

# 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

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

## ABSTRACT

Recently, deep learning techniques have enjoyed success in various multimedia applications, such as 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 datasets, 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 which has an intuitive programming model and good scalability. Our experience with developing and training deep learning models for real-life multimedia applications in SINGA shows that the platform is both usable and scalable.

## Categories and Subject Descriptors

I.5.1 [Pattern Recognition]: Models—*Neural Nets*; H.3.4 [Information Storage and Retrieval]: Systems and Software—*Distributed System*

## General Terms

Design, Experimentation, Performance

## Keywords

Deep learning, Multimedia application, Distributed training

## 1. INTRODUCTION

In recent years, we have witnessed successful adoptions of deep learning in various multimedia applications, such as image and video classification [14, 30], content-based image retrieval [25],

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

MM'15, October 26–30, 2015, Brisbane, Australia.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ACM 978-1-4503-3459-4/15/10 ...\$15.00.

DOI: <http://dx.doi.org/10.1145/2733373.2806232>.

music recommendation [28] and multi-modal data analysis [26, 9, 35]. Deep learning refers to a set of feature learning models which consist of multiple layers. Different layers learn different levels of abstractions (or features) 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 fuelled 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 convolution neural network (DCNN) is suitable for image classification [14], recurrent neural network (RNN) for language modelling [18], and deep auto-encoders for multi-modal data analysis [26, 9, 35]. Most of these models are too complex and costly to implement from scratch. An example is the GoogleNet model [24] which comprises 22 layers of 10 different types. Training algorithms are also intricate in details. For instance the Back-Propagation [16] 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, 24]. 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 [31, 21] to train the DCNN [14] with 1.2 million training images and 60 million parameters using one GPU<sup>1</sup>.

Addressing both usability and scalability challenge requires a distributed training platform that supports various deep learning models, that comes with an intuitive programming model (similar to MapReduce [7], Spark [32] and epiC [12] in spirit), and that is scalable. A recent deep learning system, called Caffe [11], addresses the first challenge but falls short at the second challenge

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

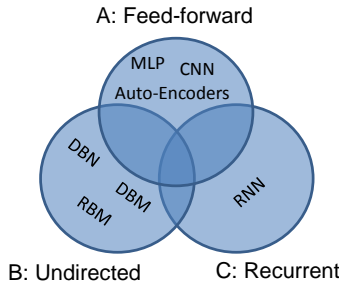


Figure 1: Deep learning model categorization.

(it is not designed for distributed training). There are systems supporting distributed training [21, 31, 13], but they are model specific and do not generalize well to other models. General distributed platforms such as MapReduce [7] and Spark [32] achieve good scalability, but they are designed for generic machine learning. As a result, they lack both the programming model and system optimizations specific to deep learning, hindering the overall usability and scalability. Recently, there are several specialized distributed platforms [6, 4, 1] that exploit deep learning specific optimizations and hence are able to achieve high training throughputs. However, they forgo usability issues: the platforms are closed-source and no detail of their programming models is given, rendering them unusable by multimedia users.

In this paper, we present our effort in bringing deep learning to the masses. In particular, we design and implement an open source distributed deep learning platform, called SINGA which tackles both usability and scalability challenge at the same time. SINGA runs on commodity servers and provides a simple, intuitive programming model which makes it accessible even to non-experts. SINGA's simplicity is driven by the observation that both the structures and training algorithms of deep learning models can be expressed by a simple abstraction: the neuron layer (or layer). In SINGA, the user defines and combines layers to create the neural network model, and the runtime takes care of issues pertaining to the distributed training such as partitioning, synchronization and communication. SINGA achieves scalability via its hybrid architecture design which consists of groups of workers (and servers). Worker groups run asynchronously over different data partitions (*data parallelism*) to improve the convergence rate, and workers within a group run synchronously over one model replica (*model parallelism*) to improve the efficiency of each iteration. SINGA enables users to find an optimal cluster setting (e.g., group size) that trades off the convergence rate and efficiency to minimize the training time to reach certain accuracy.

In summary, this paper makes the following contributions:

1. We present a distributed platform called SINGA which is designed to train deep learning models for multimedia and other applications. SINGA offers a simple and intuitive programming model based on the layer abstraction.
2. We describe SINGA's distributed architecture which runs on commodity servers. The architecture achieves scalability via hybrid parallelism. SINGA runtime handles the communication and synchronization between nodes transparently.
3. We demonstrate SINGA's usability by describing the implementation of three multimedia applications: multi-modal retrieval, dimensionality reduction and language modelling.

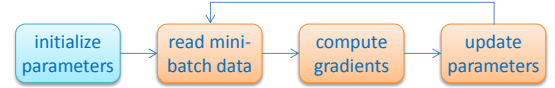


Figure 2: Flow of stochastic gradient descent algorithm.

4. We evaluate SINGA's performance by comparing with other open-source systems. The results show that SINGA is scalable and it outperforms other systems with respect to training time.

The rest of this paper is organized as follows. Section 2 provides the background on training deep learning models. An overview of SINGA as a platform follows in Section 3. The programming model is discussed in Section 4. Section 5 describes the implementations of multimedia applications in SINGA. We discuss SINGA architecture in Section 6, and the model partitioning in Section 7. The experimental study and related works are presented in Section 8 and Section 9 respectively before we conclude in Section 10.

## 2. DEEP LEARNING

Deep learning is considered a feature learning technique consisting of multiple layers in which each layer is associated with a feature transformation function. After going through all layers, raw input features (e.g., pixels of images) are converted into high-level features that are used for tasks such as classification.

We group popular deep learning models into three categories by the connection types between layers, as shown in Figure 1. Category *A* consists of *feed-forward models* wherein the layers are directly connected. The extracted features at higher layers are fed into prediction or classification tasks, e.g., image classification [14]. Examples of models in this category include Multi-Layer Perceptron (MLP), Convolution Neural Network (CNN) and auto-encoders. Category *B* contains models whose layer connections are undirected. These models are often used to pre-train other models [10], e.g., feed-forward models. Deep Belief Network (DBN), Deep Boltzmann Machine (DBM) and Restricted Boltzmann Machine (RBM) are examples of such models. Category *C* comprises models which have recurrent connections. These models are called Recurrent Neural Networks (RNN). They are widely used for modelling sequential data in which prediction of the next position is affected by previous positions. Example applications of RNN include language modelling [18] and image caption generation [17].

A deep learning model has to be trained to find the optimal parameters for the transformation functions. The training quality is measured by a loss function (e.g., cross-entropy loss) for each specific task. Since the loss functions are usually non-linear and non-convex, it is difficult to get closed-form solutions. A common approach is to use the Stochastic Gradient Descent (SGD) algorithm shown in Figure 2. SGD initializes the parameters with random values, then iteratively refines them to reduce the loss based on the computed gradients. There are three typical algorithms for gradient computation corresponding to the three model categories above: Back-Propagation (BP), Contrastive Divergence (CD) and Back-Propagation Through Time (BPTT).

## 3. OVERVIEW

SINGA trains deep learning models using SGD over the worker-server architecture, as shown in Figure 3. Workers compute parameter gradients and servers perform parameter updates. To start a training job, the user (or programmer) submits a job configuration consisting of four components:

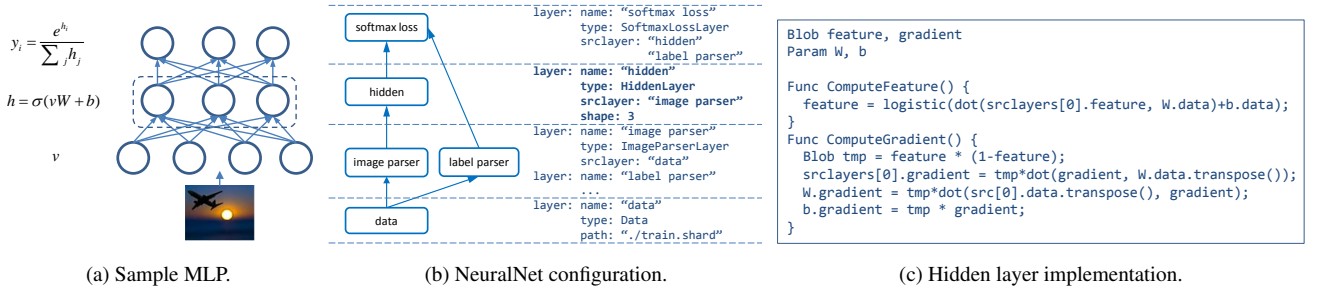


Figure 4: Running example using an MLP.

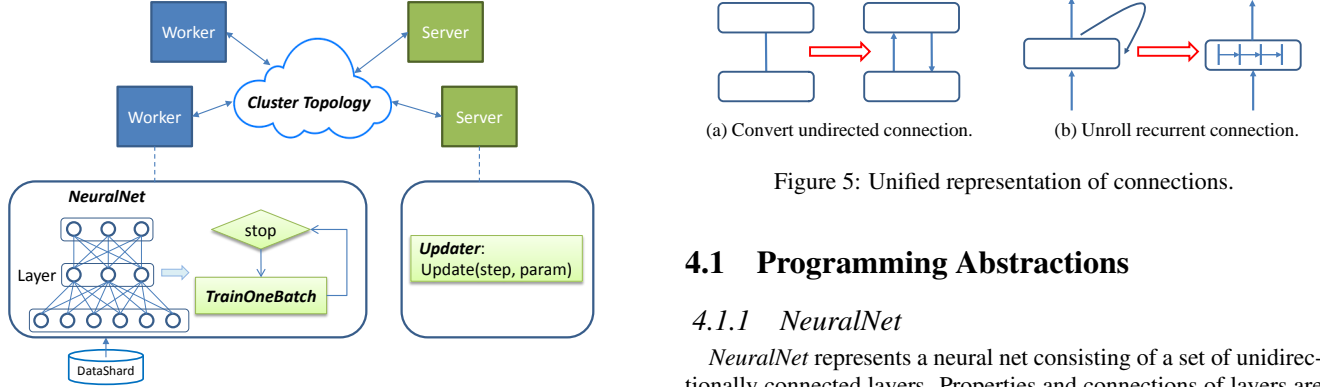


Figure 5: Unified representation of connections.

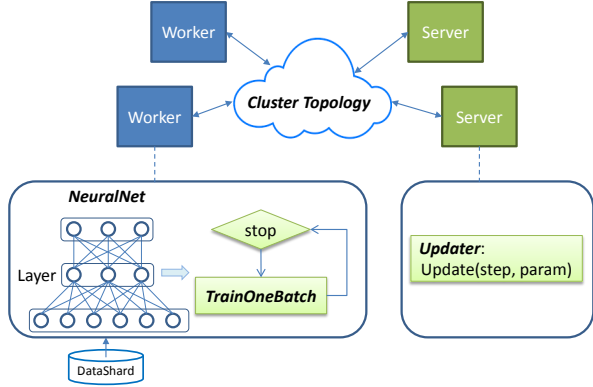


Figure 3: SINGA overview.

- A *NeuralNet* describing the neural network (or neural net) structure with the detailed layers and their connections. SINGA comes with many built-in layers (Section 4.1.2), and users can also implement their own layers.
- A *TrainOneBatch* algorithm for training the model. SINGA implements different algorithms (Section 4.1.3) for all three model categories.
- An *Updater* defining the protocol for updating parameters at the servers (Section 4.1.4).
- A *Cluster Topology* specifying the distributed architecture of workers and servers. SINGA architecture is flexible and can support both synchronous and asynchronous training (Section 6).

Given a job configuration, SINGA distributes the training tasks over the cluster and coordinates the training. In each iteration, every worker calls *TrainOneBatch* function to compute parameter gradients. *TrainOneBatch* takes a *NeuralNet* object representing the neural net, and it visits (part of) the model layers in an order specific to the model category. The resultant gradients are sent to the corresponding servers for updating. The workers then fetch the updated parameters at the next iteration.

## 4. PROGRAMMING MODEL

This section describes SINGA's programming model, particularly the main components of a SINGA job. We use MLP model for image classification (Figure 4a) as a running example. The model consists of an input layer, a hidden feature transformation layer and a Softmax output layer.

## 4.1 Programming Abstractions

### 4.1.1 NeuralNet

*NeuralNet* represents a neural net consisting of a set of unidirectionally connected layers. Properties and connections of layers are specified by users. The *NeuralNet* object is passed as an argument to the *TrainOneBatch* function.

Layer connections in *NeuralNet* are not represented explicitly; instead each layer records its own source layers as specified by users (Figure 4b). Recall that different model categories have different types of layer connections. However, they can be unified using directed edges as follows. For feed-forward models, nothing needs to be done as their connections are already directed. For undirected models, users need to replace each edge with two directed edges, as shown in Figure 5a. For recurrent models, users can unroll a recurrent layer into directed-connecting sub-layers, as shown in Figure 5b.

### 4.1.2 Layer

*Layer* is SINGA's core abstraction. It performs a variety of feature transformations for extracting high-level features. In every SGD iteration, all layers in the *NeuralNet* are visited by the *TrainOneBatch* function during the process of computing parameter gradients.

Figure 6 shows the definition of a basic layer. Each layer consists of two main fields and two functions. The *srcLayer* field represents in-coming edges to the layer. The *feature* field is a set of feature vectors (blob) computed from the source layers. Some layers may require parameters (e.g., a weight matrix) for their feature transformation functions. In this case, these parameters are represented by a *Param* object containing a *data* field for the parameter values and a *gradient* field for the gradients. The *ComputeFeature* function evaluates the feature blob by transforming features from the source layers. The *ComputeGradient* function computes the gradients associated with this layer. These two functions are invoked by the *TrainOneBatch* function during training (Section 4.1.3).

SINGA provides a variety of built-in layers to make it easy for users. Table 1 lists the layer categories in SINGA. For example, the data layer loads a mini-batch of records via the *ComputeFeature* function in each iteration. Users can also define their own layers

```

Layer:
Vector<Layer> srcLayer
Blob feature
Func ComputeFeature()
Func ComputeGradient()

Param:
Blob data, gradient

```

Figure 6: Layer abstraction.

for their specific requirements. Figure 4c shows an example of implementing the hidden layer  $h$  in the MLP. In this example, besides the basic feature blob there is another gradient blob storing the loss gradients with respect to the feature blob. There are two *Param* objects: the weight matrix  $W$  and the bias vector  $b$ . The *ComputeFeature* function rotates (multiply  $W$ ), shifts (plus  $b$ ) the input features and then applies non-linear (logistic) transformations. The *ComputeGradient* function computes the layer’s parameter gradients, as well as the source layer’s gradients that will be used for evaluating the source layer’s parameter gradients.

#### Algorithm 1: BPTrainOneBatch

**Input:** net

```

1 foreach layer in net.layers do
2   Collect(layer.params()) // receive parameters
3   layer.ComputeFeature() // forward prop
4 foreach layer in reverse(net.layers) do
5   layer.ComputeGradient() // backward prop
6   Update(layer.params()) // send gradients

```

#### 4.1.3 TrainOneBatch

The *TrainOneBatch* function determines the sequence of invoking *ComputeFeature* and *ComputeGradient* functions in all layers during each SGD iteration. SINGA implements several *TrainOneBatch* algorithms for the three model categories. For feed-forward and recurrent models, Back-Propagation (BP) algorithm is provided. For undirected modes (e.g. RBM), the Contrastive Divergence algorithm is provided. Users simply select the corresponding algorithm in the job configuration. Should there be specific requirements for the training workflow, users can define their own *TrainOneBatch* function following a template shown in Algorithm 1. Algorithm 1 implements the BP algorithm which takes a *NeuralNet* object as input. The first loop visits each layer and computes their features, and the second loop visits each layer in the reverse order and computes parameter gradients.

#### 4.1.4 Updater

Once the parameter gradients are computed, the workers send these values to servers to update the parameters. SINGA implements several mechanisms for parameter update, such as AdaGrad[8], and AdaDelta [33]. Users can also define their own update protocols by overriding the *Update* function.

### 4.2 MLP Example

To train the MLP model shown in Figure 4a, the first step is to prepare a training dataset. Next, we specify a neural network comprising five layers: data, image parser, label parser, hidden, and

Table 1: Layer categories.

Category	Description
Data layer	Load records from file, database or HDFS.
Parser layer	Parse features from records, e.g., pixels and label.
Neuron layer	Feature transformation, e.g., convolution, pooling.
Loss layer	Compute objective loss, e.g., cross-entropy loss.
Other layer	Utility layers for neural net partitioning.

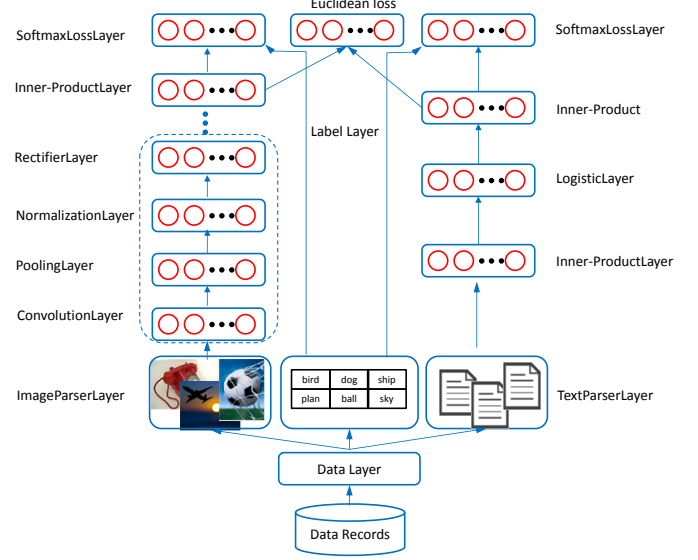


Figure 7: Structure of MDNN.

Softmax loss layer. Because these layers used are built-in, the *NeuralNet* configuration becomes very simple (Figure 4b). Users can customize their own layers in a similar manner shown in Figure 4c. Once the layers are defined, we then specify the *TrainOneBatch* function and parameter update protocol. We select the BP algorithm for the *TrainOneBatch* function, and AdaGrad for parameter update. We now have a complete configuration for training the MLP model. In Section 6, we shall discuss how SINGA carries out distributed training of this model.

## 5. MULTIMEDIA APPLICATIONS

This section demonstrates the use of SINGA for multimedia applications. We discuss the training of three deep learning models for three different applications: a multi-modal deep neural network (MDNN) for multi-modal retrieval, a RBM for dimensionality reduction, and a RNN for language modelling.

### 5.1 MDNN for Multi-modal Retrieval

Feed-forward models such as CNN and MLP are widely used to learn high-level features in multimedia applications, especially for image classification [14]. Here, we demonstrate the training of the MDNN [27], which combines a CNN and an MLP. MDNN is used for extracting features for the multi-modal retrieval task [26, 9, 23] that searches objects from different modalities. In MDNN, a CNN [14] is used to extract image features, and an MLP is used to extract text features. The training objective is to minimize a weighted sum of (1) the error of predicting the labels of image and text documents using extracted features and (2) the distance between features of relevant image and text objects. As a result,

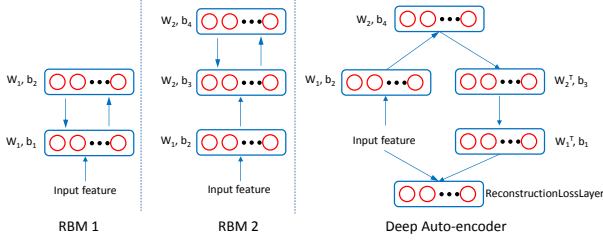


Figure 8: Structure of RBM and deep auto-encoder.

the learned features of semantically relevant objects from different modalities are similar. Finally, multi-modal retrieval is conducted using the learned features.

Figure 7 depicts the user configured neural net of MDNN model in SINGA. We can see that there are two parallel paths: one for text modality and the other for image modality. The data layer reads in records of semantically relevant image-text pairs. The image layer, text layer and label layer then parse the visual feature, text feature (e.g. tags of the image) and labels respectively from the records. The image path consists of layers from DCNN [14], e.g., the convolution layer and pooling layer. The text path includes inner-product (or fully connected) layer, logistic layer and loss layer. The Euclidean loss layer measures the distance of the feature vectors extracted from these two paths. All except the parser layers which are application specific are SINGA's built-in layers. Since this model is a feed-forward model, the BP algorithm should be selected for the *TrainOneBatch* function.

## 5.2 RBM for Dimensionality Reduction

RBM is often employed to pre-train parameters for other models. In this example application, we use RBM to pre-train a deep auto-encoder [10] for dimensionality reduction. Multimedia applications usually operate with high-dimensional feature vectors, which demands large computing resources. Dimensionality reduction techniques, such as Principal Component Analysis (PCA), are commonly applied in the pre-processing step. Deep auto-encoder is reported [10] to have better performance than PCA.

Generally, the deep auto-encoder is trained to reconstruct the input feature using the feature of the top layer. Hinton et al. [10] use RBM to pre-train the parameters for each layer, and fine-tune them to minimize the reconstruction error. Figure 8 shows the model structure (with parser layer and data layer omitted) in SINGA. The parameters trained from the first RBM (RBM 1) in step 1 are ported (through checkpoint) into step 2 wherein the extracted features are used to train the next model (RBM 2). Once pre-training is finished, the deep auto-encoder is unfolded for fine-tuning. SINGA applies the contrastive divergence (CD) algorithm for training RBM and back-propagation (BP) algorithm for fine-tuning the deep auto-encoder.

## 5.3 RNN for Language Modelling

RNN models are often selected for modelling sequential data (e.g. music, videos and text sentences) in multimedia applications. We use SINGA to train the RNN model [18] for language modelling, which can then be exploited to learn word embedding, generate sentences, etc.

The model structure (i.e., *NeuralNet*) of the RNN is configured as shown in Figure 9. This model is trained to predict the next word in a sentence given its previous words. Each sentence is parsed as a sequence of word indices in the parser layer. The embedding layer outputs the feature vector of each word. The hidden feature of each word is computed from the word feature and the hidden feature of

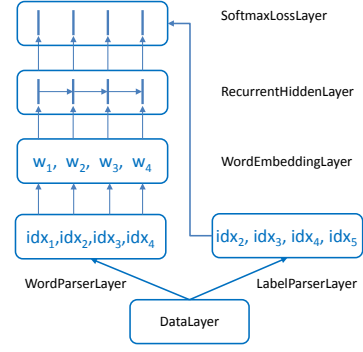


Figure 9: Structure of RNN for language modelling.

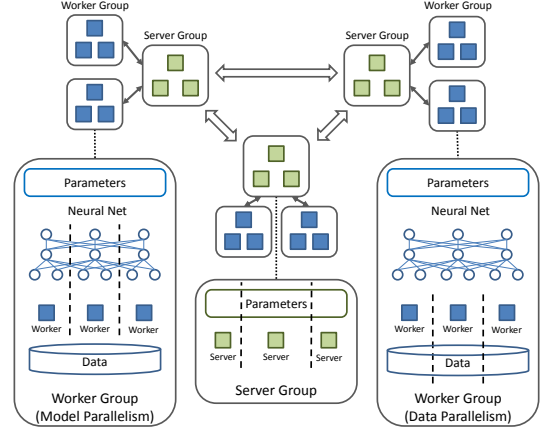


Figure 10: Logical architecture of SINGA.

the previous word. The label layer passes the word indices into the loss layer which calculates the prediction error. For each iteration, the BP algorithm forwards and backwards through all layers. The *ComputeFeature* function of the recurrent layer evaluates all unrolled features in time-order, while the *ComputeGradient* function computes the gradients in a reverse time-order. Consequently, the training is conducted as the back-propagation through time (BPTT) algorithm.

## 6. SYSTEM ARCHITECTURE

In this section, we introduce SINGA's system architecture, and discuss how it supports a variety of state-of-the-art training frameworks.

### 6.1 Overall Architecture

Figure 10 shows the logical architecture. The architecture consists of multiple server groups and worker groups, and each worker group communicates with only one server group. Each server group maintains a complete replica of the model parameters, and is responsible for handling requests (e.g. get or update parameters) from worker groups. Neighboring server groups synchronize their parameters periodically. Typically, a server group contains a number of servers, and each server manages a partition of model parameters. Each worker group trains a complete model replica against a partition of the training dataset (i.e. data parallelism), and is responsible for computing parameter gradients. All worker groups run and communicate with the corresponding server groups asynchronously. However, inside each worker group, the workers syn-



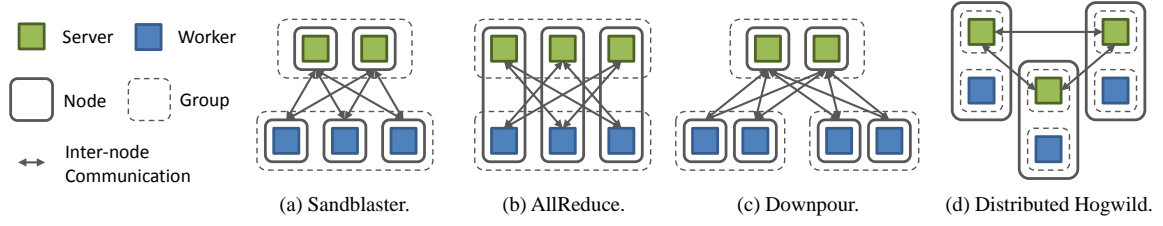


Figure 11: Training frameworks in SINGA.

chronously computes parameter updates for the model replica. There are two strategies to distribute the training workload among workers within a group: by model or by data. More specifically, each worker can compute a subset of parameters against all data partitioned to the group (i.e., model parallelism), or all parameters against a subset of data (i.e., data parallelism). SINGA also supports a hybrid partitioning (Section 7).

Servers and workers are logical training units. In our implementation, each unit executes in a thread. If two units manage the same parameter partition and they are in the same process, it is possible for them to leverage the shared memory to reduce communication cost, as discussed later in Section 6.3.

## 6.2 Distributed Training

State-of-the-art distributed training frameworks include Sandblaster and Downpour from Google’s DistBelief system [6], AllReduce in Baidu’s DeepImage system [29] and the distributed Hogwild implementation in Caffe [11]. SINGA’s architecture is general and flexible enough to support all of these aforementioned frameworks. Specifically, each framework can be realized by specifying the cluster topology: the number of worker/server groups, the number of workers/servers in each group and whether they should reside in a same node.

Existing distributed training frameworks can be classified as synchronous or asynchronous. Both are supported in SINGA: workers of the same group run synchronously, or workers of different groups run asynchronously.

### 6.2.1 Synchronous Training

A synchronous framework is realized by configuring the cluster topology with only one worker group and one server group. The training convergence rate is the same as training on a single node.

Figure 11a shows the Sandblaster framework implemented in SINGA. A single server group is setup to handle all requests from workers of the same group. A worker computes on its own part of model, and only communicates with servers handling related parameters. Figure 11b shows the AllReduce framework in SINGA, in which we bind each worker with a server on the same node, so that each node is responsible for maintaining a partition of parameters and collecting updates from all other nodes.

The synchronous training is usually limited to small or medium size clusters (e.g., less than 100 nodes). This is because when the cluster size is large, the synchronization delay is likely to be larger than the computation time for one iteration. Consequently, the training cannot scale well.

### 6.2.2 Asynchronous Training

An asynchronous framework is implemented by configuring the cluster topology with more than one worker group. The training convergence is likely to be different from single-node training, because multiple worker groups are working on different versions of parameters [34].

Figure 11c shows the Downpour [6] framework implemented in SINGA. Similar to the synchronous Sandblaster, all workers send requests to a global server group. We divide workers into several worker groups, each running independently and working on parameters from its last *get* request. Figure 11d shows the distributed Hogwild framework, in which each node contains a complete server group and a complete worker group. Parameter updates are done locally, so that communication cost during each training step is minimized. However, server groups must periodically synchronize with each other in order to maintain model accuracy and training convergence.

Asynchronous training can improve the convergence rate to some degree. But the improvement usually decreases when there are more and more model replicas. A more scalable training framework should combine both the synchronous and asynchronous training. In SINGA, we can run this hybrid training by launching multiple worker groups that run asynchronously to improve the convergence rate. Within each worker group, we create multiple workers that run synchronously to accelerate one training iteration. Given a fixed budget (e.g., number of nodes in a cluster), we can find one optimal hybrid training framework that trades off the convergence rate and efficiency to achieve the minimal training time.

## 6.3 Parameter Sharing

In SINGA, each server group and worker group have a *ParamShard* object representing a complete model replica. Since the replica may span multiple nodes, the *ParamShard* may need to be partitioned. When possible, however, *ParamShard* partitions can be configured to share the same memory space (when they are in the same process). In this case, the communication between different *ParamShard* reduces to passing pointers to the data, and hence the cost is minimized.

## 7. NEURAL NETWORK PARTITIONING

In this section, we describe how SINGA partitions the neural net to support data parallelism, model parallelism, and hybrid parallelism within one worker group.

SINGA partitions at the granularity of layer. Every layer’s feature blob is considered a matrix whose rows are feature vectors. Thus, the layer can be split on two dimensions. Partitioning on dimension 0 slices the feature matrix by row. For instance, if the mini-batch size is 256 and the layer is partitioned into 2 sub-layers, then each sub-layer would have 128 feature vectors in its feature blob. Partitioning on this dimension has no effect on the parameters, as every *Param* object is replicated in the sub-layers. Partitioning on dimension 1 slices the feature matrix by column. For example, suppose the original feature vector has 50 units, after partitioning into 2 sub-layers, each sub-layer would have 256 feature vectors, each with 25 units. This partitioning may result in *Param* object being split, as shown in Figure 12. Both the bias vector and weight matrix are partitioned into two sub-layers (workers).






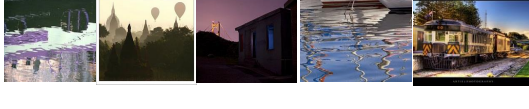
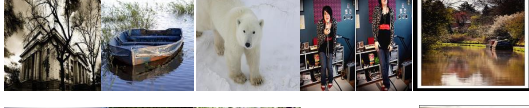

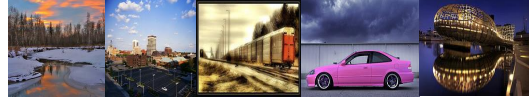
Query	Text Results	Image Results
	nature canada digital quebec nature canada digital wild quebec nature canada scenery waterfalls nature explore food culture walking castle belgium raw	
	lowers garden home lawn cow nederland cows flowers plants flowers park love quality photographer flower flowers orchid	
	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 13: Multi-Modal Retrieval. Top 5 similar text documents (one line per document) and images are displayed.

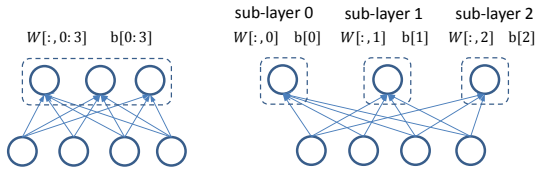


Figure 12: Partition the hidden layer in Figure 4a into three sub-layers.

SINGA partitions the layers when creating the *NeuralNet* instance. Each sub-layer is assigned a location ID based on which it is dispatched to one worker. Some connection layers are automatically inserted to connect the sub-layers. For instance, if two connected sub-layers are located at two different workers, then a pair of bridge layers is inserted to transfer the feature (and gradient) blob between them. When two layers are partitioned on different dimensions, a concatenation layer which concatenates feature rows (or columns) and a slice layer which slices feature rows (or columns) would be inserted. These connection layers help making the network communication and synchronization transparent to the users.

Users can specify the partition decision in two places. The first is in the neural net configuration. A *partition\_dim* field determines the partition dimension for every layer. The second place is in the layer configuration which overwrites the *partition\_dim* in the neural net configuration. Advanced users can also directly specify the location ID for each layer to control the dispatch of layers to workers. For the MDNN model in Figure 7, users can configure the

layers in the image path with location ID 0 and the layers in the text path with location ID 1, making the two paths run in parallel.

When every worker computes the gradients of the entire model parameters, we refer to this process as data parallelism. When different workers compute the gradients of different parameters, we call this process model parallelism. For example, partitioning on dimension 0 results in data parallelism, while partitioning on dimension 1 results in model parallelism. SINGA supports hybrid parallelism wherein some workers compute the gradients of the same subset of model parameters while other workers compute on different model parameters. For example, to implement the hybrid parallelism in [13] for the CNN model, we set *partition\_dim* = 0 for lower layers and *partition\_dim* = 1 for higher layers.

## 8. EXPERIMENTAL STUDY

We evaluated SINGA with real-life multi-media applications. Specifically, we used SINGA to train the models discussed in Section 5, which requires little effort since SINGA comes with many built-in layers and algorithms [20], and is open-sourced<sup>2</sup>. We then examined SINGA's training performance in terms of efficiency and scalability. SINGA is more efficient than three other open-source systems, and it is scalable for both synchronous and asynchronous training.

### 8.1 Apply SINGA for Multimedia Applications

We implemented the multi-modal deep neural network for multi-modal retrieval, and the RBM model for pre-training a deep auto-

<sup>2</sup><http://singa.incubator.apache.org/>,  
<http://www.comp.nus.edu.sg/dbsystem/singa/>

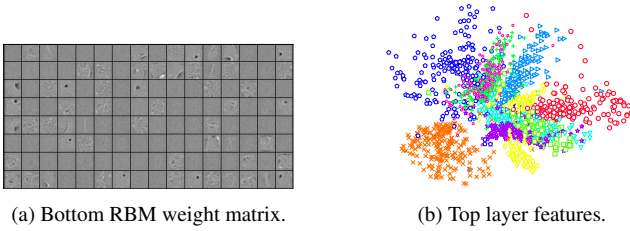


Figure 14: Visualization of the weight matrix in the bottom RBM and top layer features in the deep auto-encoder.

encoder. Due to space constraint, we omit the experiment on RNN for language modelling.

**Multi-modal Retrieval.** We trained the MDNN model for multi-modal retrieval application as described in Section 5.1. We used NUS-WIDE dataset [2], which has roughly 18,000 images after removing images without tags or from non-popular categories. Each image is associated with several tags. We used Word2Vec [19] to learn a word embedding for each tag and aggregated the embeddings of all the tags from the same image as a text feature. Figure 13 shows sample search results. We first used 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 of the first query is relevant because both images are about architecture, but the text results are not very relevant. This can be attributed to the large semantics gap between different modalities, making it difficult to locate semantically relevant objects in the latent (representation) space.

**Dimensionality Reduction.** We trained RBM models to initialize the deep auto-encoder for dimensionality reduction as discussed in Section 5.2. We used the MNIST<sup>3</sup> dataset consisting of 70,000 images of hand-written digits. All images were size-normalized and centered to a fixed size of  $28 \times 28$  pixels. Following the configuration used in [10], we set the size of each layer as 784-1000-500-250-2. Figure 14a visualizes sample columns of the weight matrix of the bottom (first) RBM. We can see that Gabor-like filters are learned. Figure 14b depicts the features extracted from the top layer of the auto-encoder, wherein one point represents one image. Different colors represent different digits. We can see that most images are well clustered according to the ground truth, except for images of digit '4' and '9' which have some overlap (in practice, handwritten '4' and '9' digits are fairly similar in shape).

## 8.2 Training Performance Evaluation

This section presents SINGA's performance in terms of its efficiency and scalability for distributed training. We evaluated both synchronous and asynchronous training frameworks on a single multi-core node and a commodity cluster.

### 8.2.1 Methodologies

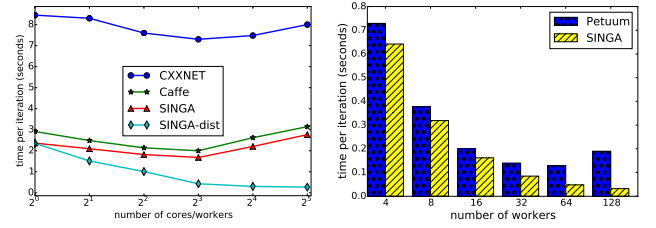
The deep convolution neural network<sup>4</sup> for image classification was used as the training model for benchmarking. The training was conducted over the CIFAR10 dataset<sup>5</sup> which has 50,000 training images and 10,000 test images.

For the single-node setting, we used a 24-core server with 500GB memory. The 24 cores are distributed into 4 NUMA nodes (Intel Xeon 7540). Hyper-threading is turned on. For the multi-node setting, we used a 32-node cluster, each cluster node is equipped with

<sup>3</sup><http://yann.lecun.com/exdb/mnist/>

<sup>4</sup><https://code.google.com/p/cuda-convnet/>

<sup>5</sup><http://www.cs.toronto.edu/~kriz/cifar.html>



(a) On a single node . (b) In a cluster of 32 nodes.

Figure 15: Synchronous training.

a quad-core Intel Xeon 3.1 GHz CPU and 8GB memory. The cluster nodes are connected by a 1Gbps switch.

SINGA uses Mshadow<sup>6</sup> and OpenBlas<sup>7</sup> to accelerate linear algebra operations (e.g. matrix multiplication). Caffe's im2col and pooling code [11] is adopted to accelerate the convolution and pooling operations. We compiled SINGA using GCC with optimization level O2.

### 8.2.2 Synchronous training

We compared SINGA with CXXNET<sup>8</sup> and Caffe [11] — both are open-source systems for training deep learning models on a single GPU node. Both systems were compiled using their default optimization levels: O3 for CXXNET and O2 for Caffe. Since synchronous training has the same convergence rate as sequential SGD, we only compared the training time for one iteration, i.e., one mini-batch.

We first conducted the training on the 24-core single node with 256 images per mini-batch. All three systems use OpenBlas, which exploits multi-threading to accelerate matrix multiplications. We varied the number of threads used by OpenBlas as shown in Figure 15a. Because SINGA is already multi-threaded (each worker and server runs in a single thread), we disabled OpenBlas's multi-threading<sup>9</sup> and launched multiple workers and servers. Specifically, we configured the cluster topology as one server group with four servers and one worker group with varying number of worker threads (Figure 15a). This cluster topology, denoted as SINGA-dist, is in fact the in-memory Sandblaster framework. We can observe that SINGA-dist has the best overall performance: it is the fastest for each number of threads, and it is also the most scalable. All systems using multi-threading OpenBlas have poor scalability. This is because OpenBlas has little awareness of the application details, and hence cannot be fully optimized. For example, it may only parallelize operations such large matrix multiplications. In contrast, SINGA-dist partitions the 256 images in one mini-batch equally. The workers run in parallel for the whole iteration, instead of only for some matrix multiplication operations. Another limitation of OpenBlas, as shown in Figure 15a, is that when the number of threads is larger than 8, the overhead caused by cross-CPU memory access starts to affect the training performance.

Next, we performed experiments on a 32-node cluster comparing with Petuum [5]. Petuum is a distributed machine learning framework. It runs Caffe as an application to train deep learning models. It implements a parameter server to conduct updates from workers (clients), while all workers run synchronously. We used a larger mini-batch size (512) and disabled OpenBlas multi-threading. We

<sup>6</sup><https://github.com/dmlc/mshadow>

<sup>7</sup><http://www.openblas.net/>

<sup>8</sup><https://github.com/dmlc/cxxnet>

<sup>9</sup>by setting OPENBLAS\_NUM\_THREADS=1



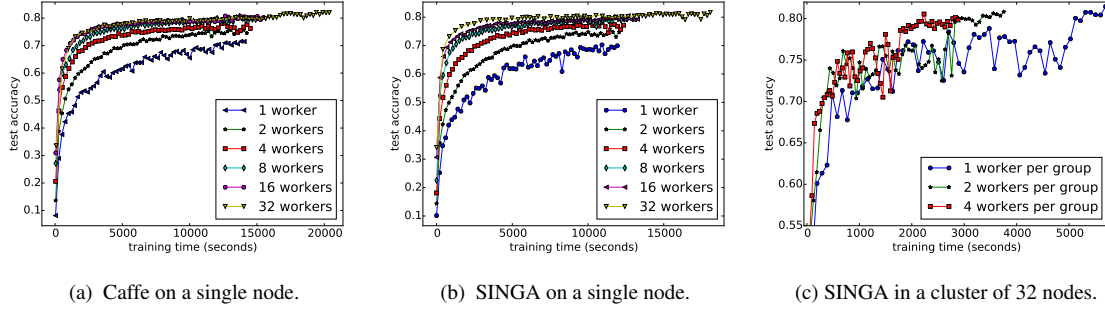


Figure 16: Asynchronous training.

configured the cluster topology to realize the AllReduce framework for SINGA. Specifically, there is one worker group and one server group, and in each node there are 4 workers and 1 server. We varied the size of worker group from 4 to 128, and the server group size from 1 to 32. Figure 15b shows that SINGA achieves almost linear scalability. Petuum scales up to 64 workers, but becomes slower when 128 workers are launched. It might be caused by the communication overhead in the parameter server as well as the synchronization delays among workers. One drawback of the synchronous distributed training is that it cannot scale to too many nodes because the mini-batch size is typically fewer than 1024. Consequently, we cannot launch more than 1024 workers (nodes) lest some workers will not be assigned any image to train. Although we could use more than one worker to train one image, it is not a common practice and could lead to a worse performance.

### 8.2.3 Asynchronous training

Both SINGA and Caffe support in-memory asynchronous training. Particularly, Caffe implements in-memory Hogwild [22] framework, and SINGA implements in-memory Downpour framework. Their main difference is that parameter updates are done by workers in Caffe and by a single server (thread) in SINGA. Figure 16a and Figure 16b show the model accuracy versus training time for Caffe and SINGA with different numbers of worker groups (i.e. model replicas). Every worker processes 16 images per iteration, for the total of 60,000 iterations. It can be seen that SINGA is faster than Caffe. Both systems scale well as the number of workers increases, with respect to (1) the time to reach the same accuracy and (2) the final accuracy. Another observation is that the training takes longer with more workers. This is because of the context-switching overhead when there are more threads (workers). Finally, the performance difference becomes smaller when the cluster size (i.e., the number of model replicas) reaches 16. This implies that there would be little benefit in having too many model replicas. Thus, we fixed the number of model replicas (i.e., worker groups) to 32 in the following experiments for the asynchronous training.

For the multi-node setting, we still used mini-batch of 16 images per worker group and 60,000 training iterations. We selected the distributed Downpour framework: there are 32 worker groups, and one server group with 32 servers (one server thread per node). We varied the number of workers within one group as shown in Figure 16c. We can see that with more workers, the training is faster, since each worker processes fewer images. However, the training is not as stable as in the single-node setting. This may be attributed to the delay of parameter synchronization between workers at the end of each iteration, whereas in single-node training, each worker's update is immediately visible to other workers as they share the memory. The final stage of training (i.e., last few points of each line) is stable because there is only one worker group running dur-

ing that time, namely the testing group. We note that using a warm-up stage, which trains the model using a single worker group at the beginning, may help to stabilize the training as reported in Google's DistBelief system [6].

## 9. 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. [28] apply deep learning techniques to learn high-level image features to bridge the 'semantic gap' that exists between the 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. [35] also exploit deep learning to learn the high-level, shared representations across textual and visual modalities. The learned representations encode strong visual and semantic evidence for visual attributes discovery. Other applications of deep learning include language modelling [18], and multi-modal retrieval [27, 26], etc. (Section 5).

Different applications use different deep learning models. It is essential to provide a general deep learning system for non-experts to implement their models without much effort. In addition, deep learning training requires a huge amount of computing resources to process large scale multimedia data. Hence, a general distributed deep learning training system is essential. Recently, some distributed training approaches have been proposed, for examples [21, 31, 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, 4, 1] exploit deep learning specific optimizations and hence are able to achieve a high training throughput. However, they are closed-source and there are no details of the programming model, rendering them unusable to developers.

## 10. CONCLUSION

In this paper, we proposed a distributed deep learning platform, called SINGA, for supporting multimedia applications. SINGA offers a simple and intuitive programming model, making it accessible to even non-experts. SINGA is extensible and able to support a wide range of multimedia applications requiring different deep learning models. The flexible training architecture gives the user the chance to balance the trade-off between the training efficiency and convergence rate. We demonstrated the use of SINGA for representative multimedia applications, and showed that the platform is both usable and scalable.

## 11. ACKNOWLEDGMENTS

This work was in part supported by the National Research Foundation, Prime Minister's Office, Singapore under its Competitive Research Programme (CRP Award No. NRF-CRP8-2011-08) and A\*STAR project 1321202073. Gang Chen's work was supported by National Natural Science Foundation of China (NSFC) Grant No. 61472348. We would like to thank the SINGA team members and NetEase for their contributions to the implementation of the Apache SINGA system, and the anonymous reviewers for their insightful and constructive comments.

## 12. 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. USENIX Association, Oct. 2014.
- [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*, 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] 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.
- [5] W. Dai, J. Wei, X. Zheng, J. K. Kim, S. Lee, J. Yin, Q. Ho, and E. P. Xing. Petuum: A framework for iterative-convergent distributed ML. *CoRR*, abs/1312.7651, 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] 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.
- [9] F. Feng, X. Wang, and R. Li. Cross-modal retrieval with correspondence autoencoder. In *MM '14*, pages 7–16, 2014.
- [10] G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504 – 507, 2006.
- [11] 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.
- [12] 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.
- [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, G. B. Orr, and K. Müller. Efficient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–50, 1996.
- [17] J. Mao, W. Xu, Y. Yang, J. Wang, and A. L. Yuille. Deep captioning with multimodal recurrent neural networks (m-rnn). *CoRR*, abs/1412.6632, 2014.
- [18] T. Mikolov, S. Kombrink, L. Burget, J. Cernocký, and S. Khudanpur. Extensions of recurrent neural network language model. In *ICASSP*, pages 5528–5531. IEEE, 2011.
- [19] 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.
- [20] B. C. Ooi, K.-L. Tan, S. Wang, W. Wang, Q. Cai, G. Chen, J. Gao, Z. Luo, A. K. H. Tung, Y. Wang, Z. Xie, M. Zhang, and K. Zheng. SINGA: A distributed deep learning platform. In *MM '15*, 2015.
- [21] 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.
- [22] 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.
- [23] H. T. Shen, B. C. Ooi, and K. Tan. Giving meanings to WWW images. In *MM*, pages 39–47. ACM, 2000.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] W. Wang, X. Yang, B. C. Ooi, D. Zhang, and Y. Zhuang. Effective deep learning-based multi-modal retrieval. *The VLDB Journal*, pages 1–23, 2015.
- [28] X. Wang and Y. Wang. Improving content-based and hybrid music recommendation using deep learning. In *MM '14*, pages 627–636, 2014.
- [29] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep image: Scaling up image recognition. *CoRR*, abs/1501.02876, 2015.
- [30] 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.
- [31] O. Yadan, K. Adams, Y. Taigman, and M. Ranzato. Multi-GPU training of convnets. *CoRR*, abs/1312.5853, 2013.
- [32] 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.
- [33] M. D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.
- [34] C. Zhang and C. Re. Dimmwwitted: A study of main-memory statistical analytics. *PVLDB*, 7(12):1283–1294, 2014.
- [35] 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.