

# SINGA: Putting Deep Learning in the Hands of Multimedia Users

Wei Wang<sup>†</sup>, Gang Chen<sup>§</sup>, Tien Tuan Anh Dinh<sup>†</sup>, Jinyang Gao<sup>†</sup>  
Beng Chin Ooi<sup>†</sup>, Kian-Lee Tan<sup>†</sup>, Sheng Wang<sup>†</sup>

<sup>†</sup>School of Computing, National University of Singapore, Singapore

<sup>§</sup>College of Computer Science, Zhejiang University, China

<sup>†</sup>{wangwei, dinhhta, jinyang.gao, ooibc, tankl, wangsh}@comp.nus.edu.sg, <sup>§</sup>cg@zju.edu.cn

## ABSTRACT

Recently, deep learning techniques have enjoyed success in various multimedia applications, like image classification and multi-modal data analysis. Two key factors behind deep learning's remarkable achievement are the immense computing power and the availability of massive training dataset, which enable us to train large models to capture complex regularities of the data. There are two challenges to overcome before deep learning can be widely adopted in multimedia and other applications. One is usability, namely the implementation of different models and training algorithms must be done by non-experts without much effort. The other is scalability, that is the deep learning system must be able to provision for a huge demand of computing resources for training large models with massive datasets. To address these two challenges, in this paper, we design a distributed deep learning platform (called SINGA) with easy-to-use programming model and good scalability. Our experiments with training deep learning models for real-life multimedia applications show that SINGA is a practical system.

## Categories and Subject Descriptors

H.3.3 [INFORMATION STORAGE AND RETRIEVAL]: Information Search and Retrieval—*Retrieval Models*; I.2.6 [ARTIFICIAL INTELLIGENCE]: Learning—*Parameter Learning*

## General Terms

Design, Experimentation

## Keywords

Deep Learning, Multimedia Application

## 1. INTRODUCTION

Recent years have witnessed successful adoption of deep learning in various multimedia applications, such as image and video classification [14, 26], content-based image retrieval [22], music recommendation [24] and multi-modal data analysis [23, 8, 30]. Deep learning refers to a set of feature learning models which consist of multiple layers in which different layers learn different levels of abstractions (or representations) of the raw input data [15]. It has been regarded as a re-branding of neural networks developed twenty years ago, since it inherits many key neural networks techniques and algorithms. However, deep learning exploits the fact that high-level abstractions are better at representing the data than raw, hand-crafted features, thus achieving better performance in learning. Its recent resurgence is mainly fueled by higher than ever accuracy obtained in image recognition [14]. Two key factors

behind deep learning's remarkable achievement are the immense computing power and the availability of massive training datasets, which together enable us to train large models to capture the regularities of data more efficiently than twenty years ago.

There are two challenges in bringing deep learning to wide adoption in multimedia applications (and other applications for that matter). The first challenge is *usability*, namely the implementation of different models and training algorithms must be done by non-experts without much effort. Many deep learning models exist, and different multimedia applications may use different models. For instance, the deep convolutional neural network (DCNN) is suitable for image classification [14], deep multi-layer perceptron [3] for recognizing handwritten digits, and auto-encoders for multi-modal data analysis [23, 8, 30]. Most of these models are too complex and costly to implement from scratch. An example is the GoogleNet model [20] which comprises 22 layers of 10 different types. Training algorithms are also intricate in details, for instance Back-Propagation [17] algorithm is notoriously difficult to debug.

The second challenge is *scalability*, that is the deep learning system must be able to provision for a huge demand of computing resources for training large models with massive datasets. It has been shown that larger training datasets and bigger models lead to better accuracy [3, 15, 20]. However, memory requirement for a deep learning model may exceed the memory capacity of a single CPU or GPU. In addition, computational cost of training may be unacceptably high for a single, commodity server. For instance, it takes 10 days [27, 18] to train the DCNN [14] with 1.2 million training images and 60 million parameters using one GPU<sup>1</sup>.

Addressing both challenges requires a distributed training platform that supports various deep learning models, an easy-to-use programming model (similar to MapReduce [7], Spark [28] and epiC [11] in spirit), and good scalability. A recent deep learning system, called Caffe [10] addresses the first challenge but falls short at the second challenge (it runs on single nodes only). Some systems support distributed training, for examples [18, 27, 13], but they are model specific and do not generalize well to other models. General distributed platforms such as MapReduce [7] and Spark [28] are possible solutions for the second challenge. However, they have been designed for generic machine learning. As a result, they lack both the programming model and system optimizations specific to deep learning, affecting the system usability and scalability. Recent specialized distributed platforms [6, 5, 1] exploit deep learning specific optimizations and hence are able to achieve a high training throughput. However, they forgo usability issues, i.e. the platforms are closed-source and are not easy to use.

<sup>1</sup>According to the authors, with 2 GPUs, the training still took about 6 days.

In this paper, we present our effort in bringing deep learning to the masses. In particular, we design and implement a distributed deep learning platform, called SINGA, which tackles both usability and scalability challenges at the same time. SINGA runs on commodity servers, and it is accessible even to non-experts by providing a simple programming model. SINGA's simplicity is based on the observation that both structures and training algorithms of deep learning models can be expressed via a simple abstraction, i.e., the neuron layer. In SINGA, the user defines and combines layers to construct the neural network structure, and the runtime takes care of distributed execution tasks such as partitioning, synchronization and communication. For scalability, SINGA adopts two key techniques. First, it integrates both existing parallelism approaches namely data parallelism to partition the training dataset and model parallelism to partition the model. This *hybrid parallelism* approach can lower communication and synchronization overhead thus enable the system to scale. Second, SINGA training architecture is flexible and can be easily customized. This flexibility allows users to switch between different architecture at different scales. For instance, the popular Downpour architecture [6] contains inherent communication and synchronization bottlenecks, which can be re-configured to a replicated server architecture to relieve the bottlenecks and achieve better scalability.

In summary, we make the following contributions:

1. We present a distributed deep learning platform called SINGA which has been designed for supporting deep learning in multimedia and other applications. The platform offers a simple programming model based on the layer abstraction. It has built-in supports for various models and training algorithms, and users can easily customize them by defining new layers and combining them through simple configurations.
2. SINGA is designed for distributed execution over commodity servers, and it is scalable. It handles the underlying communications and synchronizations transparently, and employs several optimizations to improve training performance.
3. We discuss the implementation of several multimedia applications using SINGA. We report the experimental results which show that all applications achieve the same level of accuracy as previously reported.

The rest of this paper is organized as follows. In the next section, we provide some background on training deep learning models. The programming model of SINGA is introduced in Section 3. Section 4 describes the implementations of three multimedia applications using SINGA programming model. We discuss system implementation details in Section 5. Experiments and related works are presented in Section 6 and Section 7 before we conclude.

## 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

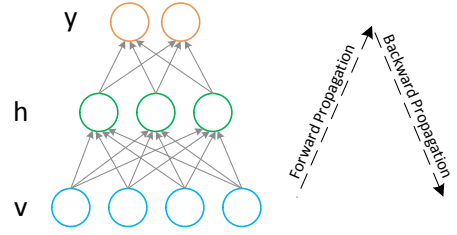


Figure 1: Training Multilayer Perceptron using Back-Propagation

(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. The bottom layer accepts the **input data** (input feature vector), denoted as  $\mathbf{v}$ <sup>2</sup>, 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,  $W$  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.  $\sigma$  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.

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

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

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 connections between layers. 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, i.e., predictions that are closer 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 accuracy of a 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 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 input feature vector  $\mathbf{v}$  and the model parameters denoted as  $\Theta$ . The

<sup>2</sup>In this paper, we use bold lowercase symbols for vectors, upper-case symbols for matrices or sets

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

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

goal of the training procedure is to find the assignment of  $\Theta$  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 related to the classification error, and find the parameters that optimize 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. If the training record has no ground truth, i.e.,  $\mathbf{t}$  is empty, the training is unsupervised.

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

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  $\Theta$ . 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 [19] are widely used for training deep learning models. In our system, we support both the basic SGD and its extensions. The basic SGD 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.

There are two typical approaches to calculate the gradients, namely Contrastive Divergence (CD) [9] 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. SINGA manages the training flow, i.e., 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

Layer	Configuration
member: data // list of feature blobs param //list of parameter objects type, name, shape // meta info srclayer // source layers method: Setup(srclayers, conf) SetupAfterPartition(srclayers, conf, shape) ComputeFeature(srclayers) ComputeGradient(srclayers)	a list of layers, each includes: meta info: layer name, type, size, etc. connection info: source layer names

Figure 2: Basic Data Structures

In this section, we introduce the programming model of SINGA. Users can implement their deep learning based applications 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. Hence, we propose a base Layer class and implement BP and CD using it. Users only need to override the base Layer class to implement their own layers and models. SINGA conducts the training by calling BP/CD and updating the parameters according to SGD algorithms.

Figure 2 shows the base data structures used to conduct the training. The base Layer class has a *data* field which is a list of memory blobs to store the features (and the gradients of the loss w.r.t. the features used in BP). Besides the *data* field, there is a *param* field which includes a list of parameter objects. Each parameter object has two base fields, one for its values and the other for its gradients, denoted as *val* and *grad* respectively. There are also some meta data fields, like layer *type*, *name* and *shape*. The *type* field determines the transformation of features (e.g., Equation 1). The *name* field is a string to identify the layer in the neural network. The *shape* field represents the shape of the *data* (e.g., the length for 1-dimensional feature, and height/width for 2-dimensional feature). The connections between layers are stored in the *srclayer* field, which is a list of Layers. Each layer in *srclayer* represents the source of a connection. If the connection between two layers are undirected (e.g., in RBM), they will appear in each other's *srclayer* field.

The base Layer class has four methods. The *Setup* function reads information from source layers and the user configuration to set the shape, allocate memory for data blobs and initialize parameters. Considering that the layer may be partitioned by SINGA due to model or data partitioning, users need to override the *SetupAfterPartition*, which re-allocates the memory according to the partitioned shape. 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 on two different nodes) transparently at runtime. The *ComputeFeature* function computes the features by transforming features from the source layers like Equation 1. The *ComputeGradient* function computes the gradients of parameters associated with this layer. For the BP algorithm, it would also compute the gradients of the loss w.r.t. to features in the source layers. Users can override the *ComputeFeature* function to implement their own feature transformation logics. In SINGA, we also extend the base Layer class to implement some utility layers for data and model partition. For example, we realize a pair of layers to transfer features (or gradients) between two connected layers that are partitioned onto two nodes.

To implement specific a deep learning model, users first implement the layers by extending the base layer class. Some common layers, like convolution layer, pooling layer, etc., have been imple-

HiddenLayer: Layer	Configuration
<pre> member:   fea, grad// feature and gradient extracted from data field   W, b // parameters extracted from param field   name, shape, type, srclayer // inherited from Layer method:   Setup(srclayers, conf){     // read info from conf and srclayers,     // setup the shape of fea and grad     // setup the shape of W, b   }   ComputeFeature(srclayers){     fea=logistic(dot(srclayers[0].fea, W.data)+b.data); // Equation 1   }   ComputeGradient(srclayers){     srclayers[0].grad=fea*(1-fea)*dot(grad, W.data.transpose());     W.grad=data*(1-fea)*dot(src[0].fea.transpose(), grad);     b.grad=data*(1-fea)*grad;   } </pre>	<pre> layer:   name: "data"   type: kDataLayer layer:   name: "label"   type: kLabelLayer layer:   name: "hidden"   type: kHiddenLayer   srclayer: "data layer"   shape: 3 // 3 neurons layer:   name: "softmax"   type: kSoftmaxLossLayer   srclayer: "hidden",     "label" </pre>

Figure 3: Pseudo Code for the MLP Example in Figure 1

mented in SINGA as built-in layers. Next, users define the model structure in a configuration file which specifies the configurations for each layer and the connections between layers. SINGA starts with parsing the configuration file, then it creates the model by instantiating the corresponding layer objects and connecting them. Data partitioning or model partitioning is conducted against the layers. After that it re-setups each layer. The SGD training is then started, which uses BP/CD to calculate the parameter gradients. The BP/CD algorithm internally calls the *ComputeFeature* and *ComputeGradient* functions of each layer.

We use the MLP example from Figure 1 to explain the programming model. We extend the base Layer class to create a HiddenLayer class for the hidden layer. Figure 3 shows the pseudo code for the HiddenLayer class. 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. APPLICATIONS

In this section, we show how to use SINGA to implement real-life multimedia applications. Three popular applications are presented, i.e., handwritten digits recognition, natural image classification and multi-modal retrieval.

### 4.1 Handwritten Digits Recognition

Handwriting is one of the main methods for communication and recording information in our daily life. Recognizing handwritten, such as notes in a PDA, postal addresses on envelopes and amounts in bank checks, is a major application of deep learning algorithms [3, 4]. In this section, we implement a simple big multilayer perceptron [3] to recognize handwritten digits.

The model [3] is simple as it has only two types of hidden layers. However, it is big because all hidden layers are fully connected (like Figure 1) which means each hidden layer is associated with a big parameter matrix. In fact, it has about 10 million parameters.

Figure 4a shows the model structure constructed by SINGA. SINGA provides a couple of data layers to read data records from disk (DiskLayer) or databases (DBLayer). We use the MNIST handwritten dataset in our experiment, which is provided as binary files. In preprocessing stage, we convert the data into SINGA defined records and store them in a database which can then be loaded by the data layer. The data records loaded by the DBLayer are parsed by two parser layers, namely the LabelLayer and the MnistLayer. The LabelLayer extracts the ground truth label, i.e. the

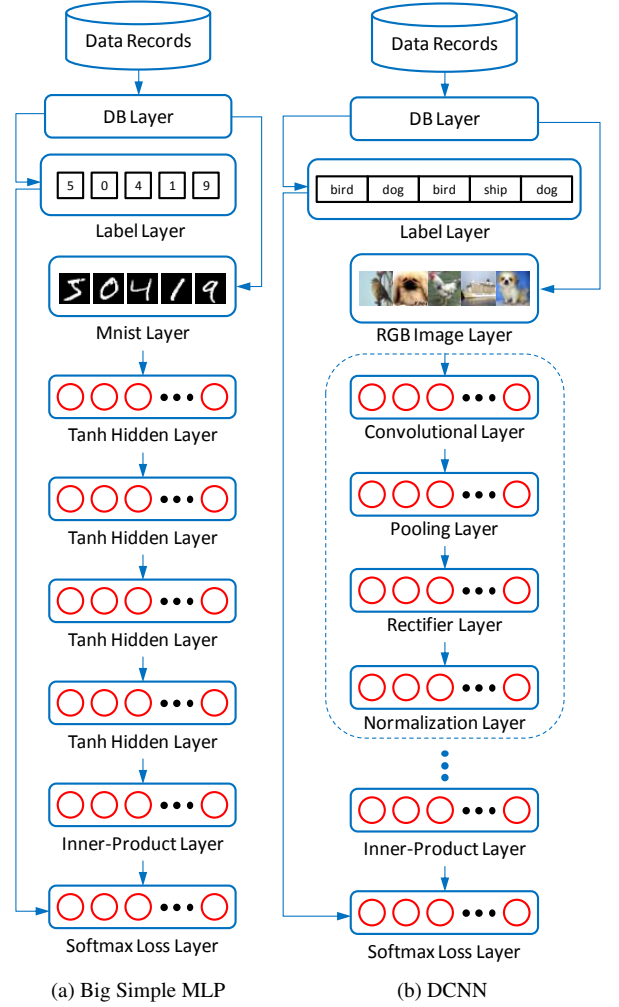


Figure 4: Network Structures Created by SINGA

digit, of each image, while the MnistLayer extracts the pixel values of each image. The TanhHiddenLayer transforms the features as Equation 1 except that the logistic activation function is replaced by the tanh function. Hence, the implementation of TanhHiddenLayer is similar to the code shown in Figure 3. The InnerProductLayer is also similar to Equation 1 except that there is no activation function. SINGA uses BP to calculate the gradients of parameters associated with the hidden layers. The *ComputeFeature* functions of each hidden layer are called from the top layer to the bottom layer. Next, the *ComputeGradient* functions are called inversely from the bottom layer until the parser layers. After that, the gradients of parameters are updated (locally for single node training or remotely for distributed training).

During on-line recognition, we replace the DBLayer with a StreamLayer which reads records from the network. Since the records are not labeled, we remove the LabelLayer. The extracted pixel values are then forward to the bottom layer which assigns a probability for each digit.

### 4.2 Natural Image Classification

Natural image classification is a popular and challenge multimedia task. In recent years, deep learning techniques have made break-through improvement on it [14, 20]. In this section, we im-



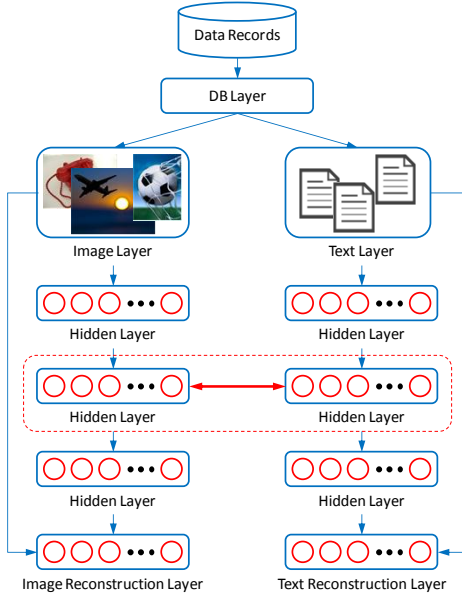


Figure 5: Multi-modal Stacked AutoEncoders

plement a deep convolutional neural network (DCNN) similar to the AlexNet [14] for classifying tiny natural images from the CIFAR10 dataset [12].

The model structure constructed by SINGA is presented in Figure 4b. Like in digit recognition, we pre-process the raw dataset by converting it into SINGA defined records and storing them in a database. An RGBImageLayer is used to parse the pixel values for each image. The DCNN has 3 sets of layers, each of which consists of a convolutional layer, a rectifier layer, a pooling layer and a normalization layer. Different layers apply different transformation functions on the features [14]. There is an inner-product layer whose neurons are fully connected with its source layer like the connection shown in Figure 1. Most of the parameters of the DCNN are associated with the inner-product layer. We will discuss the partitioning of this layer to minimize the communication overhead from model partitioning in Section 5. Its training and online prediction is similar to Section 4.1.

### 4.3 Multi-Modal Retrieval

Multi-modal retrieval is a hot multimedia research problem. In [23, 8], they use stacked auto-encoders to learn a common latent representation for data from different modalities. Multi-modal retrieval is then conducted against the latent representations. Since there are multiple modalities involved in the learning model, the model structure are more complex than the other applications we discussed above.

Figure 11 displays the model structure constructed by SINGA for the MSAE (Multi-modal Stacked AutoEncoders) from [23]. There are two sets of layers, one for the text modality and the other for the image modality. ImageLayer and TextLayer parse image and text features respectively from semantically relevant image-text pairs (e.g., the visual feature and the tag feature of an image). Hidden layers transform the features as Equation 1. The bottom layers are special layers which reconstruct the original image feature and text feature. During training, the BP algorithm forwards the features between layers along the arrow as shown in Figure 11. The learning loss measures the reconstruction error for each modality and the difference of the latent representations (generated from the

hidden layers connected by the bidirectional arrow in the figure) of a semantically relevant image-text pair. By minimizing this loss, we make the latent representations capture the semantics of intra-modality and inter-modality. After that, the BP algorithm backward propagates the gradients calculated from the reconstruction error. The gradients calculated from the difference of the latent representations are aggregated when the back-propagation reaches the corresponding layers.

After training, the latent features for images and text are extracted separately. Multi-modal retrieval are conducted against the latent features. For example, we can use the latent feature of one image to search similar text documents against the latent features of the text documents.

## 5. IMPLEMENTATION

In this section, we describe the implementation details of SINGA, including the system architecture and optimizations.

### 5.1 System Architecture

SINGA provides flexible system architectures for distributed training. Since different architectures may incur different training overhead and have different impacts on the convergence rate, users can select the best architecture for their own models. Figure 6 shows an example of three architectures supported in SINGA.

The Downpour [6] (parameter server) architecture is shown in Figure 6a, in which a worker (resp. server) represents a group of worker (resp. server) nodes. The training dataset is partitioned among the worker groups. Each group contains a complete model structure which is partitioned within the group. The group runs BP/CD algorithm to compute the gradients for all model parameters. The parameter updates are then sent to the parameter servers where each server maintains a part (or shard) of the model parameters. The servers update the parameters and send them back to the worker groups. This server-side update can be synchronous or asynchronous. In the synchronous mode [25], the server applies updates on a parameter only after it has received the updates from all worker groups. The convergence rate of this mode is the same as training on a single node but with a bigger mini-batch. In the asynchronous mode [6, 1], the server applies an update immediately after receiving it from any worker group. The convergence in this case would not be the same as in single-node training, because multiple workers are working on different versions of the parameters [29]. Regardless of the update schemes, one problem with this architecture is that the server nodes become network bottleneck when the model size increases, thus limiting the system's scalability. Specifically, each server has to receive (and send)  $s = p * n / m$  data items (float values) per training iteration, where  $p$  is the total parameter size,  $n$  is the number of worker groups and  $m$  is the number of servers. It can be seen that  $s$  increases linearly with  $n$ , thus in a commodity cluster a small number of worker groups may easily saturate the network. Adam [1] employs powerful hardware to relieve this bottleneck. Without access to such hardware, two other simple approaches can be used. First, in [31] the workers synchronize with the servers only after several iterations as opposed to after every iteration. However, it may lead to lower convergence rate. The second approach is to increase the number of servers. For example, DeepImage [25] launches one server process on each worker node. A drawback of this solution is that each worker has to communicate with all others, hence the synchronization time increases.

A replicated server architecture, as illustrated in Figure 6b, can help relieve the workload of a single server group. There are mul-

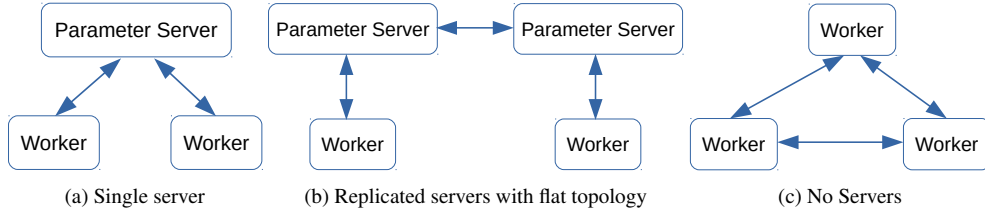


Figure 6: Different topologies of workers and parameter servers supported in SINGA .

multiple parameter servers (more precisely, multiple groups of servers) called replicas, each maintaining an entire set of parameters. Each worker group subscribes to one server group. Let  $r$  be the number of replica servers, the network demand at each server now reduces to  $\frac{1}{r}$  of the original no-replica architecture. To ensure accuracy and convergence, replicas must exchange updates with each other. Users can configure the synchronization frequency to achieve balance between convergence and communication overhead.

Figure 6c shows an architecture with no servers, in which each worker group communicates with a set of neighboring groups to synchronize their parameters. The worker group’s communication load depends on the number of neighbors. In addition, synchronization delay is long for neighboring worker groups that are far (in terms of number of hops) from each other, which affects convergence rate [31]. On the other hand, if worker groups form a fully connected graph, they would have at least the same communication load as in the Downpour architecture.

It can be seen from the above analysis that different architectures have different impact on the communication overhead and convergence rate. Therefore, we allow users to select the architecture that is most suitable for their models. SINGA unifies the three architectures and provides a base abstraction for parameter management (PMBase) which contains two sets of functions. One set includes fetch, update operations to retrieve and send local updates (e.g. from the worker) to remote nodes (e.g., the servers). The other set includes functions for processing requests. A new architecture can be implemented by customizing PMBase. For instance, the Downpour architecture is realized by implementing a PMClient class for the workers which overrides the fetch and update functions, and a PMServer class for the servers which overrides the functions processing the fetch and update requests.

## 5.2 Partition Optimization

To enable distributed training, the common approach is to partition the training dataset among worker groups and to partition the model within each group. However, pure model partitioning is not optimal in terms of communication overhead. For example, the top layers of the DCNN (Section 4.2) have larger feature vectors and fewer parameters, while its bottom InnerProductLayer has smaller feature vectors and more parameters. Using pure model partitioning, i.e., partitioning each layer onto all workers in the same group, communication cost is large due to transferring of feature vectors (the *ComputeFeature* function fetches feature vectors from source layers to compute the features of the current layer). Given that the top layers have fewer parameters, we can apply data partitioning for these layers. More specifically, each worker holds the whole layer and computes gradients for the weights associated with this layer. Since the parameter size is small, the communication overhead caused by synchronizing with the servers is smaller than that of transferring feature vectors within the same group due to pure model partitioning.

SINGA iterates through all possible partitioning schemes for

every layer and selects one with minimal overall communication overhead. We call this *hybrid partitioning* scheme which achieves hybrid parallelism. Note that similar optimization is used in [13], but it is specific to the AlexNet model[14] and single-node, multi-core settings. With the layer abstraction, we can easily implement the hybrid partitioning in SINGA to support general distributed training. In specific, the partition algorithm generates a new network structure by splitting the original layers into sub-layers whose shapes are changed accordingly. Each sub-layer has a residence node ID. At runtime, only local layer’s *ComputeFeature* and *ComputeGradient* functions are executed. A set of utility layers are inserted to connect these sub-layers. For instance, we implement a pair of bridge layers denoted as SrcBridgeLayer and DstBridgeLayer to connect sub-layers that resident on different nodes but their original layers are connected. The *ComputeFeature* function of the SrcBridgeLayer sends feature vectors via the network to DstBridgeLayer where they are received in the *ComputeFeature* function. These bridge layers are transparent to users. Hence, users can implement their layers without worrying about the locations of layers and the underlying data transferring<sup>3</sup>.

## 6. EXPERIMENTAL STUDY

In this section, we present the results of using SINGA in real-life multi-media applications. We implemented the three applications discussed in Section 4 on SINGA, all of which achieved the same level of accuracy reported in previous studies. We demonstrated that SINGA achieves almost linear scalability in terms of training time to reach certain accuracy. We compared SINGA against the current state-of-the-art open source system, namely H2O, and found that SINGA reaches higher throughputs. Finally, we showed that the cost (in training time) of hybrid parallelism is lower than that of pure model or data parallelism.

### 6.1 Experiment Setup

We conducted the experiments over a commodity cluster with 1Gbps Ethernet. Each node has a quad-core Intel Xeon 2.4GHz CPU, 8GB memory and two 500 GB SCSI disks. We used OpenBlas<sup>4</sup>, mshadow<sup>5</sup> and SIMD to accelerate numeric computations. We also borrowed some utility code from Caffe [10] for training deep learning models on a single node. For simplicity, we use the traditional, no-replica parameter server architecture (Figure 6a).

### 6.2 Handwritten Digits Recognition Using MLP

We train the deep simple multi-layer perceptron discussed in Section 4.1 for handwritten digits recognition using MNIST [16] dataset. The dataset consists of 60,000 training images and 10,000 test images. All images are size-normalized and centered to a fixed

<sup>3</sup>For more details on the hybrid partitioning, please refer to the supplementary material.

<sup>4</sup><http://www.openblas.net/>

<sup>5</sup><https://github.com/dmlc/mshadow>

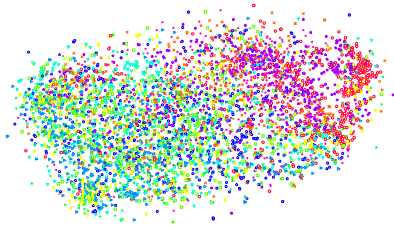


Figure 7: First convolutional layer

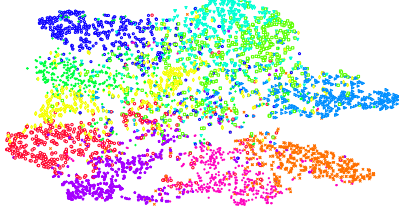


Figure 8: Inner-product layer

airplane (0)	auto mobile (1)	bird (2)	cat (3)	deer (4)	dog (5)	frog (6)	horse (7)	ship (8)	truck (9)
		0					3	3	
		0	0			3		3	0
			4						
	9				0	0			1
							9		3

Figure 9: Sample Classification Results Using DCNN on CIFAR10 dataset.

size of 28\*28 pixels. Following the training configuration from [3], we achieve 98.5% accuracy without image augmentation. Figure 10 shows samples of mis-recognized digits from the test dataset. We can see that some digits are hard to recognize. For instance, the digit '4' at the first row and second column is likely to be mistaken as '9' even by humans. Similarly the digit '5' at the second row and forth column is likely to be mistaken as '6'. There are several cases that are easy for humans but prove difficult for the model. For example, the digit '3' at the forth row and the first column is clearly written, but the model mis-recognized it as '0'.

8 <sub>9</sub>	4 <sub>9</sub>	5 <sub>8</sub>	9 <sub>3</sub>	2 <sub>1</sub>
5 <sub>6</sub>	8 <sub>3</sub>	4 <sub>4</sub>	5 <sub>6</sub>	8 <sub>3</sub>
8 <sub>3</sub>	2 <sub>3</sub>	4 <sub>4</sub>	3 <sub>7</sub>	4 <sub>4</sub>
3 <sub>0</sub>	3 <sub>9</sub>	1 <sub>1</sub>	3 <sub>5</sub>	5 <sub>5</sub>
3 <sub>8</sub>	5 <sub>0</sub>	3 <sub>3</sub>	0 <sub>5</sub>	8 <sub>6</sub>

Figure 10: Samples of Mis-Recognized Digits (number on the corner is the prediction)

### 6.3 Image Classification using CNN

This application takes as input the CIFAR-10 [12] dataset consisting of 60,000 color images. Each image is of size 32\*32 pixels, and is associated with one label from 10 categories. There are 6000 images per category, and the dataset is split into 50,000 training images and 10,000 test images.

We follow the recommended configuration<sup>6</sup>. The accuracy is the same as achieved by the original authors, that is 82% without data augmentation. Figure 7 and 8 visualize the image features from the first convolutional layer and the inner-product layer respectively by projecting them into a 2-dimensional space via tSNE [21] (refer to

the supplementary material for feature visualization of other layers). Different color represents different categories. We can see that the high-level features from the InnerProductLayer are clustered based on their semantic categories, while the low-level features are mixed. It indicates the learned high-level features capture semantics well, thus are meaningful. Figure 9 shows a sample of the classification results. For each class, we randomly sample 5 images from the test and list the classification results in one column. The mis-classified images are marked with its prediction class IDs on the right side, all other images are correctly classified. Note that the low image resolution makes it difficult even for humans to label them correctly. The model classified all images from the first class correctly, because the airplane has unique shapes and is usually surrounded by clouds in blue sky. The small sets of shape and color help improve classification results. Other classes, for example the birds, have many more shapes and colors, hence it is harder for the model to remember all such information to make correct classifications.

### 6.4 Multi-Modal Retrieval Using Auto-Encoders

The multi-modal retrieval application takes as input the NUS-WIDE dataset [2], which has about 18,000 images after removing images without tags or from non-popular categories. Each image is associated with a couple of tags, which are used as text documents. We achieve the same accuracy reported in [23] by following its settings.

Figure 11 shows sample results from the test dataset. There are two modalities involved: image and text modality. We first use images as queries to retrieve similar images and text documents. It can be seen that image results are more relevant to the queries. For instance, the first image result for the first image query is relevant because both images are about architectures. But the text results are not very relevant. This can be attributed to the large semantic gap between different modalities making it difficult to locate semantically relevant objects close in the latent (representation) space. When the images are of the same modality as the queries, their representations can be located close if they share similar semantic features such as color and shape. For queries which are text documents, we find that text results are more relevant than image results. For instance, all five text results are relevant to the first text query, but only three of the image results are relevant. In other words, re-

<sup>6</sup><http://code.google.com/p/cuda-convnet/>








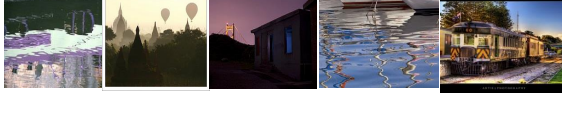
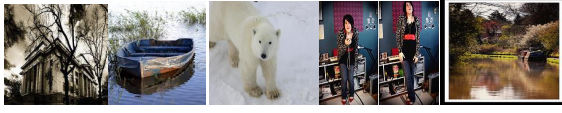


Query	Text Results	Image Results
	nature canada digital quebec nature canada digital wild quebec nature canada scenery waterfalls nature explore food culture walking castle belgium raw	
	lowers garden home lawn cow nederland cows flowers plants flowers park love quality photographer flower flowers orchid	
	water colors love heart joy sepia golden contrast awesome exposure digital great dream photographers blue color needles blue sea portrait mountain cold moon	
water beach sun pink lake boat cloud evening shore	sky blue bravo reflection ocean rainbow sky water ocean boat rocks reflections blue water bravo explore white italy rain boats canal rainy clouds green sea bay cliff needles	
sky architecture quebec montreal	city river lights new lines barge sky night art eos moon stars wales sky night lights bazaar sky sunset building office skyscraper city river old boat house ship thailand	
white house home interior modern french furniture traditional	light portrait window warehouse old man walking mosque red germany decay berlin dress court light dark shoes clothes shirt clothing red car window field head hawk	

Figure 11: Multi-Modal Retrieval. Top 5 similar text documents (one line per document) and images are displayed.

sults from the same modality as the queries are more relevant than those from other modalities.

## 6.5 Training Performance

### 6.5.1 Scalability Test

By distributing the computation onto more nodes, we expect the training time to reach a certain accuracy to be smaller. To measure SINGA scalability, we use the natural image classification as our workload over the parameter server architecture (Figure 6a) with synchronous updates. Specifically, one parameter server node is used with varying number of worker nodes (one worker per group). A training mini-batch consists of 400 images. We ran the training for 10,000 iterations and the results are shown in Figure 12. We can see that SINGA scales almost linearly, i.e. training time to reach the same accuracy level is smaller if we use more worker nodes. This is because the computation load on each worker is smaller with more workers, which reduces the training time per iteration. Furthermore, the number of parameters of the DCNN model (Section 4.2) is modest (85,000 in total), hence the network bandwidth of the server is not saturated, which enables SINGA to scale well.

To study the effect of a large model, we train the big simple MLP from Section 4.1 which has about 10 million parameters. We measure SINGA performance in terms of training throughput defined as the number of images processed per second and compare it against H2O<sup>7</sup> — the current state-of-the-art open source, in-

<sup>7</sup><http://h2o.ai>

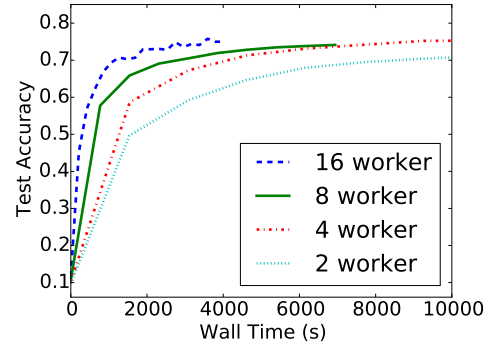


Figure 12: Test Accuracy VS Training Time

memory system for machine learning. H2O's workers run asynchronously, hence we also run SINGA in asynchronous mode. Figure 14a shows the performance when using mini-batch size of 100 (every worker process 100 images per iteration) with varying worker nodes. We can see that SINGA's throughputs are higher H2O's. There are two main reasons. First, H2O is written in Java whereas SINGA is written in C++ which has lower performance overhead. Second, we use batch training while H2O can only conduct on-line training which means parameters are updated after processing every image. Another observation from Figure 14a is that H2O has better scalability than SINGA. To understand this, we measure the performance of H2O's map functions wherein the training is done. We find that these functions are CPU-bounded



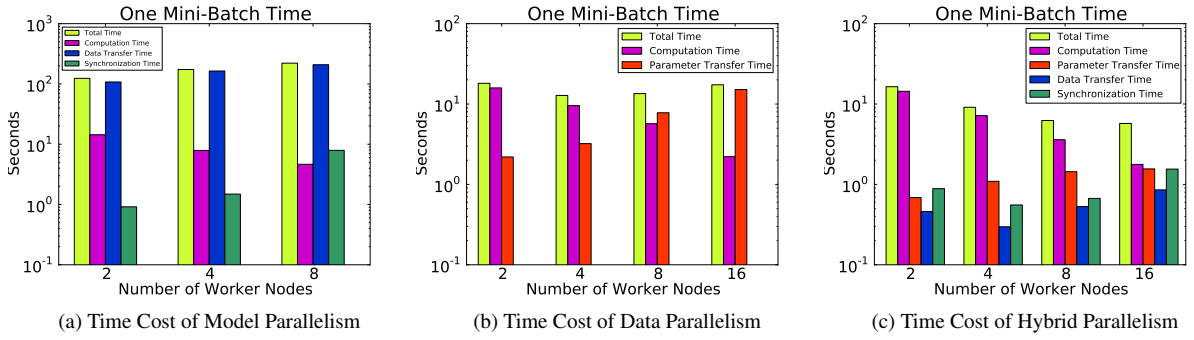


Figure 13: Time Cost Analysis of Different Parallelism Approaches VS. Number of Worker Nodes.

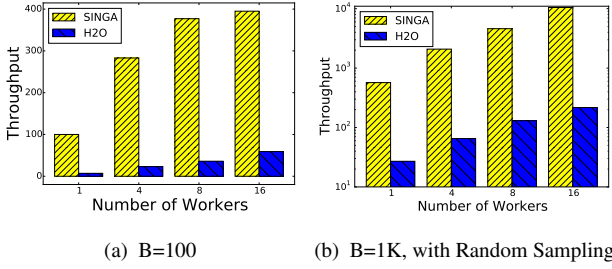


Figure 14: Comparison With H2O on Throughput (Images processed per second). B is mini-batch size.

and can reach only a small throughput of less than 10 images per second. Communication cost which increases with more nodes is small compared to the computation cost. In contrast, SINGA workers process mini-batches very fast, but the server performance is bounded by the network communication. When the cluster size increases, the servers have to send and receive more data. Once the network is saturated, the workers need wait longer for responses from the servers. As a result, the throughput per worker decreases. This is the bottleneck of the Downpour architecture as we analyzed in Section 5.1. To alleviate server bottleneck, we configure each worker to sample only a fraction of parameters when communicating with the servers. The sampling ratio can be controlled to ensure that the network would not be saturated. The results of this random sampling approach is shown in Figure 14b, which demonstrates linear scalability of SINGA.

### 6.5.2 Partitioning Optimization

We train a large DCNN model from [14] to evaluate the performance of different partition approaches. The model has similar structure as the DCNN in Section 4.2, but with much larger size of parameters (about 60 million parameters). We use one worker group and seven server nodes. Figure 13 shows the results when varying the number of workers in the group. The computation time represents the time for running the BP algorithm excluding the network communication time.

For model parallelism (i.e., pure model partitioning), we update the parameters locally, hence there is no parameter transfer between workers and servers. Figure 13a shows that as the number of worker nodes increases, the total time does not decrease but instead increases. Although computation time is reduced when the training is distributed onto multiple worker, the overall time does not drop because it is dominated by the data transfer time from fetching/sending remote feature vectors (by the bridge lay-

ers). With larger group size, the time for synchronizing workers also increases.

Figure 13b shows the time breakdown for data parallelism (i.e., pure data partition). There is no data transfer or synchronization cost, since workers do not communicate with each other. The overall time decreases when up to 4 workers are added, and increases when there are more than 4 workers. This is because the training is computation bounded when the number of workers is small, thus adding more workers reduces the load per worker. On the other hand, more workers also leads to more parameters (and gradients) being transferred. With more than 4 workers, the communication overhead bypasses the gain from reduced computation load.

Figure 13c depicts the time profile of hybrid parallelism. We can see that hybrid parallelism scales better than both model and data parallelisms. The overall time decreases with more workers, and it can be up to  $20\times$  lower than pure model parallelism. There are two reasons. First, computation time decreases because the load is distributed to multiple workers. Second, both the data transfer time and the parameter transfer time increase, but only slowly. In particular, we employ 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 parameter transfer time increases only slowly.

## 7. RELATED WORK

Due to its capabilities in capturing complex regularities of multimedia data (e.g., image and video), deep learning techniques are being adopted by more and more multimedia applications. It has been shown that deep learning models can learn high-level semantic features from raw image data [15]. Based on this, Ji et al. [24] applies deep learning techniques to learn high-level image features to bridge the ‘semantic gap’ that exists between high-level semantic concepts perceived by human and low-level image pixels captured by machines. The learned features are effective for content-based image retrieval. Zhang et al. [30] also exploits deep learning to learn shared high-level representations across textual and visual modalities. The learned representations encodes strong visual and semantic evidence for visual attributes discovery. There are many other applications for deep learning, like the handwritten digits recognition (Section 6.2), natural image classification (Section 6.3) and multi-modal retrieval (Section 4.3).

Different applications use different deep learning models which. It is essential to provide a general deep learning system for non-experts to implement their models without much effort like Caffe [10]. In addition, deep learning training requires huge amount of computing resources to process large scale multimedia data. Hence, a general distributed deep learning training system is desired. Re-

cently, some distributed training approaches have been proposed, for examples [18, 27, 13]. They are specifically optimized for training the AlexNet model [14], thus cannot generalize well to other models. The general distributed deep learning platforms [6, 5, 1] exploit deep learning specific optimizations and hence are able to achieve a high training throughput. However, they forgo usability issues, i.e. the platforms are closed-source and are not easy to use.

## 8. CONCLUSION

In this paper, we proposed a general distributed deep learning system, called SINGA, for supporting multimedia applications. SINGA provides a simple and general programming model to make it easy to use and extensible for a wide range of multimedia applications which use different deep learning models. Flexible training architectures are provided for users to balance the trade-off between the overhead of distributed training and convergence rate. Training optimizations are applied to improve the system's performance. Three representative multimedia applications were implemented to demonstrate that SINGA is a practical distributed training system for deep learning models and multimedia applications.

## 9. 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] T.-S. Chua, J. Tang, R. Hong, H. Li, Z. Luo, and Y.-T. Zheng. Nus-wide: A real-world web image database from national university of singapore. In *CIVR'09*, Santorini, Greece., July 8–10, 2009.
- [3] 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.
- [4] D. C. Ciresan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. *CoRR*, abs/1202.2745, 2012.
- [5] 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.
- [6] 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.
- [7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *(OSDI 2004)*, San Francisco, California, USA, December 6–8, 2004, pages 137–150, 2004.
- [8] F. Feng, X. Wang, and R. Li. Cross-modal retrieval with correspondence autoencoder. In *MM '14*, pages 7–16, 2014.
- [9] G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002.
- [10] 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.
- [11] D. Jiang, G. Chen, B. C. Ooi, K. Tan, and S. Wu. epic: an extensible and scalable system for processing big data. *PVLDB*, 7(7):541–552, 2014.
- [12] A. Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.
- [13] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.
- [14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1106–1114, 2012.
- [15] 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.
- [16] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.
- [17] Y. LeCun, L. Bottou, G. B. Orr, and K. Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade, this book is an outgrowth of a 1996 NIPS workshop*, pages 9–50, 1996.
- [18] 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.
- [19] 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.
- [20] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.
- [21] L. van der Maaten. Accelerating t-sne using tree-based algorithms. *Journal of Machine Learning Research*, 15(1):3221–3245, 2014.
- [22] J. Wan, D. Wang, S. C. H. Hoi, P. Wu, J. Zhu, Y. Zhang, and J. Li. Deep learning for content-based image retrieval: A comprehensive study. In *MM '14*, pages 157–166, 2014.
- [23] 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.
- [24] X. Wang and Y. Wang. Improving content-based and hybrid music recommendation using deep learning. In *MM '14*, pages 627–636, 2014.
- [25] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep image: Scaling up image recognition. *CoRR*, abs/1501.02876, 2015.
- [26] Z. Wu, Y. Jiang, J. Wang, J. Pu, and X. Xue. Exploring inter-feature and inter-class relationships with deep neural networks for video classification. In *MM '14*, pages 167–176, 2014.
- [27] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato. Multi-gpu training of convnets. *CoRR*, abs/1312.5853, 2013.
- [28] 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.
- [29] C. Zhang and C. Re. Dimmwwitted: A study of main-memory statistical analytics. *PVLDB*, 7(12):1283–1294, 2014.
- [30] H. Zhang, Y. Yang, H. Luan, S. Yang, and T. Chua. Start from scratch: Towards automatically identifying, modeling, and naming visual attributes. In *MM '14*, pages 187–196, 2014.
- [31] S. Zhang, A. Choromanska, and Y. LeCun. Deep learning with elastic averaging SGD. *CoRR*, abs/1412.6651, 2014.

## APPENDIX

### A. FEATURE VISUALIZATION

We visualize the image features from the convolutional layers and the inner-product layer of the DCNN model in Section 4.2. We sample 5000 images from the test dataset. Their feature vectors

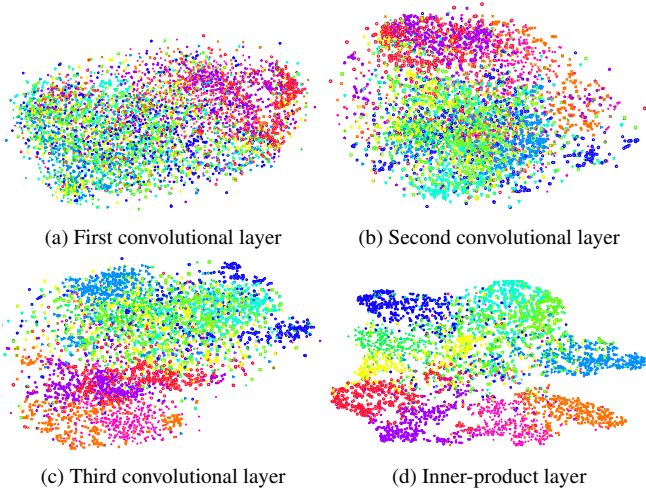


Figure 15: Visualization of Features from DCNN

are projected into a 2-dimensional space via tSNE [21]. Different color represents different categories. We can see that from the low-level layer (Figure 15a) to the high-level layer (Figure 15d), the image features are clustered closer based on their semantic categories. In other words, the learned higher level features capture semantics better than lower level features.

## B. HYBRID PARTITION

In Figure 16, we show the network structure of the DCNN model for Section 4.2 after applying hybrid partition on it over two nodes. For each layer, the numbers indicate the shape of its feature blob. The first number represents the mini-batch size (e.g., 100) of images processed by this layer. The other numbers are for the channel, height and width of the image feature respectively. There are two sets of layers: the circles represent utility layers, e.g., the bridge layers; the rectangles represent neuron layers that apply transformations to the feature vectors. Neuron layers are implemented by users, while utility layers are inserted by SINGA automatically and are transparent to users.

Data partitioning is adopted for the top layers (i.e., C1 to P3), hence the 100 images from the RGBImageLayer are partitioned into two sets by the SliceLayer, each with 50 images. The first 50 images are processed on the local node (i.e., node 0) as the RGBImageLayer, while the other 50 images are processed by another node (i.e., node 1). A pair of bridge layers are added to connect the layers resident on different nodes and to transfer the image features. Model partitioning is adopted by the bottom InnerProductLayer, hence its layer is split into two sub-layers. Each sub-layer processes half of the feature vector for all images from the mini-batch. To concatenate the feature vectors for the whole mini-batch from two nodes, a ReplicateLayer (replicating the features), a pair of bridge layers and a ConcatenateLayer (concatenating features) are added by SINGA. Finally, the two sub-feature vectors from two nodes are concatenated into one feature vector and fed to the SoftmaxLossLayer.

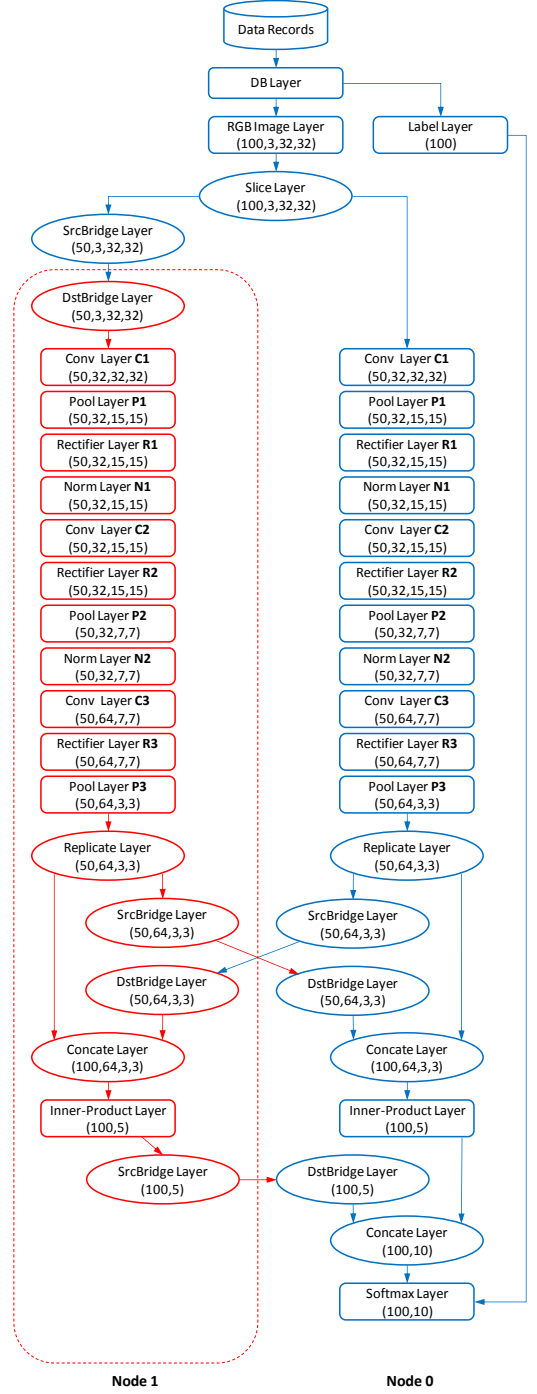


Figure 16: Network Structure for DCNN using Hybrid Partitioning