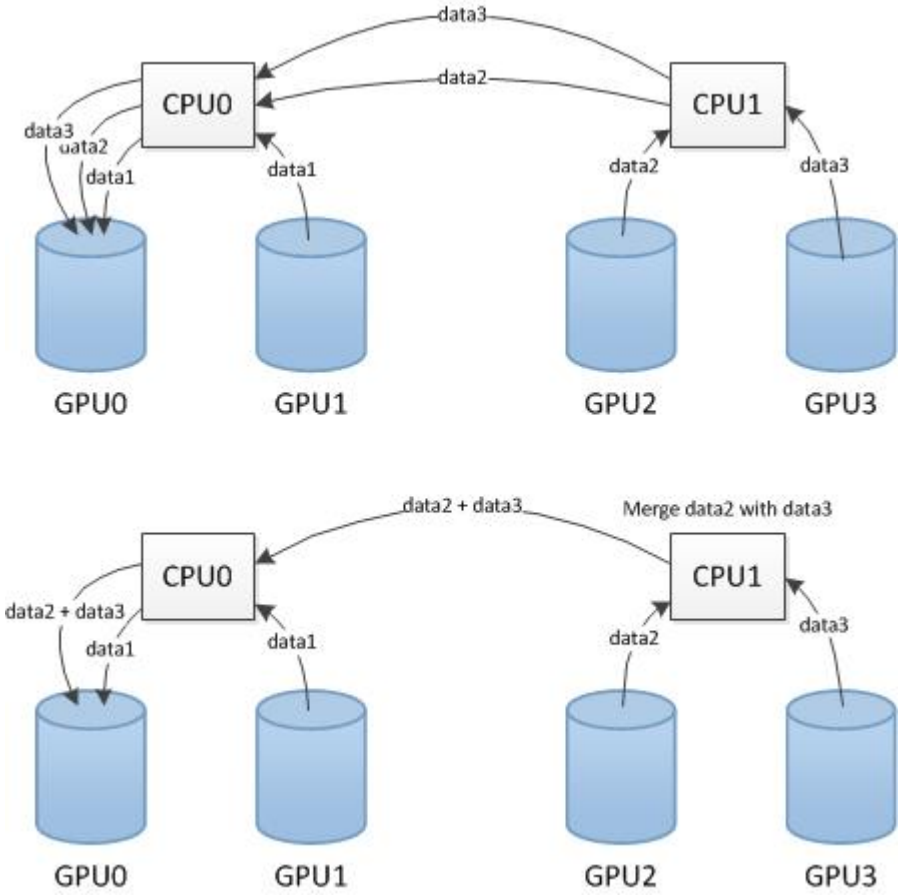

Figure 7–1: Suppose each GPU (except GPU0) sends the weight update to GPU0. The naive implementation results in double CPU-CPU transmissions. Merging data from GPU2 and GPU3 locally can reduce the communication overhead.

Table 7–1: Scale-up for CNN training with Minerva

model	1-GPU	2-GPU	4-GPU
AlexNet	198 img/s	388.7 img/s (1.95x)	731.4 img/s (3.69x)
VGG net	15.3 img/s	22.53 img/s (1.48x)	29.6 img/s (1.93x)
GoogleNet	101.0 img/s	199 img/s (1.99x)	355.5 img/s (3.52x)

Table 7–2: Performance for CNN training on 1-GPU (Caffe v.s. Minerva)

model	Caffe	Minerva
AlexNet	201 img/s	198 img/s
VGG net	15.5 img/s	15.3 img/s
GoogleNet	82.0 img/s	101.0 img/s

Table 6-2 shows the performance comparing to the implementation of these CNN models using Caffe, a task-spefic and hand-tuned tools for single CPU-GPU environment. One important optimization that helps much is to parallelize the convolutional execution (Thanks to Yutian). We use the same grouping settings as Caffe, achieving competitive results on single CPU-GPU hardware environment. Limited by Caffe's tight couple between the implementation and the underlying hardware, it is almost impractical for users to take advantages of the multi-GPUs execution. Thus to scale up Caffe's implementation would take lots of extra efforts on designing and tuning, which is beyond the scope of this paper.

7.2 Long Short Term Memory (LSTM)

To our surprise, the performance of Minerva's implementation is poor in the first place. There is constant time of overhead for even the smallest configuration of the model, while a simple numpy code can run extremely fast under small configurations. Current typical settings of LSTM have only about hundreds of units in the hidden layer and the embedding layer. By the unrolling procedure, the training of one training sample involves a deep chain of small-scale matrix computations. The caveat is that GPU acceleration and MKL libraries are probably not able to deliver the performance gain as we expected in CNN training. As we find out, the overhead comes from two things: the semaphores and the transpose operations (Thanks to Chuntao). The transpose operations can be implemented together with matrix multiplication in a more clever way. The semaphore overhead on the other hand, is difficult to get over due to the nature of DAG processing. In the case of a very large DAG, this semaphore overhead would be non-negligible.

To compare the batch training speed with existing tools, we use dimensions of 512 for the embedding layer and the LSTM memory, which is the exact

configuration of Google's show and tell task [38]. The model is fed with IMDB dataset. Table 6-3 shows the performances of Minerva and Theano under such a configuration. Both models have underlying GPU implementation supports. As we can see the batch size may affect the training efficiency greatly. Minerva outperforms Theano under large-scale settings, but also has drawbacks of system overhead under small-scale settings.

Table 7–3: Performance for LSTM training on 1-GPU (Theano v.s. Minerva)

batch size	Theano	Minerva
128 samples	435 samples/s	274 samples/s
256 samples	609 samples/s	511 samples/s
512 samples	704 samples/s	931 samples/s

As we can see, the performance of Minerva increases almost as quick as the speed-up. This implies that there is still potential for the model to grow wider and deeper. For a large model with large hidden size and batch size, Minerva may show good performance. However, for a small hidden size with small batch size, the system overhead would be a severe drawback for Minerva. CPU-based programs, such as RNNLIB [13] by Alex Graves, would do remarkably better under small configuration. It is worth to notice that increasing the batch size would give us faster epochs, as well as hurting the accuracy for one training epoch. Thus to pick the right batch setting is very tricky, which depends on specific tasks.

7.2.1 Multi-GPU Training

To take the advantage of multi-GPU, the procedure goes similarly as in CNN training (Section 6.2.2). We tried this approach on a 2-GPU cluster. 2 GPUs execute sample by sample in parallel. After both GPUs finish one training epoch, a sequential weight update is carried on. This naive procedure doesn't provide any speed-up in the first place. We examined our code very carefully and excluded the existence of blocking calls in training. The insight

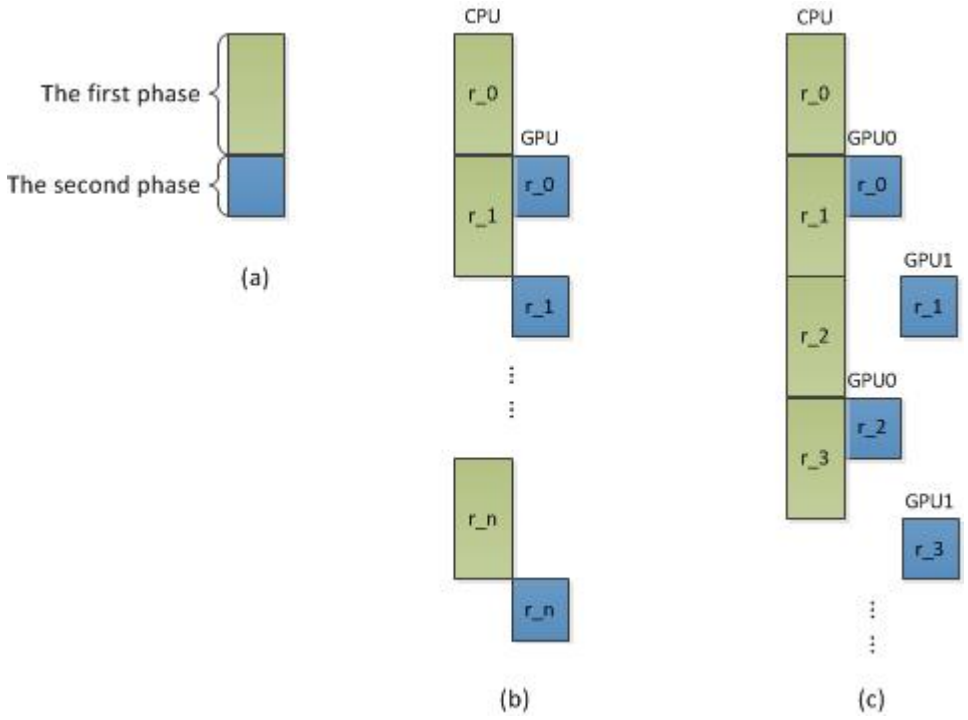

Figure 7–2: (a) The two phases of a sample-based run. (b) The execution of the training procedure on one GPU. (c) The execution of the training procedure on a 2-GPU cluster.

we later have for this phenomenon is inspiring. It helps us better understand the Minerva system's behaviour.

The problem again lies in the nature of the LSTM algorithm. The unrolling procedure creates a deep chain of low cost operations for one training sample. Recall that the execution of Minerva system consists of two phases. The first phase terminates when the actual computation is launched on device. This phase is executed on CPU. Therefore it cannot be parallelized in a unary-CPU structure. The second phase, on the other hand, executes on device and can be distributed among a multi-GPU cluster. Let us call a first-phase execution and the corresponding second-phase execution a run. The diagram 7-2 illustrates how a training procedure is composed of several runs.

In a sample-based implementation, the first phase of a run takes more time than the second phase. Therefore in this case, as we can see in 7-2(b), the GPU execution is totally dominated by the CPU execution. This explains the constant time of overhead even under the smallest configuration. It also explains why there is no speed-up with 2 GPUs. As shown in 7-2(c), the execution on the 2-GPU cluster is equivalent to a sequential execution.

7.2.2 Multi-GPU Batch Training

The ratio between the two phases in a run of AlexNet is totally different from 7-2(a). In AlexNet training, the second phase dominates the execution of a run, leading to a good scale-up. This condition is a must for a near-linear scale-up. Therefore we adopted the exact same 2-GPU approach on the batch-based implementation. In this case, the execution is illustrated by the diagram 7-3.

The execution in 7-3(c) is an ideal case. In reality, these runs do not overlap as perfectly as such. The weight update part is a blocking execution. The intermediate matrices quickly fill the GPU memory. Therefore n in 7-3(c) is typically very small comparing to the size of training samples. Data copy may also affect the overlapping of these runs. However, in general, diagram 7-3 suggests that the ratio between the two phases is essential to the scale-up performance. As the size of the model increases, the second phase becomes more significant, resulting in better scale-up. The caveat is that the GPU memory creates a ceiling on increasing the model size.

To show the real scale-up, we use a hidden layer of 1024 with different batch sizes (128, 256 and 512). With a batch size of 128, the 2-GPU version have only 1.28× speed-up. This number increases as the batch size goes larger. The speed-up is 1.44× for a batch size of 256, and 1.55× for a batch size of

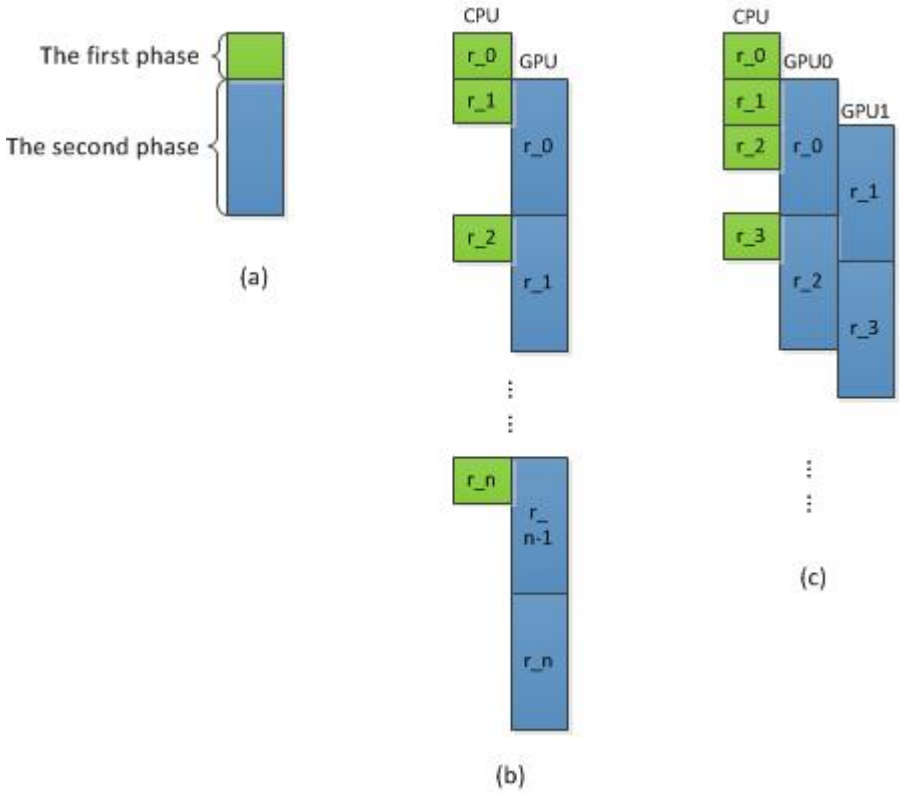

Figure 7–3: (a) The two phases of a batch-based run. (b) The execution of the training procedure on one GPU. (c) The ideal execution of the training procedure on a 2-GPU cluster.

512. As we can see, these numbers match the scale-up for VGG net, suggesting that the scale-up of VGG probably suffers due to the same reason.

CHAPTER 8
Summary and Ongoing Work

Minerva is a domain-specific engine tailored for efficient and scalable deep learning training. With more profound understanding of the domain algorithms and principled system design philosophy, we can achieve both programmability comparable to productivity-oriented tools, and performance competitive with or better than task-specific and hand-tuned deep learning tools.

Our ongoing works are multifaced. For now, users still need to have some knowledge of underlying hardware in order to get better performance. So one improvement to the system is to enhance Minerva's ability to automatically handle model parallelism. For example, we are working on a DAG reuse optimization mechanism. Such a mechanism aims at reducing the system overhead, making small-grained training like LSTM more efficient. We would also design a mechanism for the system to automatically figure out the optimal placement policy given the DAG representation. This mechanism will hide more details of the underlying hardware environments, leading to better programmability and saving lots of efforts on performance hand-tuning. Another way to improve the programmability is to have a trade-off strategy to automatically partition the matrix which may increase the model parallelism and bring performance boost (e.g. the grouping trick used in convolutional neural networks).

As we are trying to make Minerva's implementation of LSTM model more efficient on user level, based on Minerva system, there are also many advanced

deep learning algorithms and optimizations research works awaits. One interesting work is to visualize the network's behaviour during the training procedure can be very helpful for machine learning researchers, to understand the nature of deep learning approaches, especially for Recurrent Neural Network and Long Short Term Memory architectures. There are a large number of algorithms from a variety of industry applications that can benefit from Minerva. We are now actively moving towards releasing Minerva as an open source project so that the community may contribute to this progress.

References

[1] David Abrahams and Ralf W Grosse-Kunstleve. Building hybrid systems with boost. python. *CC Plus Plus Users Journal*, 21(7):29–36, 2003.

[2] 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, page 3, 2010.

[3] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, 2008.

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

[5] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[6] Christopher Clark and Amos Storkey. Teaching deep convolutional neural networks to play go. *arXiv preprint arXiv:1412.3409*, 2014.

[7] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Technical Report IDIAP-RR 02-46, IDIAP, 2002.

[8] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

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

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision*

and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on, pages 248–255. IEEE, 2009.

[11] Jeff Donahue, Lisa Anne Hendricks, Sergio Guadarrama, Marcus Rohrbach, Subhashini Venugopalan, Kate Saenko, and Trevor Darrell. Long-term recurrent convolutional networks for visual recognition and description. *arXiv preprint arXiv:1411.4389*, 2014.

[12] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[13] Alex Graves. Rnnlib: A recurrent neural network library for sequence learning problems, 2008.

[14] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *Signal Processing Magazine, IEEE*, 29(6):82–97, 2012.

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

[16] MKL Intel. Intel math kernel library, 2007.

[17] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the ACM International Conference on Multimedia*, pages 675–678. ACM, 2014.

[18] Eric Jones, Travis Oliphant, and Pearu Peterson. Scipy: Open source scientific tools for python. *http://www. scipy. org/*, 2001.

[19] A Krizhevskey. Cuda-convnet, 2014.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[21] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[22] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, 2012.

[23] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[24] T Mikolav et al. Word2vec project. *Retrieved September*, 3, 2014.

[25] Tomas Mikolov, Stefan Kombrink, Anoop Deoras, Lukar Burget, and J Cernocky. Rnnlm-recurrent neural network language modeling toolkit. In *Proc. of the 2011 ASRU Workshop*, pages 196–201, 2011.

[26] 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.

[27] CUDA Nvidia. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15, 2008.

[28] T Oliphant. A guide to numpy, vol. 1. *Spanish Fork: Trelgol Publishing*, 2006.

[29] 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.

[30] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[31] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 49–68. ACM, 2013.

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

[33] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[34] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[35] Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112, 2014.

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[37] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S Schreiber. Presto: distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 197–210. ACM, 2013.

[38] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

[39] Minjie Wang, Tianjun Xiao, Jianpeng Li, Jiaxing Zhang, Chuntao Hong, and Zheng Zhang. Minerva: A scalable and highly efficient training platform for deep learning, 2014.

[40] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, volume 8, pages 1–14, 2008.

[41] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J 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.