SINGA: A Distributed System for Deep Learning
(Technical Report)

Wei Wang[†], Gang Chen[‡], Tien Tuan Anh Dinh[†], Jinyang Gao[†], Beng Chin Ooi[†], Kian-Lee Tan[†]
†School of Computing, National University of Singapore, Singapore
‡College of Computer Science, Zhejiang University, China
†{wangwei, dinhtta, jinyang, tankl, ooibc}@comp.nus.edu.sg, ‡cg@zju.edu.cn

ABSTRACT
In this paper, we propose and design a distributed deep learning platform, called SINGA, for training large-scale deep learning models. Our design is based on two observations. First, the structures and training algorithms of deep learning models can be expressed using two simple abstractions, namely the layer and the edge. SINGA allows users to write their own training algorithms by exposing intuitive programming abstractions, consisting of data objects (layers and edges) that define the model, and of computation functions over the data objects. Users has little awareness of model (or data) partitioning, synchronizations and communications.

Second, there are multiple approaches to partitioning the model and the training data over multiple machines for model parallelism and data parallelism respectively. Each of these methods has a different communication and synchronization overhead. Such overhead directly affects the system's scalability. We analyze the fundamental trade-offs of existing parallelism approaches, and propose an optimization algorithm that generates the parallelism scheme with minimal overhead. Experiments show that SINGA is scalable than existing parallelism approaches.

Categories and Subject Descriptors
H.3.4 [**INFORMATION STORAGE AND RETRIEVAL**]: Systems and Software—*Distributed systems*; I.2.6 [**ARTIFICIAL INTELLIGENCE**]: Learning—*Parameter Learning*

General Terms
Design, Performance

Keywords
Deep Learning,Distributed Memory

1. INTRODUCTION
Deep learning refers to a set of feature (or representation) learning models that consist of multiple (non-linear) layers, where different layers learn different levels of abstractions (representations) of the raw input data [18]. Examples of deep learning models inlcude Deep Convolutional Neural Network (DCNN) [15] and Deep Belief Network (DBN) [12]. In recent years, deep learning has been employed successfully in various fields such as computer vision [15, 8, 30], natural language processing [22, 4] , automatic speech recognition [26, 6, 25], and cross media analysis [32]. Consequently, it has gained a lot of traction in both academia and industry.

For deep learning to be widely adopted, there are two key challenges to be overcome. First, the accuracy of deep learning models is influenced by the model parameters and the training datasets. In particular, a larger training data and a bigger model, in terms of the number of parameters, can lead to better accuracy [2, 16]. However, the memory consumption for a larger training dataset and model may exceed the memory capacity of a single CPU or GPU. In addition, computational cost during training may be unacceptably high. For instance, it takes about 10 days [33, 23] to train the deep convolutional neural network [15] with 1.2 million training images and 60 million parameters using one GPU [1].

To address the first challenge, distributed deep learning that runs on a cluster of machines have been developed [5, 3, 1]. In particular, there are two parallel paradigms that have been adopted to scale the tranining: *model parallelism* and *data parallelism*. In model parallelism, the model is partitioned into sub-models and distributed across a group of CPU (GPU) workers such that each sub-model is small enough to be processed in the memory of a single CPU (or GPU). Since upper layers of the model require data from lower layers residing on different workers within the same group, frequent synchronization and communications between workers are required and these constitute the major overhead. On the other hand, in data parallelism, the training dataset is partitioned into shards, and each group of CPU (GPU) workers will run a complete model on its assigned shard. In other words, multiple worker groups concurrently and independently compute updates for the parameters, which are then aggregated and applied on the parameters by the parameter servers. The overhead of data parallelism is the communication cost of trasferring the parameters (e.g., 60 Million parameters for the deep CNN model [15]) and updates between the workers and the parameter servers.

The second challenge is that existing deep learning platforms are model specific [23, 33, 14]. These systems have been designed to parallelize the training of specific deep convolutional neural network model [15], and hence they are not able to support other deep learning models. To support multiple deep learning models require developing and implementing each of these models, which is time consuming and impractical. Moreover, while distributed platforms such as GraphLab [21] and Spark [34] are designed for machine learning, there is a lack of optimization based on the properties (e.g., big model size) of deep learning models.

In this paper, we propose and design a distributed deep learning platform, called SINGA, for training large-scale deep learning models on commodity computers. Our design is based on two observations. First, the structures and training algorithms of deep learning models can be expressed using two simple abstractions,

[1]According to the authors, with 2 GPUs, the training still took about 6 days.

namely the *layer* and the *edge*. We investigated popular deep learning models listed in Table 1, and found that they share a similar structure comprising multiple layers connected by edges. Moreover, the operations involved in the training algorithms can be associated to the edges or layers. This means we can potentially design SINGA to shield users of model (or data) partitioning, synchronizations and communications. Second, there are multiple approaches to partition the training model and the training data over multiple machines for model parallelism and data parallelism respectively. Each of these methods has a different communication and synchronization overhead. Such overhead directly affects the system's scalability. This offers opportunities for optimization to build an efficient and scalable SINGA.

We have made the following contributions:

1. We design and implement a system called SINGA specifically catered for distributed deep learning. The system offers a *simple programming model* and is highly optimized to handle large-scale deep learning models. New deep learning models, can be implemented by customizing the *layer* and *edge* abstractions to create new layers and edges. The model structure, i.e., how the layers and edges are assembled and the type of each layer and each edge, is specified in a model configuration file. SINGA will optimize the training processing in the distributed environment based on the model structure and the meta information provided by each layer and each edge, e.g., layer size. Thus, the developers need not be aware of the distributed training environment. Moreover, SINGA handles the underlying communications and synchronizations.

2. SINGA is highly scalable. Given a deep learning job and a cluster setting, SINGA optimizes the job running time at two levels. First, optimization techniques (high-performance numerical libraries and specialized hardware) are employed to speed up the computation within one node. Second, internode overhead (communication and synchronization) from model parallelism and data parallelism is minimized by choosing the optimal partitioning scheme. These optimizations enable SINGA to train large models using large datasets efficiently.

3. We implement real-life deep learning applications in SINGA, and report the implementation and performance results, in terms of model accuracy, system efficiency and scalability. Our extensive performance evaluation shows that SINGA is a practical system for deep learning.

2. DEEP LEARNING

In this section, we provide some background about training deep learning models. First, we give a simple example to illustrate the concepts involved in deep learning models. Second, we introduce the training procedure.

2.1 A Sample Deep Learning Model

We describe the concepts involved in deep learning through a simple deep learning model, called Multilayer Perceptron (MLP). MLP is a feedforward neural network model that maps input feature (data) onto appropriate outputs. There are multiple layers in a MLP, where each layer is fully connected to the upper layer. Every **layer** consists of a vector of neurons, with every neuron being a float (or double) variable. The neuron variables constitute the **feature vector** of the layer, and are calculated based on the feature vectors of the lower layers. Figure 1 shows a sample MLP, which consists of three layers. We use the term **edge** to refer to the connections between neurons of two connected layers. The bottom layer accepts the **input data** (input feature vector), denoted as $\mathbf{v}$ [2], e.g., the pixel values of an image. The second layer is called the hidden layer, whose feature vector, denoted by $\mathbf{h}$, is computed by Equation 1. In Equation 1, W_1 is a **parameter matrix** that represents the connection weights between neurons, $\mathbf{b}$ is a **parameter vector** with each element associated to a neuron on the hidden layer. σ is the logistic (sigmoid) activation function. The top layer is called the output layer or prediction layer, which is computed based on the hidden layer. In our example, we use the softmax function as shown in Equations 2, which has no parameters.

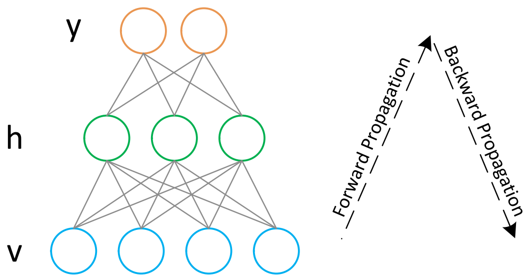


Figure 1: Training Multilayer Perceptron using Back-Propagation

$$\mathbf{h} = \sigma(\mathbf{v}^T W + b) \qquad (1)$$

$$y_i = \frac{e^{h_i}}{\sum_j e^{h_j}} \qquad (2)$$

Given an input feature vector $\mathbf{v}$, the output feature vector $\mathbf{y}$ is computed from the bottom layer to the top layer according to Equations 1 and 2. We can see, the output is determined by three factors. The first one is the structure of the model, including the number of layers, the size of each layer and the edges. The **model partition** we will study in the later sections refers to the partitioning of the structure, e.g., splitting one layer into two sub-layers. The second factor is the type of each layer, which decides how to compute the values of the feature vector based on the parameters and feature vectors from other connected layers. For instance, we use Equation 1 and 2 respectively for computing the feature vectors of the hidden layer and the output layer in the sample MLP. The third one is the parameter assignment. Some assignments would lead to better outputs (predictions), i.e., predictions that are close to the ground truth. These three factors together determine the deep learning model.

2.2 Training of Deep Learning Models

As introduced in Section 2.1, there are three factors that determine the performance, e.g., the prediction accuracy measured by the distance between the predicted outputs and the ground-truth, of the deep learning model. When we train a deep learning model, we should train all the three factors. In practice, the structure of the model and the types of each layer are trained by trials. Only the parameters are trained automatically. In this paper, we also refer to the training of a deep learning model as the training of the parameters of the deep learning model and assume the other two factors are pre-determined.

To describe the training algorithms, we represent the deep learning model as a function $f(\mathbf{v}|\Theta)$ that computes the output for an

[2] In this paper, we use bold lowercase symbols for vectors, uppercase symbols for matrices or sets

input feature vector $\mathbf{v}$ and the model parameters denoted as Θ. The goal of the training procedure is to find the assignment of Θ that optimizes a training objective function based on $f(\mathbf{v}|\Theta)$. The objective function is defined to capture the semantics of the training goal. For example, suppose we want to train a deep learning model for image classification. Our goal then is to minimize the classification error. To achieve this goal, we set the training objective function to calculate some measure in accordance to the classification error, and find the parameters that minimize this objective function. The training procedure typically minimizes the objective function. Hence we call it the loss function or loss in the remainder of this paper. The goal of the training procedure can be expressed in the form of Equation 3, where $(\mathbf{v}, \mathbf{t})$ is a training pair, $\mathbf{t}$ is the ground truth for data $\mathbf{v}$, and $\mathcal{L}()$ denotes the loss function. The last term $\Phi()$ is a regularization term for over-fitting [31]. If the training record has no ground truth, i.e., $\mathbf{t}$ is empty, the training is unsupervised.

$$\min_{\Theta} \sum_{\mathbf{v},\mathbf{t}} \mathcal{L}(f(\mathbf{v}|\Theta), \mathbf{t}) + \Phi(\Theta) \qquad (3)$$

We can observe from Equation 3 that the training procedure is actually solving a numerical optimization problem. Because deep learning models are usually non-convex and non-linear, there is no closed-form solution for Θ. As such, gradient based optimization methods such as the mini-batch Stochastic Gradient Descent (SGD) are typically employed to solve Equation 3, SGD and its extensions [27, 7] are widely used for training deep learning models. In our system, we support both the basic SGD and its extensions. The basic SGD is shown in Algorithm 1. It iteratively updates the parameters until convergence (e.g., the change of parameters is within a threshold) based on the gradients (derivatives) of the loss w.r.t, the parameters. One key step in the algorithm is to compute the gradients of the loss w.r.t. all the parameters. Most computation is spent on the gradient calculation. The parameter update involves simple algebraic operations, whose complexity is linear to the size of parameters.

Algorithm 1: Mini-Batch SGD

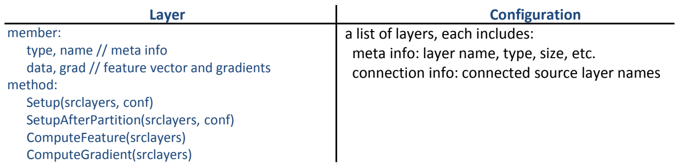
```
  Input: $D$, $b$ //training dataset,mini-batch size
  Input: $\alpha$, $\tau$ // learning rate, convergence threshold
1 repeat
2     $B = \text{miniBatch}(b, D)$
3     $\nabla\Theta = \frac{1}{b}\sum_{(\mathbf{v},\mathbf{t})\in B} \frac{\partial\mathcal{L}(\mathbf{v},\mathbf{t})}{\partial\Theta}$
4     $\Theta = \Theta - \alpha\nabla\Theta$
5 until $||\nabla\Theta||_2 < \tau$;
```

There are two typical approaches to calculate the gradients, namely Contrastive Divergence (CD) [11] and Back-Propagation (BP) [17]. Table 1 summarizes the applications of the two algorithms for popular deep learning models. As there are many deep learning models, it is tedious to implement the respective CD or BP for each of them. To develop a system that is easy to use, we support the CD and BP in our system based on some abstract functions, so that the users can implement their deep learning models and calculate the gradients by overloading these functions. We only need to manage the training flow, e.g., calling the corresponding functions. In the next section, we shall describe the programming model in our system and use the training of the sample MLP as our example.

3. PROGRAMMING MODEL

Table 1: Summary of Deep learning models according to gradient calculation algorithms.

Algorithm	Model
Back-Propagation (BP)	Multilayer Perceptron (MLP) Sparse Auto-Encoder Denoising Auto-Encoder (DAE) Recurrent Neural Network (RNN) Convolutional Neural Network (CNN)
Contrastive Divergence (CD)	Restricted Boltzman Machine (RBM) Deep Belief Network (DBN) Deep Boltzman Machine (DBM) Conditional RBM
Hybrid	DBN-DNN



Layer	Configuration

member:
type, name // meta info
data, grad // feature vector and gradients
method:
Setup(srclayers, conf)
SetupAfterPartition(srclayers, conf)
ComputeFeature(srclayers)
ComputeGradient(srclayers)
	a list of layers, each includes:
	meta info: layer name, type, size, etc.
	connection info: connected source layer names

Figure 2: Basic Data Structures

In this section, we introduce the programming model of SINGA. Users can train their deep learning models by following this programming model.

After investigating popular deep learning models, we find that all of them can be expressed as a combination of layers. In addition, the gradients of loss w.r.t. model parameters used in SGD algorithms can be calculated by two algorithms, namely Contrastive Divergence(CD) [12] and Back-Propagation(BP) [13]. Table 1 summarizes the applications of the two algorithms for some popular deep learning models. Considering that it is tedious to implement the BP or CD algorithm for every model, we propose a base Layer class and implement the BP and CD algorithm against it. Then users only need to override the base Layer class to implement their own layers and models. SINGA takes care of the training by calling BP/CD and update the parameters according to SGD algorithms.

Figure 2 shows the base data structures used to conduct the training. The base Layer class has some members for meta information like layer type, name and shape. It contains two data blobs, one for the feature vector and the other one for the gradient vector. Depending on the layer type, it may also has some parameters. There are four basic methods. The Setup function reads information from connected layers and configuration to set the meta data of this layer, e.g., the shape feature vector. Considering that the layer may be partitioned by SINGA automatically, users need to override the SetupAfterPartition method to re-setup the corresponding members. This is the only point that the users need to be aware of the model or data partitioning. SINGA will handle the data transferring between nodes (e.g., when two connected layers are located onto different nodes) transparently to the users at runtime. Users extend the base Layer class to implement their own layers. The model structure is defined in a configuration file. SINGA parses the configuration file and creates the model by instantiating the corresponding layer objects and connecting them. After that it conducts the SGD training, which uses BP/CD to calculate the parameter gradients. The BP/CD algorithm internally calls the ComputeFeature and ComputeGradient functions of each layer. We have implemented common layers such as convolution layer, pooling layer, etc. as built-in

```
HiddenLayer: Layer
member:
    data, grad… /inherits from base Layer
    W, b // parameters
method:
    Setup(srclayers, conf){
    // read info from conf and srclayers,
    // setup the shape of data, and grad
    // setup the shape of W, b
    }
ComputeFeature(srclayers){
     data=logistic(dot(srclayers[0].data,W.data)+b.data); // Equation 1
}
ComputeGradient(srclayers){
     srclayers[0].grad=data*(1-data)*dot(grad, W.data.transpose());
     W.grad=data*(1-data)*dot(src[0].data.transpose(), grad);
     b.grad=data*(1-data)*grad;
}
```

```
Configuration
layer:
    name: “data layer”
    type: kDataLayer
layer:
    name: “label”
    type: kLabelLayer
layer:
     name: “hidden layer”
     type: kHiddenLayer
     srclayer: “data layer”
      shape: 3
layer:
      name: “softmax”
      type: kSoftmaxLossLayer
      srclayer: “hidden layer”,
                “label layer”
```

Figure 3: Pseudo Code for MLP example

layers. Users can use these built-in layers directly to construct a deep learning model and train it over user provided dataset.

We use the MLP example from Figure 1 to explain the programming model. In Figure 3, we extend the base Layer class to create a HiddenLayer class for the hidden layer. Similarly we can extend the base Layer to implement the data layer and softmax loss layer. Once all layers are defined, we create the model by configuring the type of each layer and their connections as shown in the figure. Then we submit the training job to SINGA and let it conduct the training.

4. TRAINING OPTIMIZATION

One goal of our system is to exploit large scale parallelism to speed up the training process with a large amount of data. Specifically, we want to reduce the training time to reach a given loss (or accuracy), i.e., Equation 3 by adding more nodes into the system. The total training time is determined by the number of iterations (mini-batches) to converge and the time for processing one iteration. In [35], the first factor is called the *statistical efficiency* and the second factor is called the *hardware efficiency*. The statistical efficiency depends on the size of the mini-batch, i.e., the number of training records being processed in one mini-batch, the learning rate α in Algorithm 1, and some other hyper-parameters [10]. There are researches [7, 29] on optimizing the statistical efficiency by changing these hyper-parameters and the updating equation (Line 4 in Algorithm 1). In this paper, we shall focus on optimizing the hardware efficiency by distributing the computation for one mini-batch onto multiple computing nodes. Hence, we fix the mini-batch size and other factors that affect the statistical efficiency to make sure the training procedure converges to the same loss using the same number of mini-batches as trained on a single node.

Given a mini-batch of training records, we aim to find the fastest parallelization scheme to compute the gradients to update the parameters. In this section, we give two parallelization approaches, namely, data parallelism and model parallelism, that can be adopt in our system, followed by a detailed cost analysis. Then we discuss how these two approaches can be integrated into a more efficient parallelization scheme.

During the cost analysis, we ignore the synchronization cost by assuming that all the machines run at the same speed. This synchronization cost will be analyzed at the end of this section. For each parallelization approach, we partition the computation load evenly onto workers. Therefore, the computation cost for all the parallelization approaches is the same. The cost we are going to analyze reduces to the communication overhead introduced by the different parallelization approaches for one mini-batch. To simplify the analysis, we assume that all communication goes through a single switch. The communication cost is then determined by the total amount of data being transferred through the network switch.

We will use B to denote the mini-batch size, N to denote the number of machines (nodes) to be used, D to denote the total size of feature vectors in all layers of the deep learning model, and P to denote the size of parameters.

4.1 Data Parallelism

For mini-batch SGD, the gradient of one parameter is the average of gradients contributed by (calculated based on) all the training records from the mini-batch. To parallelize the SGD algorithm, one basic idea is to distribute the mini-batch onto multiple nodes (called workers). All workers have a replica of the model parameters and run in parallel to compute the gradients of the (whole) model parameters. Figure 4a shows the feature vectors (there are 3 training records in one mini-batch. Each has a feature vector) and parameters of one layer. Figure 4b shows that using data parallelism, the training records are partitioned onto 3 workers. Hence, the feature vectors are partitioned onto 3 workers. The parameters are replicated on each worker. After one iteration, every worker sends the gradients (averaged among the training records it owns) to a parameter server [20, 5] (may consist of multiple nodes), which averages the gradients and updates the parameters. In the next iteration (mini-batch), every worker fetches the fresh parameters from the parameter server and computes gradients against its partition of the next mini-batch.

In data parallelism, for one iteration, every worker needs to communicate with the parameter server twice, one for fetching the fresh parameters and the other one for sending the gradients of all parameters. The communication cost of data parallelism, denoted by C_d, in terms of number of floats being transferred is shown in Equation 4. The training data is partitioned at the beginning of training and is stored on the local disk of each worker.

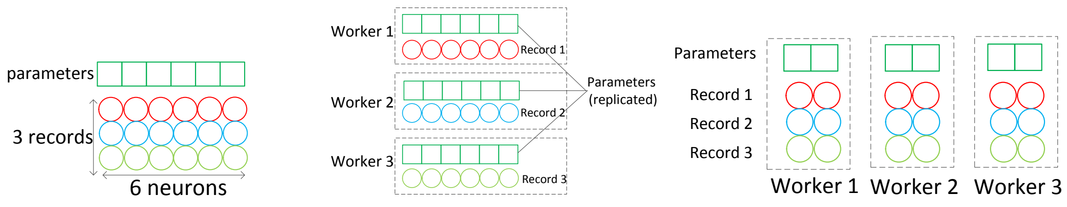
$$C_d = P * N + P * N = 2P * N \quad (4)$$

Unfortunately, the cost here grows linearly with the number of machines. For most traditional machine learning problems, the parameter size is quite small (e.g. less than 1K) and the extra cost here are considered as trivial. However, for deep learning problem, the parameter size is much larger and the cost here cannot be ignored.

4.2 Model Parallelism

The overhead of data parallelism comes from transferring large size of parameters (and gradients) between the workers and the parameter server(s). To avoid this data transferring overhead, model parallelism can be used to partition the whole model onto all worker nodes in the cluster. Specifically, as shown in Figure 4c, under model paralleism, the feature vector of every layer (including the input layer) is partitioned into sub-feature vectors (one for each worker). At the same time, the parameters associated with every layer are also partitioned so that each node maintains (i.e., stores and updates) a partition of the parameters. The worker computes gradients only for the parameters it maintains against the whole mini-batch training records. Consequently, it averages the gradients of the parameters for the whole mini-batch locally, and applies the update for the parameters it owns locally. Therefore, there is no communication cost from transferring parameters or gradients in this scheme.

However, model parallelism introduces another communication cost. The feature vectors of upper layers are computed based on feature vectors of lower layers. Because every feature vector is partitioned onto multiple workers, the *ComputeFeature* function (See Section 3) called by one worker may need the sub-feature vectors of lower layers which are maintained by remote workers. In such



(a) Feature Vectors & Parameters (b) Data Parallelism For One Layer (c) Model Parallelism For One Layer

Figure 4: Two Basic Parallelism Approaches For One Layer

Table 2: Properties of Sample Layers of DCNN [15]

Layer	Number of Parameters	Feature Vector Size
convolution layer	34,944	290,400
pooling layer	0	69,984
fully connected layer	37,752,832	4096

a case, the sub-feature vectors must be transferred through the network. For example, in the sample MLP (Section 2.1), the feature vector of the output layer is computed based on the whole feature vector of the hidden layer. Similarly, the *ComputeGradient* function may fetch sub-gradient vectors of upper layers from remote workers. In the worst case, every layer is fully connected to neighbor layers. The extra communication cost (overhead) involved by model parallelism is $2D * N$ for one training record. The total overhead of model parallelism for a mini-batch, denoted by C_m, is shown in Equation 5.

$$C_m = 2D * N * B \tag{5}$$

Unfortunately, the size of feature vectors in deep learning models can also be very large e.g., larger than 1M per training record in [15]. That is to say, none of these two basic approaches can distribute the deep learning model as desired. Moreover, the communication cost of these two approaches are still linear with the number of machines. Although we can reduce the computation cost of each worker to $1/N$, the communication cost increases linearly as the number of workers increases. Moreover this communication cost is likely to dominate the total cost as shown in our experiments.

4.3 Hybrid Parallelism

To keep the communication overhead acceptable, we design a hybrid approach. The hybrid parallelization scheme first partitions the nodes in the cluster into multiple groups. Let M be the group size, i.e., number of worker nodes within one group. Then it applies the data parallelism across groups. Specifically, it partitions the training data into N/M partitions, one for each group. The model is replicated for every group. The overhead from data parallelism (model replication) is the parameter (and gradient) transferring cost. Since every group transfers P parameters and P gradients, the total cost is $2P * N/M$. Within one group, we can simply apply the model parallelism as [5]. The overhead of model parallelism for one group, denoted as C_g, is shown in Equation 6. The total cost of this simple hybrid parallelism, denoted by C_h, is shown in Equation 8. The lowest overhead is achieved by setting $M = \sqrt{PN/DB}$. However, this simple approach does not consider the model structure. Instead, we propose to estimate the cost of model parallelism and data parallelism for each layer, and choose the best combination for all layers.

$$\begin{aligned} C_g &= 2D * M * B/(N/M) = 2DBM^2/N && (6)\\ C_h &= 2P * N/M + C_g * N/M && (7)\\ &= 2PN/M + 2DBM \geq 2\sqrt{PDB} && (8) \end{aligned}$$

Since there are multiple layers in one deep learning model, these layers may have different properties. For example, some layers may have large feature vectors but small number of parameters, while other layers may have small feature vectors but large number of parameters. The DCNN model is one such example that has three typical layers as listed in Table 2. We can see the convolutional layer and pooling layer have larger feature vectors than the fully connected layer. But they have fewer parameters than the fully connected layer. In this case, according to the analysis for model parallelism and data parallelism, we should choose different parallelism approaches for different layers.

The naive way to optimize for all layers is to enumerate the parallelization approaches for every layer and select the one with the overall smallest overhead. Algorithm 2 shows the optimization procedure; we run the algorithm by calling *OptimizeLayer*(layers, 0). The last saved setting is the best setting for each layer.

Algorithm 2: OptimizeLayer

```
  Input: layers, l //all layers and the current level
  Input: $min_C$ //current smallest latency
1 if layers[l] is the top layer then
2     $C = \sum_{i=0}^{l} Cost(layer[l])$;
3     if $C < min_C$ then
4         $min_C = C$;
5         save the layer setting.
6 else
7     for p ∈{data parallelism, model parallelism} do
8         layers[l].approach=p;
9         OptimizeLayer(layers, l+1, $min_C$);
```

The $Cost(layer[l])$ function estimates the overhead for the l-th layer. There are two types of overhead. One is from transferring parameters and gradients, which is just double the size of the number of parameters associated with this layer. The other one is transferring feature vectors and gradient vectors between connected layers. Line 2 is called when all layers' parallelization approaches have been determined (i.e., one parallelization combination is generated). Hence we know the partition scheme of the upper layers and the lower layers. In the best case, the connected layers all use data parallelism. Then there is no need to transfer any feature vectors. In other cases, without the layer type information which can be provided by programmers, SINGA estimates the cost by simply assuming the whole feature vector from the lower layer and the whole gradient vector from the upper layer will be transferred. If all layers use model parallelism, then the group uses pure model parallelism and the total overhead from all groups is as given Equation 8. Take the DCNN model as an example. Its lower layers mainly consists of convolutional layers and pooling layers which have small number of parameters and large size of feature vectors. Its upper layers mainly consists of fully connected layers which have large number of parameters and small feature vectors. The best parallelization combination should select data parallelism for the lower layers and model parallelism for the upper layers. This parallelization combination would have smaller overhead than that from pure model parallelism for the group as the overhead from data parallelism for the lower layers is smaller than that from model parallelism.

4.4 Asynchronous SGD

In the above analyses, we assume all nodes run in the same speed and ignore the synchronization cost. However, in reality, nodes are likely to be running at different rates. There are two synchronization points in hybrid parallelism. First, if remote fetching of sub-feature vectors happens, then we need to synchronize (barrier) all workers within this group before the fetching operations. Otherwise we may fetch invalid feature vectors. For example, if a remote node runs slower, then it may not finish computing the required sub-feature vectors when the fetch operation is issued. The fetched data is then invalid. We provide simple synchronization strategy based on the parallelism of connected layers. Specifically, if the connected two layers are both data parallelism, which means that there would be no remote fetching, then no synchronization is needed. Otherwise, we synchronize (barrier) all workers from one group before calling *ComputeFeature* and after calling *ComputeGradient*. Users can disable the synchronization and control the data consistency by themselves by calling DAry's Barrier function.

The second synchronization point is at the end of one mini-batch (iteration). The parameter server has to wait till the gradients sent from all groups have been received. Then it updates the parameters and send them back to the groups. The workers also has to wait until the new parameters from the servers have been received before starting the next iteration. We call this sequential training procedure as synchronous mini-batch SGD. [27] extends the mini-batch SGD and proposed asynchronous SGD. Asynchronous SGD relaxes the restriction on the parameter server to wait for all groups. Instead, it conducts the update for every parameter once it receives the gradient for this parameter from any group. Consequently, it responses to the groups fetching for fresh parameters immediately. Both working groups and servers run asynchronously. This asynchronous training mode would affect the statistical efficiency [35]. However it results in faster processing of the training dataset. The total training time to reach a certain loss is smaller than synchronous SGD [5]. Our optimization technique for one group, i.e., selecting the best parallelization combination for one group, benefits both synchronous SGD and asynchronous SGD. We support both SGDs in SINGA as two data consistency models, which we will describe further in the next section.

5. IMPLEMENTATION

In this section, we describe the implementation details of SINGA to support the afore mentioned parallelization approaches.

5.1 Overview

Figure 5 shows the basic components of SINGA. SINGA starts training a deep learning model by parsing a model configuration, which specifies the layer and edge structure, at the master (See Section 5.2). Next, based on the resources available in the cluster, the master assigns nodes for worker groups and table (parameter) servers (See Section 5.3). After that, it initializes the parameter table and starts workers to run their tasks. There are multiple *table servers*, each of which maintains a partition (i.e., a set of rows) of a distributed parameter table where model parameters are stored. *Worker groups* run in parallel to compute the gradients of parameters. Each group consists of one or more workers, running against a *data shard*, i.e., a partition of the training dataset. In one iteration, every group fetches fresh model parameters from the table servers, runs BP or CD algorithm to compute gradients against a mini-batch from the local data shard, and then sends gradients of all model parameters to the table servers. The data shard is created by loading training data from HDFS off-line (See Section 5.2). The master monitors the training progress and stops the workers and table servers once the model has converged to a given loss.

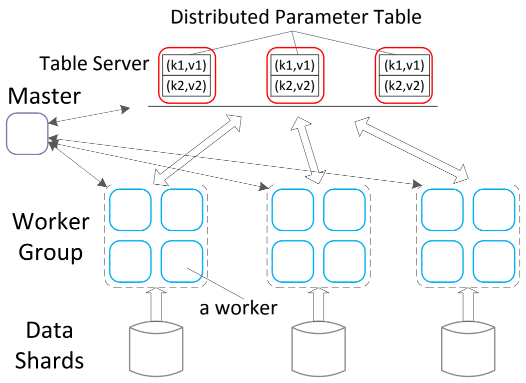
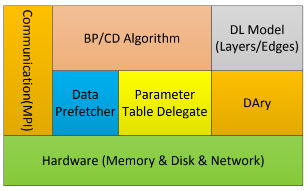


Figure 5: SINGA architecture, whose logical components include, *table servers* maintaining a distributed parameter table, *worker groups* running the BP or CD algorithm in parallel against a data shard containing a partition of the training data, and a master who is in charge of partitioning the model and training data as well as of resource coordination.

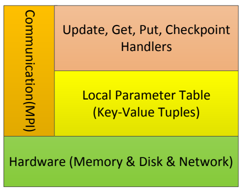
Figure 6a shows the software stack at the worker side. The BP (and CD) algorithm is implemented with the layer and edge abstractions as discussed in Section 3. It manages the interactions with the table servers through the parameter *table delegate* (See Section 5.3). Prefetching of training data is enabled and controlled by the BP (CD) algorithm. Users can customize (inherit) the layer and edge abstractions to implement their own deep learning models using DAry operations (See Section 3). The parameter table delegate and DAry use MPI [9] for network communication. Figure 6a displays the software stack of the table server. The table server maintains the parameters as key-value tuples and responds to the request from the table delegate by calling the corresponding handlers.

5.2 Master, Worker Group and Data Shard

At the initial stage, the master decides the parallelization approach of each layer according to Algorithm 2. Then it partitions the model by setting the partition dimension of the DAry objects, e.g., the feature, gradient and parameter of each layer. After that, it sends the partition configuration to every worker of each group. The worker reads the partition configuration and sets up its DAry objects. For example, if the hidden layer of the sample MLP is partitioned into two sub-layers, then its DAry objects would be partitioned along the second dimension (the first dimension is for data



(a) Software Stack of Workers


Communication(MPI)
Update, Get, Put, Checkpoint Handlers
Local Parameter Table (Key-Value Tuples)
Hardware (Memory & Disk & Network)

(b) Software Stack of Table Servers

Figure 6: Software Stack of Workers and Table Servers

parallelism). For the first worker of one group, when it constructs the DAry objects of the feature vector gradient vector and parameters of the hidden layer according to the partition configuration and its ID in the group, these DAry objects will be set up with half the size of the full feature vector and with the index range of the second dimension being $[0, l/2)$ (l is the length of the feature vector). Once every group has been set up (i.e., the layers and edges objects have been initialized), the master asks one group to initialize the parameter table by assigning (Gaussian or Uniform) random values for the model parameters and putting them into the parameter table. After that it notifies all worker groups to start training. The master monitors the training progress, e.g., the training step of each group.

The whole training dataset is partitioned into shards off-line, one per worker group. The master randomly shuffles the training record IDs and sends each group a sub-list of record IDs. Every worker fetches the records from HDFS based on the record ID list it receives. At runtime, worker groups run against data shards on their local disk. Data (records) prefetching is enabled to parallelize the IO and computation.

5.3 Distributed Parameter Table

Observing that the model parameters are frequently accessed by multiple worker groups, we store the model parameters in a distributed in-memory table in order to avoid high latency and single server bottleneck. The table is maintained by a set of table servers, as shown in Figure 5, where each table server is in charge of a partition, i.e., a set of rows, of the table.

The distributed table consists of (key, value) tuples, where both key and value can be customized to implement different table schemes. Basically, the key is an id assigned to a parameter, and the value field is just the newest value of the parameter. However, this simple scheme would lead to massive network messages because one message has to be generated for every parameter when the workers are getting the model parameters from the servers or sending the gradients to the servers. In fact, for deep learning models, parameters are usually processed as linear algebra objects. For example, in the MLP example in Section 2.1, as shown in Equation 1, the weight parameters are represented by a matrix $\mathbf{W}$, and the bias parameters are represented by a vector $\mathbf{b}$. Consequently, parameters from the same linear algebra object are always fetched or updated together. Hence, we can store a vector of parameters using one key. For example, we can store $\mathbf{W}$ as a long parameter vector by concatenating all rows. If the parameter vector is partitioned due to model parallelism, then we should store each of the partitioned parameter vector as a tuple in the parameter table.

Every worker has a *table delegate* that provides four basic APIs as shown in the following code snippet (left), to interact with the distributed parameter table. *get* and *collect* is a pair of operations to fetch parameters. *get* sends a request to the table server to fetch a tuple and returns immediately after the request has been sent. *collect* tries to read tuples returned from the table servers until it receives the tuple with the specified key. The *put* and *update* functions are non-blocking operations, which send the parameters and gradients of parameters to the table servers respectively. The design of *get* and *collect* seeks to parallelize the transmission of parameters and the computation. For instance, in the BP algorithm, the gradients of parameters from the top layers are computed first. Then we can send the gradients to the server and call the *get* function to request that parameter vector immediately. Next, we continue to compute the gradients of parameters of other layers. The computation and parameters (gradients) transmission is parallelized. In the next iteration, we call *collect* to collect the previous requested parameters.

```
API:                     Callback Function:
get(key)                 handle_get(key,&value)
collect(key)
put(key, value)          handle_put(key, value)
update(key, value)       handle_update(key, value)
```

To make the table server extensible, we provide three customizable handle functions on the server side to process the requests. For example, we overload the *handle_get* and *handle_update* to implement different consistency models in Section 5.4. The server puts all requests into a queue which is checked by calling the handle functions repeatedly. The request is removed from the queue until it is successfully handled (i.e., returns True). The handle functions are shown in the above code snippet (right).

5.4 Concurrency and Consistency

To maximize parallelism, we run all worker groups and servers asynchronously. As discussed in Section 4.4, there are two synchronization points. The first is at the worker side, and the second is at the table server side. As different SGD algorithms may require different consistency models, we provide the details on how the table servers implement different consistency models to control the synchronization to support the two SGD algorithms (synchronous SGD and asynchronous SGD).

SINGA provides two simple data consistency models by default: the strict consistency and the eventual consistency to support the synchronous SGD and asynchronous SGD respectively. Other consistency models can be implemented in a similar way as the implementation of the default consistency models described in the following paragraphs.

The consistency models are implemented by customizing the *value* field of the tuple, and the *handle_get* and *handle_update* functions. A version number is associated with each tuple, which is incremented every time the tuple is updated. In addition, we count the number of workers for each group, denoted as $count[group\_id]$, that have sent updates to the server. As shown in Algorithm 3, we overload the *handle_update* function to aggregate the gradients for a tuple until it receives $threshold$ updates from one group or all group for this tuple. Then it updates the parameters, reset the counter(s) and increase the version ID. We also overload the *handle_get* function as shown in Algorithm 4. *handle_get* returns False if the requested tuple version (the iteration/step ID of the requesting worker group) is larger than the newest one in the table or the update counter is not 0, which means that the updates from this group has not been applied. As a result, the get request stays in the request queue until the tuple in the table is updated to the requested version and the updates from the request group have been applied.

Next, we describe how synchronous SGD and asynchronous SGD are supported by Algorithm 4 and Algorithm 3. For synchronous SGD, all worker groups should see the same newest version of the

Algorithm 3: handle_update

```
Input: $key$, $value$ // key and gradients
1  t=table[key]; // current tupe value
2  g=key.gid; // group id
3  k=t.threshold;
4  if (t.count[gid]<k) then
5      t.grad[g]+=value;// aggregate gradients
6      t.count[g]+=1;
7      if (t.count[g]==k) then
8          apply_update($t$);
9          reset t.count[g], t.grad[g];
10         t.version+=1
11 if ($\sum_g t.count[g] == k$) then
12     apply_update($t$);
13     reset t.count, t.grad;
14     t.version+=1
15 return True;
```

model parameters in every iteration. This is done by setting the $threshold$ to be the total number of workers calculating gradients for this parameter vector. The get request from the next iteration from any group is blocked until all updates from the previous iteration have been finished, i.e., Line 12 in Algorithm 3 is executed. In this way, the worker groups run in Bulk Synchronous Parallel (BSP). Since all worker groups see the result from the last write operation to every parameter, this consistency model is strict consistency. For asynchronous SGD, every tuple should be updated asynchronously by all worker groups. This is achieved by setting the $threshold$ to be the number of workers within one group calculating gradients for this parameter vector. Once all workers within one group sent their gradients to the server, the server will execute Line 8 to update the parameters and increase the version ID. Then the next fetching request will return the fresh parameters without waiting for other groups to finish their updates. In this way, we synchronize the workers within one group. For different groups, they can run asynchronously. As a result, different groups may overwrite each other's update. The consistency model is eventual consistency.

Algorithm 4: handle_get

```
Input: $key$, $value$
// requested key and return value pointer
1  t=table[key]
2  g=key.gid
3  if ($key.version \leq t.version$ and $t.count[g] == 0$) then
4      $value \leftarrow table[key]$;// copy parameter vector
5      return True;
6  else
7      return False;
```

5.5 Failure Recovery

In SINGA, the worker (group) has no state information except the training step (i.e., the number of mini-batches that have been processed). Hence, we can easily restore the workers by resetting their training step and resending them the partition configuration which are maintained by the master. On the contrary, the table servers store parameter values, aggregated gradients, parameter versions, etc. It is necessary to replicate or checkpoint the table content in case of failure. In fact, the SGD algorithm requires checkpointing of the parameters. Due to the potential for over-fitting of the training, the best version of the parameters may be obtained before the convergence. In that case, we should select the checkpoint version that is the nearest one to the best version as the final version. Considering the best version discovery, SINGA uses checkpointing for failure recovery. Specifically, it performs checkpointing after every n_{cp} updates. After every successful update from *handle_update*, the table server calls *checkpoint_now(key, value)* (can be overloaded by users) to check whether it is time to do checkpointing. We use the simple checkpointing strategy which returns true for every n_{cp} updates, where n_{cp} is the checkpointing frequency. The checkpoint is conducted by appending the tuple onto disk file with the checkpoint version v_{cp} being the tuple version. At the initialization stage, each table server writes meta data about the table onto disk, e.g., number of tuples, denoted as n_{tpl}, checkpointing frequency, etc. By setting the checkpointing frequency (n_{cp}) to be larger than the number of groups, denoted as n_g, we can ensure that the checkpoint logs are in sequential order. In other words, tuples from the previous checkpoint appear before the next checkpoint. This is because every group updates all tuples stored in all table servers for each iteration, and so the version difference between tuples is at most n_g. When we do the current checkpointing, all parameters' version is at least $v_{cp} - n_g > v_{cp} - n_{cp} = v'_{cp}$ (v'_{cp} is the previous checkpoint version). In other words, the previous checkpointing has completed. With this property, we can find the tuples of the last checkpoint by simply checking the last $2 * n_{tpl}$ tuples in the checkpoint file. The checkpoint version may be too old (if n_{cp} is large) in terms of parameters on other table servers. To avoid large inconsistency that may affect the training convergence, we rollback all table servers to the last checkpoint version, i.e., the same version as the recovered table.

6. EXPERIMENT STUDY

In this section we analyze the performance of SINGA by training real-life deep learning models. First, we study the training cost of different parallelization approaches using synchronous SGD. Second, we analyze the training cost using asynchronous SGD. Finally we evaluate the effect of the number of table servers on performance, and studies the impact of table servers failure recovery. We use OpenBlas [3] and SIMD (i.e., SSE) [4] to accelerate numeric computations. We also use some code from Caffe [13], which is an open source project for training deep learning models on a single node.

6.1 Experiment Setup

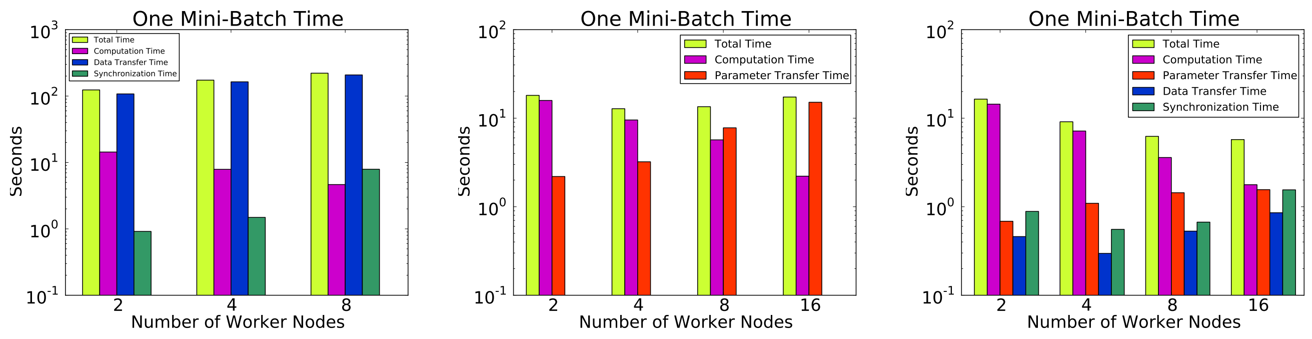
The experiments are conducted on an in-house cluster with 72 nodes hosted on two racks. Every rack has a 1 Gbps switch to connect intra-rack nodes. Inter-rack nodes are connected by a 10 Gbps cluster switch. Each node has a quad-core Intel Xeon 2.4GHz CPU, 8GB memory and two 500 GB SCSI disks.

6.2 Benchmark Task and Dataset

We use the benchmark from Large Scale Visual Recognition Challenge 2012 (ILSVRC2012) [28]. It has about 1.2 million training images from ImageNet, where each image has a label from 1000 categories. The task is to predict the labels of 100,000 images in a test dataset (from ImageNet). We use the Deep Convolution

[3]http://www.openblas.net/

[4]https://gitorious.org/arraymath



(a) Time Cost of Model Parallelism (b) Time Cost of Data Parallelism (c) Time Cost of Hybrid Parallelism

Figure 7: Time Cost Analysis of Different Parallelism Approaches VS. Number of Worker Nodes.

Neural Net (DCNN) [15] from the winner group in our experiment, which consists of 5 convolutional layers, 3 fully connected layers and one softmax layer as the output layer. The model has about 60 million parameters and is trained by mini-batch SGD using BP to compute the gradients. Setting the mini-batch size to be 256, this model occupies about 4GB memory to store the feature vectors, gradient vectors , parameters and gradients of parameters of all layers.

6.3 Training by Synchronous SGD

We analyze the time costs of three different parallelization approaches, namely the pure model parallelism, the pure data parallelism and the optimized hybrid parallelism for training one mini-batch of 256 images. Figure 7 shows the time costs by varying the number of worker nodes. We use only one group in this set of experiment. Test with different number of groups is conducted using asynchronous SGD in the next section. In this experiment, we fix the number of server nodes to be 7. The training time on a single node is 27.3 seconds using Caffe [13]. Before discussing the following figures, we explain the time measurements shown in the figures. All time measurements are averaged by running one mini-batch 10 times. The total time and computation time are easy to understand. For the parameter transfer time, we measure it by timing the *get*, *collect* and *update* operations. *get* and *update* are non-blocking, while *collect* would be blocked until the parameters requested by *get* arrive. This blocking time includes the network transfer time and the processing time at the table server. The processing would be blocked at the server side if some workers from the same group have not sent gradients for the same parameter(s) (See Section 5.4). The data transfer time measures the time to fetch (and put) remote (sub) feature vectors due to model partition. The synchronization time is the blocking time from the Barrier operation to synchronize the workers from the same group (See Sections 3 and 4.4).

Figure 7a shows the time profiling for model parallelism. As the number of worker nodes increases, the total time does not decrease; instead, it increases. This is because the time for transferring data across workers increases as shown in the figure. The *ComputeFeature* function called by the upper layers would fetch the feature vectors of lower layers. When model partition is used, the feature vectors of the lower layers are partitioned onto multiple workers. It means that remote fetches would be triggered if the required (sub) feature vectors are on other workers. Although the computation time cost drops when the computation load is partitioned onto more worker nodes. The total time cost is dominated by the data (feature vector) transfer time. In addition, the synchronization time increases when more workers are working together in one group, this is in accordance to Equation 5. According to Section 4.2, parameters are updated locally for pure model parallelism. There is no parameter transfer time, i.e., time of fetching parameters from the table servers and sending updates (gradients) to the table servers.

Figure 7b displays the time cost of data parallelism. By adding more workers, we can see that the total time decreases when the number of workers are small. With more than 4 worker nodes, the total time increases. This is because when the number of workers is small, the computation load on each worker is large. The computation time is larger than other time. By adding more workers, we reduce the computation load on each worker. Therefore the total time decreases. However, for data parallelism, every worker holds a whole copy of the model parameters. By adding more workers, the amount of parameters (and gradients) being transferred increases too. With more and more workers, the parameter transfer cost would finally bypass the computation cost and dominates the total cost as shown in the figure. For data parallelism, workers do not communicate with each other. Therefore, there is no data transfer cost or synchronization cost.

Figure 7c depicts the time cost of the hybrid parallelism approach. We can see that this approach has better scalability than the other two approaches. The total time decreases when more workers are added. We observe that hybrid parallelism can be up to 20 times faster than pure model parallelism and 5 times faster than Caffe running on single node. This is because, on the one hand, the computation time decreases with the computation load being partitioned onto more workers. On the other hand, both the data transfer time and the parameter transfer time increase slowly. We do model partition only for the layers that have small feature vectors and large size of parameters. These parameters are partitioned (not replicated) onto workers. Therefore the total time of parameter transfer increases slowly by adding workers. Although model parallelism introduces data (feature vector) transfer cost, this cost would be small as long as the feature vectors are small. Similarly, the overhead of pure data parallelism would be small if the size of parameters of the whole mode is small. However for deep learning models, the parameter size is usually large, which makes it not practical to deploy pure data parallelism.

6.4 Training by Asynchronous SGD

From Figure 7c, we can see that the benefit of distributing worker load onto multiple workers would be lost when the group size, i.e., number of workers of one group, is large (e.g., larger than 16). The overhead from synchronization and communication increases

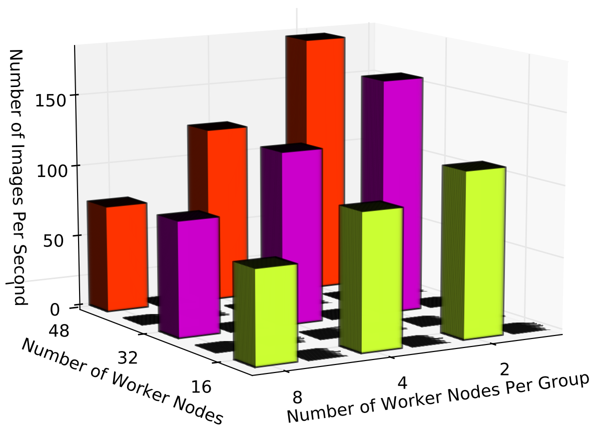


Figure 8: Throughput (In terms of images processed per second)

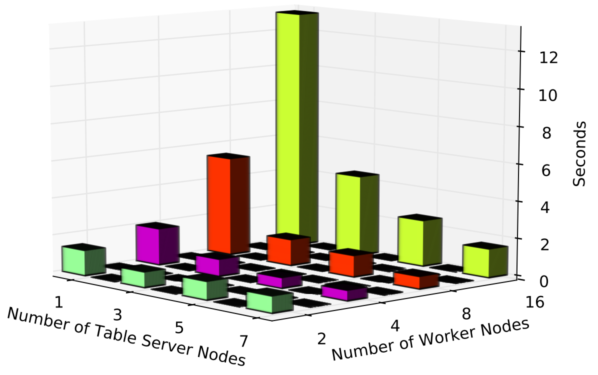


Figure 10: Parameter Transfer Time VS Number of Worker Nodes and Table Server Nodes

with larger group size. To make the system scalable, we apply the asynchronous SGD which relaxes the synchronization restrictions among groups. Hence, we can train multiple groups concurrently.

Figure 8 shows the throughput of SINGA in terms of number of training records (i.e., images) being processed in one second. We vary the total number of workers and the group size. we observe that by adding more workers to the system, the throughput increases. The highest throughput is 180 images per second, which is 20 times larger than that of Caffe.

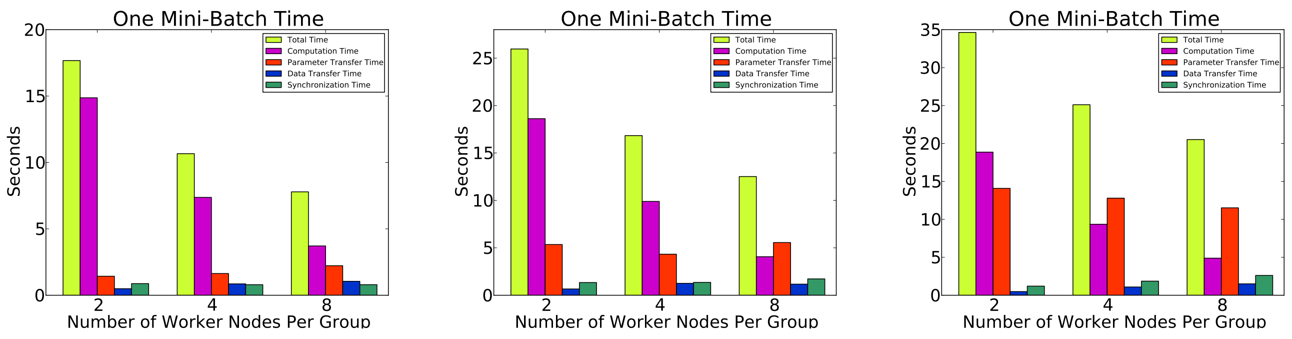
The detailed time profiling is shown in Figure 9. For each set of workers, we vary the number of group size. Figure 9a shows the detailed time costs for 16 workers with different group size. We can see that when the group size is doubled, the computation time halves. Since the main cost is from computation, the total time reduces when the computation cost decreases. The overhead increases when the group becomes larger. For the parameter transfer time, it increases because the when the group size is large, the server needs to wait for more workers' gradients (i.e., longer wait) to conduct the update and then reply to the *get* request. Therefore, with larger group, the get request (i.e., the collect operation) would be blocked for a longer time. The total parameter transfer time is thus larger. However, the parameter transfer time in Figure 9c drops when the group grows. The reason is that with larger groups, there are fewer copies of model parameters, which alleviates the busy network resulted from communication from too many (i.e., 48) workers. The data transfer time increases because there are more (sub) feature vectors being fetched in larger groups, which is in accordance to Equation 5. Since there are more workers to wait to reach the barrier, the synchronization time also increases. Figure 9b and 9c show similar trends for the time costs. However, we see an effect on the parameter transfer time discussed above. This is because more workers are used in the experiments in Figures 9b and 9c, which results in larger overhead as compared to the result in Figure 9a which involves fewer workers. But, with more workers, we can train more groups concurrently. The resultant effect is that the total throughput increases.

6.5 Evaluation of Table Servers

Since all parameters are maintained by the table servers and all workers communicate with them, it is important to analyze the effect of table servers on the training time and its failure recovery. In this set of experiment, similar to Section 6.3, we set all workers as one group and run the synchronous SGD with hybrid parallelism. But to make the table servers busy enough, we do not do local update for parameters. In other words, all parameters are stored and updated by the table servers.

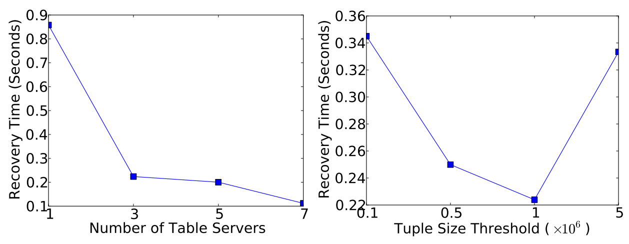
Figure 10 shows the training cost by varying the number of workers and the number of table servers. Generally, when the number of table servers increases, the parameter transfer time decreases. The is because when the tuples are distributed to more servers, the workload (i.e., *get* and *update* request) of one server is reduced. But when the number of workers is small (e.g., 2), the workload of table server is small enough. Therefore there is not much difference in the parameter transfer time when varying the number of table servers as shown in the figure. When the number of worker nodes increases, there are more parameters and more requests. The workload of the table servers also increases. Hence, the parameter transfer time increases. But the parameter transfer time drops when the number of workers increases from 2 to 4. The reason is that, when the are fewer workers, each worker holds a larger amount of parameters of the model. Then every worker needs to transfer more parameters and gradients with the servers. With more than 4 workers, each worker's load becomes small, but the load of server side becomes heavy. That is why the parameter transfer time increases when there are more workers.

Next we study the failure recovery of the table servers. We use the parameters from the benchmark model, i.e., the DCNN. The largest parameter object has about 38 million parameters. The total number of parameters is 62 million. We store the parameters and their gradients in the table servers, which take about 470 Mega Bytes in total. In the recovery evaluation, we test the effects of the number of table servers and the tuple size threshold, i.e., the largest number of parameters in one tuple. We do recovery after checkpoint for 3 times. Figure 11a shows that with more table servers, the recovery time drops. This is because with more table servers, the table partition maintained by each server is smaller. During recovery, each table server reads less data from the disk. Therefore the time decreases. Figure 11b shows the effect of the tuple size threshold, we vary the threshold from 0.1 million parameters per tuple to 5 million parameters per tuple. When the threshold increases, the total number of tuples decreases. This reduces the number of put (i.e., insert) operations, which dominates the total recovery time when there are many tuples (The disk read time is small because the total table size is not that large). Consequently, the recovery time drops. When the threshold is too large, e.g., 5 million, the cost of allocating memory and copy data from buffer is large, which makes the total recovery time even larger than that of a mild threshold. That is why there is a jump in the figure when the tuple size threshold changes from 1 million to 5 million.



(a) 16 Workers (b) 32 Workers (c) 48 Workers

Figure 9: Time Cost Analysis of Different Number of Workers VS. Number of Worker Nodes Per Group.



(a) Effect of Table Servers (b) Effect of Tuple Size

Figure 11: Recovery Time.

7. RELATED WORK

Large scale deep learning models [15, 5, 3, 1] have attracted big interest from both industry and academia due to its success in various areas. However, the training of large scale deep learning models is non-trivial. Larger models (in terms of model parameters) take larger memory and require more training data to reduce over-fitting. Complex numeric operations make the training computation intensive. There have been two approaches to tackle this problem. One is optimizing the training by exploiting specific hardwares, e.g., GPUs [3, 24, 14, 33] for (specific) deep learning modes [15]. The other approach uses distributed platforms consisting of commodity computers to distribute the training workload and parallelize the training. SINGA belongs to the second category.

General distributed platforms such as MapReduce, Spark [34], and GraphLab [21] lack optimizations specific to deep learning models. For example, [36] uses MapReduce to train deep learning models. It runs a set of mappers to compute gradients of parameters, which are shuffled to reducers to update the parameters. The parallelization strategy is the data parallelism we discussed in Section 4.1, which performs poorly when the number of parameters is large as reported in our experiment. Moreover, MapReduce is not suitable for machine learning problems involving iterative computations [19].

Recently, several distributed deep learning systems have been developed, e.g., DistBelief [5] from Google and Adam [1] from Microsoft. At a high level, SINGA shares the same design with these systems. They all adopt the parameter server framework [20] to maintain the model parameters, and partition the model and data onto a cluster of commodity computers to parallelize the computation. While DistBelief is scant on the system's detail and focuses more on benchmarking specific algorithms, Adam takes a more system-driven approach to deep learning but does not optimize on the parallelism approaches which brings different communication overhead for different deep learning models. SINGA optimize on this by analyzing the cost of different parallelism approaches, and selecting the one with overall minimal overhead. SINGA also differs to these framework in our attempt at a flexible and easy to use programming model which makes it easy for implementing, testing and deploying deep learning systems.

Parameter server is an important component of SINGA. General parameter server [20] is optimized at the system level for general machine learning tasks but does not consider all optimization techniques we described in this paper. For example, SINGA use asynchronous *get* and *collect* operation to parallelize the BP computation and parameter transferring. The consistency control of SINGA is customizable to support both synchronous SGD and asynchronous SGD.

8. REFERENCES

[1] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, pages 571–582, Broomfield, CO, Oct. 2014. USENIX Association.

[2] D. C. Ciresan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep big simple neural nets excel on handwritten digit recognition. *CoRR*, abs/1003.0358, 2010.

[3] A. Coates, B. Huval, T. Wang, D. J. Wu, B. C. Catanzaro, and A. Y. Ng. Deep learning with cots hpc systems. In *ICML (3)*, pages 1337–1345, 2013.

[4] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. P. Kuksa. Natural language processing (almost) from scratch. *JMLR*, 12:2493–2537, 2011.

[5] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. W. Senior, P. A. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, pages 1232–1240, 2012.

[6] L. Deng, M. L. Seltzer, D. Yu, A. Acero, A. rahman Mohamed, and G. E. Hinton. Binary coding of speech spectrograms using a deep auto-encoder. In *INTERSPEECH*, pages 1692–1695, 2010.

[7] J. C. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, 2011.

[8] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Learning hierarchical features for scene labeling. *IEEE Trans. Pattern*

Anal. Mach. Intell., 35(8):1915–1929, 2013.

[9] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2Nd Ed.): Portable Parallel Programming with the Message-passing Interface*. MIT Press, Cambridge, MA, USA, 1999.

[10] G. Hinton. A Practical Guide to Training Restricted Boltzmann Machines. Technical report, 2010.

[11] G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002.

[12] G. E. Hinton, S. Osindero, and Y. W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006.

[13] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[14] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114, 2012.

[16] Q. V. Le, M. Ranzato, R. Monga, M. Devin, G. Corrado, K. Chen, J. Dean, and A. Y. Ng. Building high-level features using large scale unsupervised learning. In *ICML*, 2012.

[17] Y. LeCun, L. Bottou, G. B. Orr, and K. Müller. Effiicient backprop. In *Neural Networks: Tricks of the Trade, this book is an outgrowth of a 1996 NIPS workshop*, pages 9–50, 1996.

[18] H. Lee, R. B. Grosse, R. Ranganath, and A. Y. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *ICML*, page 77, 2009.

[19] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using mapreduce. *ACM Comput. Surv.*, 46(3):31, 2014.

[20] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, pages 583–598, Broomfield, CO, Oct. 2014. USENIX Association.

[21] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[22] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[23] T. Paine, H. Jin, J. Yang, Z. Lin, and T. S. Huang. Gpu asynchronous stochastic gradient descent to speed up neural network training. *CoRR*, abs/1312.6186, 2013.

[24] T. Paine, H. Jin, J. Yang, Z. Lin, and T. S. Huang. Gpu asynchronous stochastic gradient descent to speed up neural network training. *CoRR*, abs/1312.6186, 2013.

[25] A. rahman Mohamed, G. E. Dahl, and G. E. Hinton. Acoustic modeling using deep belief networks. *IEEE Transactions on Audio, Speech & Language Processing*, 20(1):14–22, 2012.

[26] A. rahman Mohamed and G. E. Hinton. Phone recognition using restricted boltzmann machines. In *ICASSP*, pages 4354–4357, 2010.

[27] B. Recht, C. Re, S. J. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, pages 693–701, 2011.

[28] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. Imagenet large scale visual recognition challenge, 2014.

[29] A. W. Senior, G. Heigold, M. Ranzato, and K. Yang. An empirical study of learning rates in deep neural networks for speech recognition. In *ICASSP*, pages 6724–6728, 2013.

[30] P. Sermanet, K. Kavukcuoglu, S. Chintala, and Y. LeCun. Pedestrian detection with unsupervised multi-stage feature learning. In *CVPR*, pages 3626–3633, 2013.

[31] I. V. Tetko, D. J. Livingstone, and A. I. Luik. Neural network studies. 1. comparison of overfitting and overtraining. *Journal of Chemical Information and Computer Sciences*, 35(5):826–833, 1995.

[32] W. Wang, B. C. Ooi, X. Yang, D. Zhang, and Y. Zhuang. Effective multi-modal retrieval based on stacked auto-encoders. *PVLDB*, 7(8):649–660, 2014.

[33] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato. Multi-gpu training of convnets. *CoRR*, abs/1312.5853, 2013.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 15–28, 2012.

[35] C. Zhang and C. Re. Dimmwitted: A study of main-memory statistical analytics. *PVLDB*, 7(12):1283–1294, 2014.

[36] K. Zhang and X. Chen. Large-scale deep belief nets with mapreduce. *IEEE Access*, 2:395–403, 2014.