A

Deep Learning At Scale and At Ease

Wei Wang, School of Computing, National University of Singapore, Singapore
Gang Chen, College of Computer Science, Zhejiang University, China
Haibo Chen, NetEase, Inc., China
Tien Tuan Anh Dinh and Jinyang Gao and Beng Chin Ooi and
Kian-Lee Tan and Sheng Wang, School of Computing, National University of Singapore, Singapore
Meihui Zhang, Singapore University of Technology and Design

Recently, deep learning techniques have enjoyed success in various multimedia applications, such as image classification and multi-modal data analysis. Large deep learning models are developed for learning rich representations of complex 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 especially when the model is large and complex. 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 based on the common layer abstraction of deep learning models. Good scalability is achieved through flexible distributed training architecture and specific optimization techniques. SINGA runs on GPUs as well as on CPUs, and we show that it outperforms many other state-of-the-art deep learning systems. 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.

CCS Concepts: •**Computing methodologies** → **Neural networks;** •**Information systems** → *Multimedia and multimodal retrieval;* •**Software and its engineering** → *Data flow architectures;*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Multimedia, Deep Learning, Distributed Training

ACM Reference Format:
Wei Wang, Gang Chen, Haibo Chen, Tien Tuan Anh Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Meihui Zhang. 2016. Deep Learning At Scale and At Ease. *ACM Trans. Multimedia Comput. Commun. Appl.* V, N, Article A (January YYYY), 23 pages.
DOI: 0000001.0000001

1. INTRODUCTION

In recent years, we have witnessed successful adoptions of deep learning in various multimedia applications, such as image and video classification [Krizhevsky et al. 2012; Wu et al. 2014], content-based image retrieval [Wan et al. 2014], music recommendation [Wang and Wang 2014] and multi-modal data analysis [Wang et al. 2014; Feng et al. 2014; Zhang et al. 2014]. 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 [Le et al. 2012]. 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 [Krizhevsky et al. 2012]. Three key factors behind deep learning's remarkable achievement are the advances of neural net structures, immense computing power and the availability of

massive training datasets, which together enable us to train large models to capture the regularities of complex 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 with little effort. The user must be able to choose among many existing deep learning models, as different multimedia applications may benefit from different models. For instance, the deep convolutional neural network (DCNN) is suitable for image classification [Krizhevsky et al. 2012], recurrent neural network (RNN) for language modelling [Mikolov et al. 2011], and deep auto-encoders for multi-modal data analysis [Wang et al. 2014; Feng et al. 2014; Zhang et al. 2014]. Furthermore, the user must not be required to implement most of these models and training algorithms from scratch, for they are too complex and costly. An example of complex models is the GoogleLeNet [Szegedy et al. 2014] which comprises 22 layers of 10 different types. Training algorithms are intricate in details. For instance the Back-Propagation [LeCun et al. 1996] 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. As larger training datasets and bigger models are being used to improve accuracy [Ciresan et al. 2010; Le et al. 2012; Szegedy et al. 2014], memory requirement for training the model may easily exceed the capacity of a single CPU or GPU. In addition, the computational cost of training may be too high for a single commodity server, which results in unreasonably long training time. For instance, it takes 10 days [Yadan et al. 2013; Paine et al. 2013] to train the DCNN [Krizhevsky et al. 2012] with 1.2 million training images and 60 million parameters using one GPU [1].

Addressing both usability and scalability challenges requires a distributed training platform that supports various deep learning models, that comes with an intuitive programming model, and that is scalable. Popular deep learning systems, including Caffe [Jia et al. 2014], Torch [Collobert et al. 2011] and Theano [Bastien et al. 2012], address the first challenge but fall short at the second challenge (they are not designed for distributed training). There are several systems supporting distributed training [Paine et al. 2013; Yadan et al. 2013; Krizhevsky 2014], but they are model specific and do not generalize well to other models. General distributed platforms such as MapReduce and epiC [Jiang et al. 2014], achieve good scalability, but they are designed for general data processing. As a result, they lack both the programming model and system optimization specific to deep learning, hindering the overall usability and scalability. Recently, there are several specialized distributed platforms [Dean et al. 2012; Coates et al. 2013; Chilimbi et al. 2014] that exploit deep learning specific optimization and hence are able to achieve high training throughput. However, they forgo usability issues: the platforms are closed-source and no details of their programming models are given, rendering them unusable by multimedia users.

In this paper, we present our effort in bringing deep learning to the masses. In particular, we extend our previous work [Ooi et al. 2015; Wang et al. 2015] on distributed training of deep learning models. In [Wang et al. 2015], we designed and implemented an open source distributed deep learning platform, called SINGA[2], which tackles both usability and scalability challenges at the same time. In this paper, we will introduce optimization techniques and GPU support for SINGA. SINGA 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 using a simple abstraction: the neuron layer (or layer). In SINGA, the user defines and connects layers to form the neural network model, and the runtime transparently manages other issues pertaining to the distributed training such as partitioning, synchronization and communication. Particularly, the neural network is represented as a dataflow computation graph with each layer being a node. During distributed training, the graph is partitioned and each sub-graph can be trained on CPUs or on GPUs. SINGA's scalability comes from its flexible

[1] According to the authors, with 2 GPUs, the training still took about 6 days.
[2] http://www.comp.nus.edu.sg/~dbsystem/singa/

system architecture and specific optimization. Both synchronous and asynchronous training frameworks are supported with a range of built-in partitioning strategies, which enables users to readily explore and find an optimal training configuration. Optimization techniques, including minimizing data transferring and overlapping computation and communication, are implemented to reduce the communication overhead from distributed training.

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 and optimization for reducing the communication overhead in distributed training.
(3) We demonstrate SINGA's usability by describing the implementation of three multimedia applications: multi-modal retrieval, dimensionality reduction and sequence modelling.
(4) We evaluate SINGA's performance by comparing it with other open-source systems. The results show that SINGA is scalable and outperforms other systems in terms of training time.

This paper is an extension of our conference paper [Wang et al. 2015]. In [Wang et al. 2015], we have presented the basic SINGA framework for a homogeneous architecture (where we consider only CPU nodes). In this paper, we extend the framework to a heterogeneous setting that consists of both GPU and CPU processors. Optimization techniques in terms of reducing communication overhead from distributed training are introduced in this paper. Correspondingly, we conducted experiments on GPUs in comparison with existing systems. The rest of this paper is organized as follows. Section 2 provides the background on training deep learning models and related work. An overview of SINGA as a platform follows in Section 3. The programming model is discussed in Section 4. We discuss SINGA architecture and training optimization in Section 5. The experimental study is presented in Section 6 before we conclude in Section 7.

2. BACKGROUND

Deep learning is considered as a feature learning technique. A deep learning model typically consists of multiple layers, each 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 the task of interest, e.g., image classification.

2.1. Models and Training Algorithms

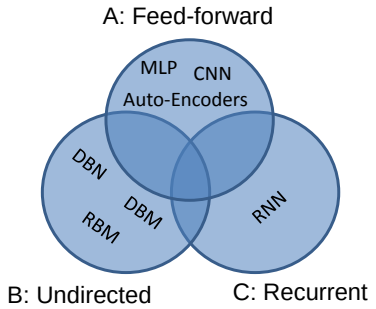


Fig. 1: Deep learning model categorization.

We group popular deep learning models into three categories based on 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 [Krizhevsky et al. 2012]. Example models in this category include Multi-Layer Perceptron (MLP), Convolutional 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 [Hinton and Salakhutdinov 2006], 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 that 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. Language modelling [Mikolov et al. 2011] is a popular application of RNN.

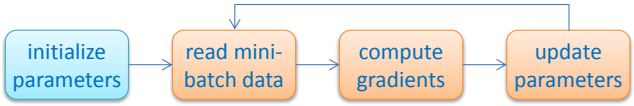


Fig. 2: Flow of stochastic gradient descent algorithm.

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, and 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).

2.2. Related Work

Due to its outstanding 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, e.g., image retrieval [Wan et al. 2014], multi-modal retrieval [Wang et al. 2015; Wang et al. 2014], sentiment analysis [You et al. 2015], etc. In recent years, we have witnessed fast increase of deep learning models' depth, from tens of layers (e.g., AlexNet [Krizhevsky et al. 2012], VGG [Simonyan and Zisserman 2014]) to hundreds of layers [He et al. 2015]. It has been shown that deeper models work better for the ImageNet challenge task [Szegedy et al. 2014; Simonyan and Zisserman 2014]. Meanwhile, training datasets are also becoming larger, from 60,000 images in the MNIST and Cifar datasets to millions of images in the ImageNet dataset. Complex deep models and massive training datasets require a huge amount of computing resources for training.

Table I: Comparison of existing open source systems (and libraries).

	SINGA	Torch	Caffe	MxNet	TF	Theano	CNTK
Programming style	I	I	I	I, D	D	D	D
Flexibility and extensibility	***	****	**	****	****	****	****
Distributed training	****	**	**	***	***	**	***
Hardware support	**	***	***	**	***	***	**

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. Recently, some distributed training approaches have been proposed, for examples [Paine et al. 2013; Yadan et al. 2013; Krizhevsky 2014]. They are specifically optimized for training the AlexNet model [Krizhevsky et al. 2012], thus cannot generalize well to other models. Other general distributed deep learning

platforms [Dean et al. 2012; Coates et al. 2013; Chilimbi et al. 2014] exploit deep learning specific optimization and hence are able to achieve high training throughput. However, they are closed-source and there are no details of the programming model, rendering them unusable to developers. There are also some popular open source systems for training deep learning models on a single node, including TensorFlow [et al. 2015] (TF), Caffe [Jia et al. 2014], Torch [Collobert et al. 2011], MxNet [Chen et al. 2015], Theano [Bastien et al. 2012] and CNTK[et al. 2014]. Distributed training is being added for some of these systems. We briefly compare these systems (or libraries) in Table I, which mainly includes the usability (the first two rows) and efficiency (the last two rows).

Two major programming styles are used in these systems, namely imperative programming and declarative programming. SINGA, Caffe and Torch use imperative programming (denoted as I), which is easy to get started and debug. Tensorflow, Theano and CNTK follow the declarative programming model (denoted as D), where users simply declare the learning objective and the system creates a computation graph (dataflow graph) for automatically optimizing the learning objective. The computation graph provides opportunities for speed and memory optimization [Chen et al. 2016b], but is not easy to debug and requires some effort to get started.

Layer is an inherent abstraction of neural networks. Almost all system provide the layer abstraction (may use different names). Caffe uses Layer as the lowest computation unit. Other system provides Tensor abstractions for algebra operations. Caffe's layer abstraction was designed for feed-forward neural networks, and was latter extended to support RNN, but has no support for energy models. SINGA's layer abstraction supports all three popular neural networks. Other systems, since provides both Tensor and Layer abstractions, are more flexible to implement general machine learning algorithms.

Distributed training is becoming more and more important. All systems in the table support distributed training to improve the training efficiency. Caffe, Torch and Theano were originally designed for easy to prototype new models in research on a single node. Training with multi-GPU cards are supported now. But training in a GPU cluster is not officially supported. SINGA, Tensorflow, CNTK and MxNet are designed with distributed training considered. SINGA is flexible to provide distributed training frameworks (See section 5.2), and exploits asynchronous data transferring and hybrid partitioning to optimize the communication cost (See Section 5.4). MxNet has a fixed two layer distributed training architecture. Tensorflow and CNTK are also working on improving distributed training, e.g., CNTK uses parameter compressing to reduce communication cost [Seide et al. 2014], and Tensorflow uses synchronous SGD with auxiliary workers [Chen et al. 2016a].

Hardware acceleration is vital for the success of deep learning models. All systems in the table use CUDA and the cuDNN library[3] to run computation intensive operations on GPUs, e.g., convolution and pooling. Caffe, Torch and Theano has partial support of OpenCL for deployment on small devices.

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 specifying the following four components:

— 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 for feature transforming or data reading (writing).
— 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).

[3]https://developer.nvidia.com/cudnn

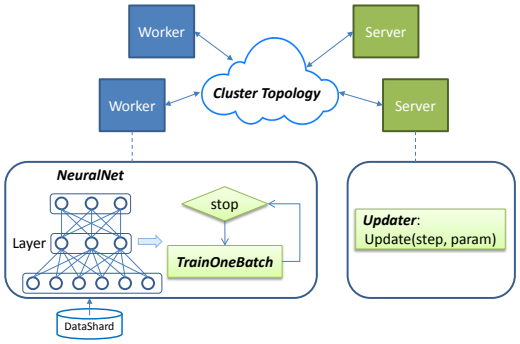


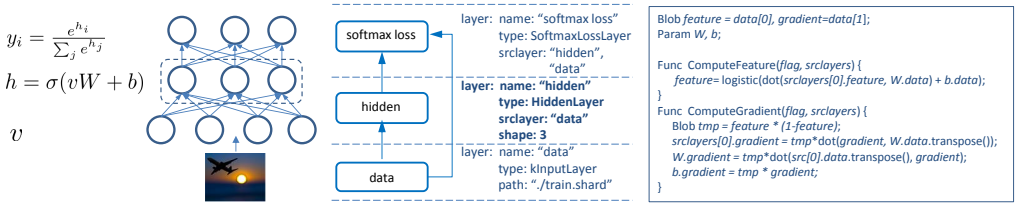
Fig. 3: SINGA overview.

— A *Cluster Topology* specifying the distributed architecture of workers and servers. SINGA's architecture is flexible and can support both synchronous and asynchronous training (Section 5).

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 computed gradients are sent to the corresponding servers for updating. 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 the 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.



(a) Sample MLP. (b) Net configuration. (c) Pseudo code for the hidden layer.

Fig. 4: Running example using an MLP.

4.1. Programming Abstractions

4.1.1. NeuralNet. *NeuralNet* represents a neural net instance in SINGA. It comprises 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 designed explicitly; instead each layer records its own source layers as specified by users (Figure 4b). Although different model categories have different types of layer connections, 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 7. For recurrent models, users can unroll a recurrent layer into directed-connecting sub-layers, as shown in Figure 8.

```
Layer:
  vector<Blob> data
  vector<Param> param
  Func ComputeFeature(flag, srclayers);
  Func ComputeGradient(flag, srclayers);

Param:
  Blob data, gradient;
```

Fig. 5: Layer abstraction.

4.1.2. Layer. *Layer* is a core abstraction in SINGA. Different layer implementations perform different feature transformations to extract 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. From the dataflow perspective, we can regard the neural net as a graph where each layer is a node. The training procedure passes data along the connections of layers and invokes functions of layers. Distributed training can be easily conducted by assigning sub-graphs to workers.

Figure 5 shows the definition of a base layer. The *data* field records data (blob) associated with a layer. Some layers may require parameters (e.g., a weight matrix) for their feature transformation functions. In this case, these parameters are represented by *Param* objects, each with 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 help users build their models. Table II 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 for their specific requirements. Figure 4c shows an example of implementing the hidden layer h in the MLP. In this example, beside feature blobs there are gradient blobs storing the gradients of the loss with respect to the feature blobs. 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.

Table II: Layer categories.

Category	**Description**
Input layers	Load records from file, database or HDFS.
Output layers	Dump records to file, database or HDFS.
Neuron layers	Feature transformation, e.g., convolution.
Loss layers	Compute objective loss, e.g., cross-entropy loss.
Connection layers	Connect layers when neural net is partitioned.

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 two *TrainOneBatch* algorithms for the three model categories. For feed-forward and recurrent models, the BP algorithm is provided. For undirected modes (e.g., RBM), the CD 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.

ALGORITHM 1: BPTrainOneBatch

Input: net
foreach *layer in net.layers* **do**
  Collect(layer.params()) `// receive parameters`
  layer.ComputeFeature() `// forward prop`
end
foreach *layer in reverse(net.layers)* **do**
  layer.ComputeGradient()`// backward prop`
  Update(layer.params())`// send gradients`
end

4.1.4. Updater. Once the parameter gradients are computed, workers send these values to servers to update the parameters. SINGA implements several parameter updating protocols, such as AdaGrad[Duchi et al. 2011]. Users can also define their own updating protocols by overriding the *Update* function.

4.2. 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 sequence modelling.

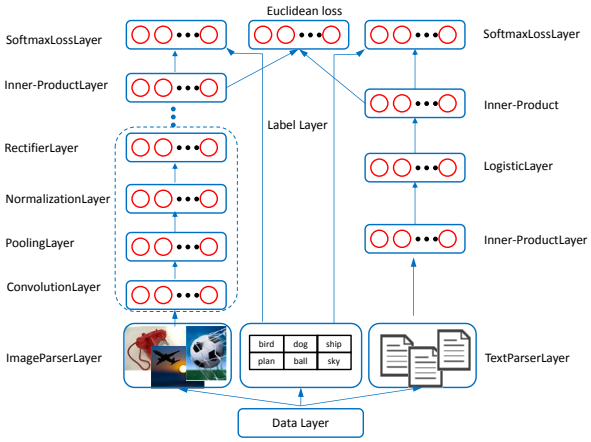


Fig. 6: Structure of MDNN.

4.2.1. 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 [Krizhevsky et al. 2012]. Here, we demonstrate the training of the MDNN [Wang et al. 2015] using SINGA to extract features for the multi-modal retrieval task [Wang et al. 2014; Feng et al. 2014; Shen et al. 2000] that searches objects from different modalities. In MDNN, there is a CNN [Krizhevsky et al. 2012] for extracting image features, and a MLP for extracting 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.

Figure 6 depicts the neural net of the 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 [Krizhevsky et al. 2012], e.g., the convolutional layer and pooling layer. The text path includes an inner-product (or fully connected) layer, a logistic layer and a 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 is selected for the *TrainOneBatch* function.

4.2.2. 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 deep auto-encoders [Hinton and Salakhutdinov 2006] for dimensionality reduction. Multimedia applications typically 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-encoders are reported [Hinton and Salakhutdinov 2006] to have better performance than PCA.

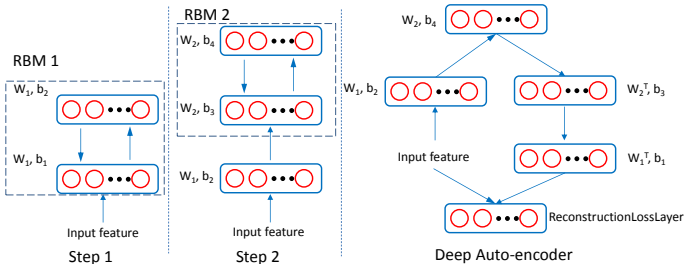


Fig. 7: Structure of RBM and deep auto-encoders.

Generally, the deep auto-encoders are trained to reconstruct the input feature using the feature of the top layer. Hinton et al. [Hinton and Salakhutdinov 2006] used RBM to pre-train the parameters for each layer, and fine-tuned them to minimize the reconstruction error. Figure 7 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-encoders are 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.

4.2.3. RNN for Sequence Modelling. Recurrent neural networks (RNN) are widely used for modelling sequential data, e.g., natural language sentences. We use SINGA to train a Char-RNN model [4] over Linux kernel source code, with each character as an input unit. The model predicts the next character given the current character.

Figure 8 illustrates the net structure of the Char-RNN model. In each iteration, the input layer reads $unroll\_len + 1$ ($unroll\_len$ is specified by users) successive characters, e.g., "int a;" and passes the first $unroll\_len$ characters to OneHotLayers (one per layer), and passes the last $unroll\_len$ characters as labels to the label layer (the label of the i^{th} character is the $(i+1)^{th}$

[4] https://github.com/karpathy/char-rnn

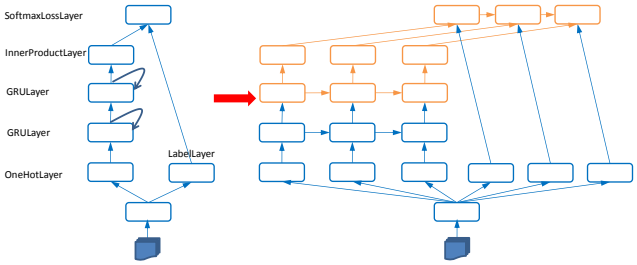


Fig. 8: Structure of 2-stacked Char-RNN (left before unrolling; right after unrolling).

character, i.e., the objective is to predict the next character). The model is configured similarly as for feed-forward models except the training algorithm is BPTT, and unrolling length and connection types are specified for recurrent layers. Different colors are used for illustrating the neural net partitioning which will be discussed in Section 5.3.

5. DISTRIBUTED TRAINING

In this section, we introduce SINGA's architecture, and discuss how it supports a variety of distributed training frameworks.

5.1. System Architecture

Figure 9 shows the logical architecture, which 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 the 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 compute parameter updates synchronously 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 hybrid parallelism (Section 5.3).

In SINGA, servers and workers are execution units running in separate threads. If GPU devices are available, SINGA automatically assigns g GPU devices (g is user specified) to the first g workers on each node. A GPU worker executes the layer functions on GPU if they are implemented using GPU API (e.g., CUDA). Otherwise, the layer functions execute on CPU. SINGA provides several linear algebra functions for users to implement their own layer functions. These linear algebra functions have both GPU and CPU implementation and they determine the running device of the calling thread automatically. In this way, we keep the implementation transparent to users. Workers and servers communicate through message passing. Every process runs the main thread as a stub that aggregates local messages and forwards them to corresponding (remote) receivers. SINGA uses the ZeroMQ library for message passing over the network.

5.2. Training Frameworks

In SINGA, worker groups run asynchronously and workers within one group run synchronously. Users can leverage this general design to run both synchronous and asynchronous training frameworks. Specifically, users control the training framework by configuring the cluster topology, i.e., the number of worker (resp. server) groups and worker (resp. server) group size. In the following, we will discuss how to realize popular distributed training frameworks in SINGA, including Sand-

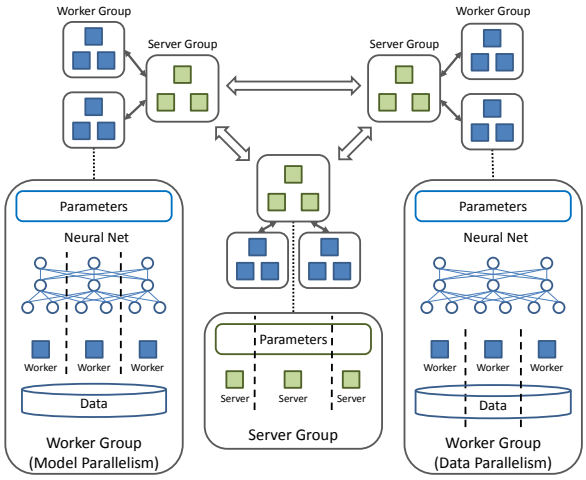


Fig. 9: Logical architecture of SINGA.

blaster and Downpour from Google's DistBelief system [Dean et al. 2012], AllReduce from Baidu's DeepImage system [Wu et al. 2015] and distributed Hogwild from Caffe [Jia et al. 2014].

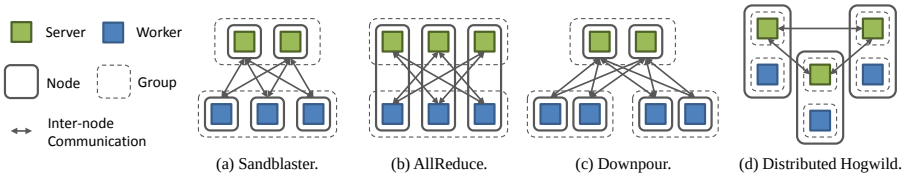


Fig. 10: Training frameworks in SINGA.

5.2.1. 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 that on a single node. Figure 10a shows the Sandblaster framework implemented in SINGA. A single server group is configured to handle requests from workers. A worker operates on its partition of the model, and only communicates with servers handling the related parameters. This framework is typically used if high performance dedicated servers with large network bandwidth are available[Chilimbi et al. 2014]. Figure 10b 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. This framework is suitable for single node multi-GPU case or small GPU clusters. For large clusters, the all-to-all connection would incur huge amount of communication cost.

Synchronous training is typically limited to a small or medium size cluster, e.g. fewer than 100 nodes. When the cluster size is large, the synchronization delay is likely to be larger than the computation time. Consequently, the training cannot scale well.

5.2.2. Asynchronous Training. An asynchronous framework is implemented by configuring the cluster topology with more than one worker groups. The training convergence is likely to be different from single-node training, because multiple worker groups are working on different versions of the parameters [Zhang and Re 2014]. Figure 10c shows the Downpour [Dean et al. 2012] framework implemented in SINGA. Similar to the synchronous Sandblaster, all workers send requests to

a global server group. We divide workers into several groups, each running independently and working on parameters from the last *update* response. Like Sandblaster, this framework also requires the servers to have large bandwidth to handle requests for multiple worker groups. Figure 10d 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, the server group must periodically synchronize with neighboring groups to improve the training convergence. The topology (connections) of server groups can be customized (the default topology is all-to-all connection). This framework is most widely used for single node Multi-GPU environment, where server groups synchronize via shared memory. For a large cluster, the synchronization among server groups would incur significant overhead and delay.

Asynchronous training can improve the convergence rate to some degree. But the improvement typically diminishes when there are more model replicas because the delay (or staleness) of parameter updates would increase. A more scalable training framework should combine both the synchronous and asynchronous training. In SINGA, users can run a hybrid training framework by launching multiple worker groups that run asynchronously to improve the convergence rate. Within each worker group, multiple workers run synchronously to accelerate one training iteration. Given a fixed budget (e.g., number of nodes in a cluster), there are opportunities to find one optimal hybrid training framework that trades off between the convergence rate and efficiency in order to achieve the minimal training time.

5.3. 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.

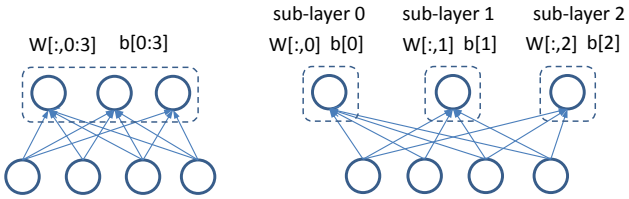


Fig. 11: Partition the hidden layer in Figure 4a.

SINGA partitions a neural net 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 (also called batch dimension) slices the feature matrix by row. For instance, if the mini-batch size is 256 and the layer is partitioned into 2 sub-layers, 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 (also called feature dimension) 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 25 units. This partitioning splits *Param* objects, as shown in Figure 11. Both the bias vector and weight matrix are partitioned into two sub-layers (workers).

Network partitioning is conducted while creating the *NeuralNet* instance. SINGA extends a layer into multiple sub-layers. Each sub-layer is assigned a location ID, based on which it is dispatched to the corresponding worker. Advanced users can also directly specify the location ID for each layer to control the placement of layers onto workers. For the MDNN model in Figure 6, 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. Similarly, for the Char-RNN model shown in Figure 8, we can place the layers of different colors onto different workers. Connection layers will be automatically added 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) are inserted. Connection layers help make the network communication and synchronization transparent to the users.

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. In particular, partitioning on dimension 0 of each layer results in data parallelism, while partitioning on dimension 1 results in model parallelism. Moreover, 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 [Krizhevsky 2014] for the CNN model, we set *partition_dim = 0* for lower layers and *partition_dim = 1* for higher layers. The following list summarizes the partitioning strategies, their trade-off is analyzed in Section 5.4.

(1) Partitioning all layers into different subsets → model parallelism.
(2) Partitioning each single layer into sub-layers on batch dimension → data parallelism.
(3) Partitioning each single layer into sub-layers on feature dimension → model parallelism.
(4) Hybrid partitioning of strategy 1, 2 and 3 → hybrid parallelism.

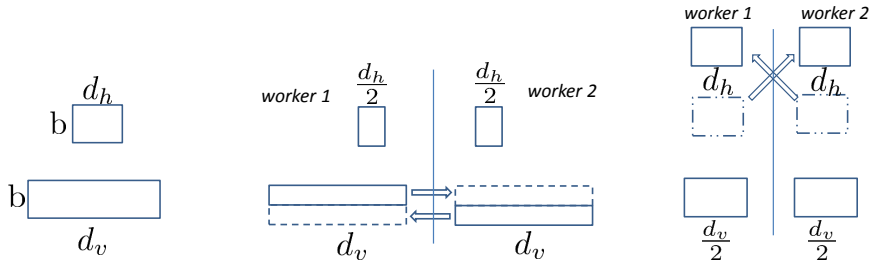
5.4. Optimization

Distributed training (i.e, partitioning the neural net and running workers over different layer partitions) increases the computation power, i.e., FLOPS. However, it introduces overhead in terms of communication and synchronization. Suppose we have a homogeneous computation environment, that is, all workers run at the same speed and get the same workload (e.g., same number of training samples and same size of feature vectors). In this case, we can ignore the synchronization overhead and analyze only the communication cost. The communication cost is mainly attributed to the data transferred through PCIe over multiple GPUs in a single node, or through the network in a cluster. To cut down the overall overhead, first we try to reduce the amount of data to be transferred. Further more, we try to parallelize the computation and communication, in order to hide the communication time. Here we discuss synchronous training only (i.e., a single worker group), which has the identical theoretical convergence as training in a single worker. Optimization techniques that may affect convergence rate of SGD are not considered, e.g., asynchronous SGD (i.e., multiple worker groups) and parameter compression [Seide et al. 2014]. The following analysis works for training either over multiple CPU nodes or over multiple GPU cards on a single node.

5.4.1. Reducing Data Transferring. There are two types of data transferring in distributed training. First, the feature vectors may be transferred as messages if two connected layers are located in different workers, e.g., by model parallelism. Second, the parameter (values and gradients) are transferred for aggregation if there are replicated due to data parallelism. The guideline for reducing data transferring is do data parallelism for layers with fewer parameters and do model parallelism for layers with smaller feature vectors. To illustrate, we use the popular benchmark model, i.e., AlexNet, as an example. AlexNet is a feed-forward model with single path, the i^{th} layer depends on $(i-1)^{th}$ layer directly. It is not feasible to parallelize subsets of layers as in MDNN, therefore we do not consider the first partitioning strategy. Next, we discuss every type of layer involved in AlexNet one by one.

Convolutional layers contain 5% of the total parameters but 90-95% of the computation, according to AlexNet [Krizhevsky 2014]. It is essential to distribute the computation from these layers. Considering that convolutional layers have large feature vectors and a small amount of parameters, it is natural to apply data parallelism.

Fully connected layers occupy 95% of the total parameters and 5-10% of computation [Krizhevsky 2014], therefore we should avoid data parallelism and use model parallelism for them. Particularly, with data parallelism, the communication overhead per worker is $O(p)$, where



(a) Two fully connected layers. (b) Partition on hidden layer. (c) Partition on visible layer.

Fig. 12: Distributed computing for fully connected layers.

p is the size of the (replicated) parameters. Let b be the effective mini-batch size (summed over all workers), K be the number of workers, and d_v (resp. d_h) be the length of the visible (resp. hidden) feature vector. Figure 12b shows the case for data partitioning for the visible layer and model partitioning for the hidden layer, the overhead is $O(b*d_v)$ for exchanging the visible features. Figure 12c applies model partitioning for the visible layer, whose overhead comes from exchanging the hidden features, i.e., $O(b * d_h)$. For the first fully connected layer in AlexNet, p is about 177 million while $d_v = d_h = 4096$. In other words, $p > b * d_v$ and $p > b * d_h$, hence data parallelism is costlier than model parallelism.

For pooling layers and local responsive normalization layers, each neuron depends on many neurons from their source layers. Moreover, they are inter-leaved with convolutional layers, thus it is cheaper to apply data parallelism than model parallelism for them. For the remaining layers, they do not have parameters and their neurons depend on source neurons element-wise, hence their partitioning strategies just need to be consistent with their source layers. Consequently, a simple hybrid partitioning strategy for AlexNet [Krizhevsky 2014] could be applying data parallelism for layers before (or under) the first fully connected layer, and then apply model parallelism or no parallelism for all other layers. Currently, we require users to configure the partitioning strategy for each layer to get the above hybrid partitioning scheme. Automatic optimization and configuration is left as a future work. Reducing data transferring could save power but may not bring speed improvement if the communication cost is hidden due to overlapping with computation as described below.

5.4.2. Overlapping Computation and Communication. Overlapping the computation and communication is another common technique for system optimization. In SINGA, the communication comprises transferring parameter gradients and values, and transferring layer data and gradients. First, for parameter gradients/values, we can send them asynchronously while computing other layers. Take Figure 4 as an example, after the hidden layer finishes *ComputeFeature*, we can send the gradients asynchronously to the server for updates while the worker continues to load data for the next iteration. The updated parameters are transferred back to the server by pushing a copy operation into the Copy queue as shown in Figure 13, which is checked and executed by the worker. Second, the transferring of layer data/gradients typically comes from model partitioning as discussed in Section 5.4.1. In this case, each worker owns a small subset of data and fetches all rest from other workers. To overlap the computation and communication, each worker can just initiate the communication and then compute over its own data asynchronously. Take the Figure 12b as an example, to parallelize the computation and communication, SINGA runs over the layers shown in Figure 13 in order. The *BridgeSrcLayer::ComptueFeature* initiates the sending operations and returns immediately. The *BridgeDestLyer::ComputeFeature* waits until data arrives (by checking a signal for the ending of data transferring). All layers are sorted in topology order followed by communication priority.

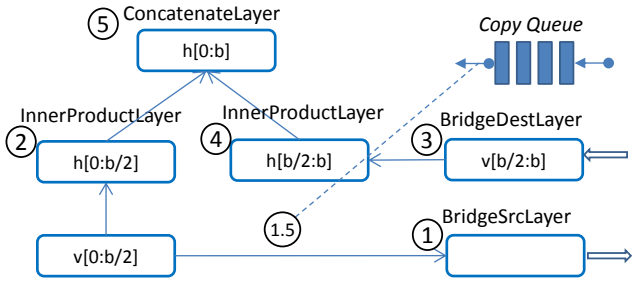


Fig. 13: Parallelize computation and communication for a GPU worker.

6. EXPERIMENTAL STUDY

We have developed SINGA using C++ on Linux platforms. OpenBLAS and cuDNN are integrated for accelerating linear algebra and neural net operations. ZeroMQ is used for message passing. In this section, we evaluate SINGA with real-life multimedia applications. Specifically, we used SINGA to train the models discussed in Section 4.2, which required little development effort since SINGA comes with many built-in layers and algorithms. We then measured SINGA's training performance in terms of efficiency and scalability when running on CPUs and GPUs. We found that SINGA is more efficient than other open-source systems, and it is scalable for both synchronous and asynchronous training.

6.1. Applications of SINGA

We trained models for the example applications in Section 4.2 using SINGA. Users can train these models following the instructions on-line[5].The neural nets are configured using the built-in layers as shown in Figure 6, 7, 8.

Multi-modal Retrieval. We trained the MDNN model for multi-modal retrieval application. We used NUS-WIDE dataset [Chua et al. 2009], which has roughly 180,000 images after removing images without tags or from non-popular categories. Each image is associated with several tags. We used Word2Vec [Mikolov et al. 2013] to learn a word embedding for each tag and aggregated the embedding of all the tags from the same image as a text feature. Figure 14 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 semantic 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. We used the MNIST[6] dataset consisting of 70,000 images of hand-written digits. Following the configuration used in [Hinton and Salakhutdinov 2006], we set the size of each layer as 784→1000→500→250→2. Figure 15(a) visualizes sample columns of the weight matrix of the bottom (first) RBM. We can see that Gabor-like filters are learned. Figure 15(b) 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' (central part) which have some overlap (in practice, handwritten '4' and '9' digits are fairly similar in shape).

Char-RNN We used the Linux kernel source code extracted using an online script[7] for this application. The dataset is about 6 MB. The RNN model is configured similar to Figure 8. Since

[5]http://singa.apache.org/v0.3.0/en/docs/examples.html
[6]http://yann.lecun.com/exdb/mnist/
[7]http://cs.stanford.edu/people/karpathy/char-rnn

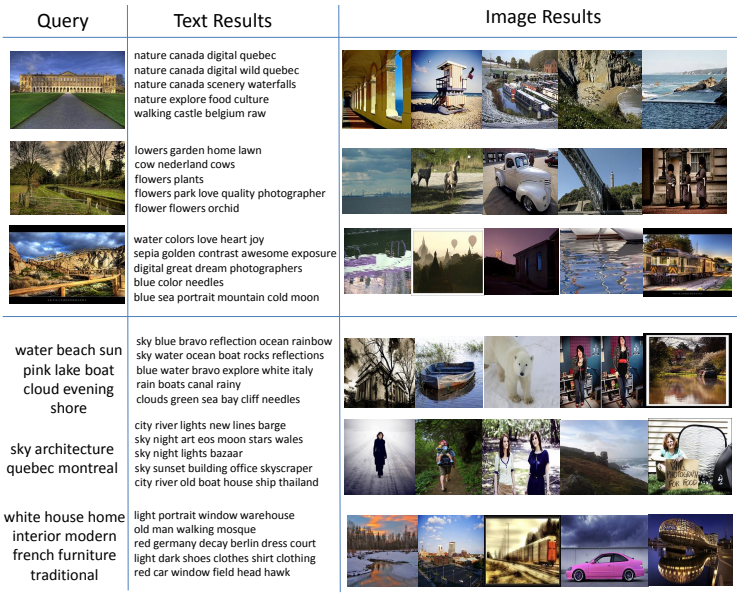


Fig. 14: Multi-Modal Retrieval. Top 5 similar text documents (one line per document) and images are displayed.

(a) Bottom RBM weight matrix.

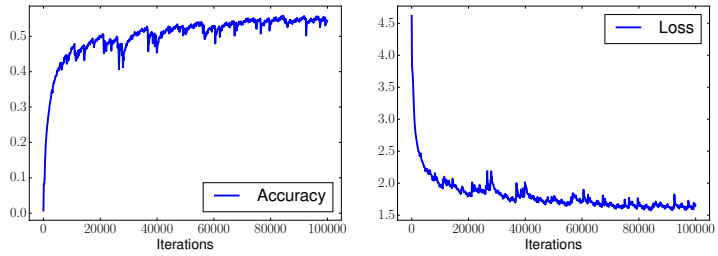
(b) Top layer features.

Fig. 15: Visualization of the weight matrix in the bottom RBM and top layer features in the deep auto-encoder.

this dataset is small, we used one stack of recurrent layers (Figure 8 has two stacks). The training loss and accuracy is shown in Figure 16. We can see that the Char-RNN model can be trained to predict the next character given previous characters in the source code more and more accurately. There some fluctuations due to the variance of data samples in different mini-batches (the loss and accuracy are computed per mini-batch).

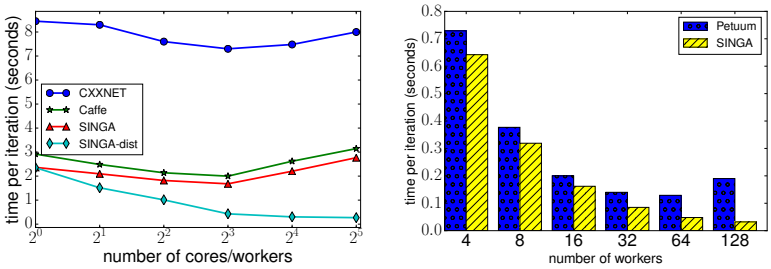
6.2. Training Performance Evaluation on CPU

We evaluated SINGA's training efficiency and scalability for both synchronous and asynchronous frameworks on a single multi-core node, and on a cluster of commodity servers.



(a) Training accuracy. (b) Training loss.

Fig. 16: Training accuracy and loss of Char-RNN.



(a) On the single node. (b) On the 32-node cluster.

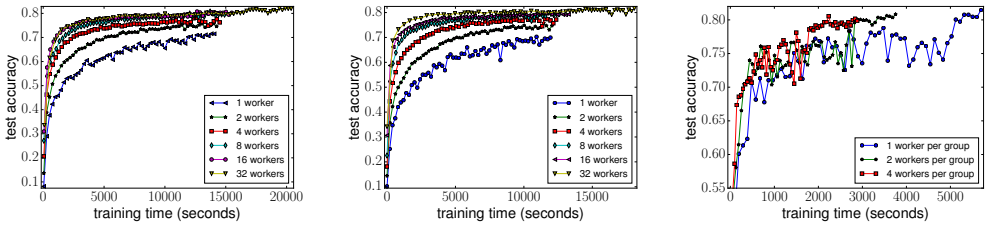
Fig. 17: Synchronous training.

6.2.1. Methodologies. The deep convolution neural network[8] for image classification was used as the training model for benchmarking. The training was conducted over the CIFAR10 dataset[9] 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 a quad-core Intel Xeon 3.1 GHz CPU and 8GB memory. The cluster nodes are connected by a 1Gbps switch.

6.2.2. Synchronous training. We compared SINGA with CXXNET[10] and Caffe [Jia et al. 2014]. All three systems use OpenBlas to accelerate matrix multiplications. Both CXXNET and Caffe were compiled with their default optimization levels: O3 for the former and O2 for the latter. We observed that because synchronous training has the same convergence rate as that of sequential SGD, all systems would converge after same number of iterations (i.e., mini-batches). This means the difference in total training time among these systems is attributed to the efficiency of a single iteration. Therefore, we only compared the training time for one iteration. We ran 100 iterations for each system and averaged the result time over 50 iterations: 30^{th} to 80^{th} iteration, in order to avoid the effect of starting and ending phases.

On the 24-core single node, we used 256 images per mini-batch and varied the number of OpenBlas's threads. The result is shown in Figure 17(a). *SINGA-dist* represents the SINGA configuration in which there are multiple workers, each worker has 1 OpenBlas thread[11]. In contrast, *SINGA*

[8]https://code.google.com/p/cuda-convnet/
[9]http://www.cs.toronto.edu/ kriz/cifar.html
[10]https://github.com/dmlc/cxxnet
[11]OPENBLAS_NUM_THREADS=1



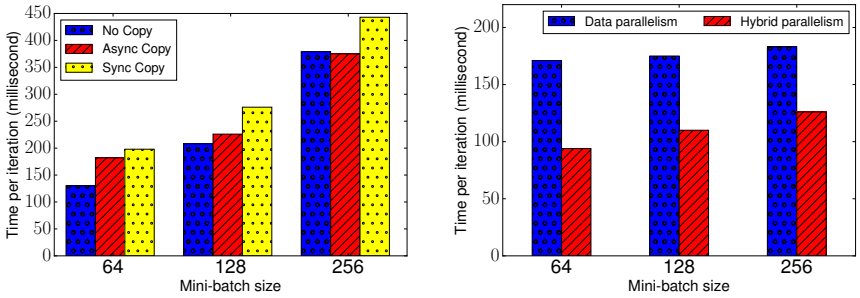
(a) Caffe on the single node. (b) SINGA on the single node. (c)SINGA on the cluster.

Fig. 18: Asynchronous training.

represents the configuration which has only 1 worker. We configured SINGA-dist with the cluster topology consisting of one server group with four servers and one worker group with varying number of worker threads (Figure 17(a)). In other words, SINGA-dist ran as the in-memory Sandblaster framework. We can see that SINGA-dist has the best overall performance: it is the fastest for each number of threads, and it is also the most scalable. Other systems using multi-threaded OpenBlas scale poorly. This is because OpenBlas has little awareness of the application, and hence it cannot be fully optimized. For example, it may only parallelize specific operations such as large matrix multiplications. In contrast, in SINGA-dist partitions the mini-batch equally between workers and achieves parallelism at the worker level. Another limitation of OpenBlas, as shown in Figure 17(a), is that when there were more than 8 threads, the overheads caused by cross-CPU memory access [Tan et al. 2015] started to have negative effect on the overall performance.

On the 32-node cluster, we compared SINGA against another distributed machine learning framework called Petuum [Dai et al. 2013]. Petuum runs Caffe as an application to train deep learning models. It implements a parameter server to perform updates from workers (clients), while the workers run synchronously. We used a larger mini-batch size (512 images) and disabled OpenBlas multi-threading. We configured SINGA's cluster topology to realize the AllReduce framework: there is 1 worker group and 1 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. We note that one drawback of synchronous distributed training is that it cannot scale to too many nodes because there is typically an upper limit on the mini-batch size (1024 images, for instance). Consequently, there is an upper bound on the number of workers we can launch (1024 workers, for instance), otherwise some workers will not be assigned any image to train. Figure 17(b) shows that SINGA achieves almost linear scalability. In contrast, Petuum scales up to 64 workers, but becomes slower when 128 workers are launched. It might be attributed to the communication overheads at the parameter server and the synchronization delays among workers.

6.2.3. Asynchronous training. We compared SINGA against Caffe which has support for in-memory asynchronous training. On the single node, we configured Caffe to use the in-memory Hogwild [Recht et al. 2011] framework, and SINGA to use the 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 18(a) and Figure 18(b) show the model accuracy versus training time with varying numbers of worker groups (i.e. model replicas). Every worker processed 16 images per iteration, for a total of 60,000 iterations. We can see that SINGA trains faster than Caffe. Both systems scale well as the number of workers increases, both in terms of the time to reach the same accuracy and of the final converged accuracy. We can also observe that the training takes longer with more workers. This is due to the increased overhead in context-switching when there are more threads (workers). Finally, we note from the results that the performance difference becomes smaller when the cluster size (i.e., the number of model replicas) reaches 16. This implies that there would



(a) Hide communication cost. (b) Reduce data transferring.

Fig. 19: Effect of optimization techniques.

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 distributed asynchronous training.

On the 32-node cluster, we used mini-batch of 16 images per worker group and 60,000 training iterations. We varied the number of workers within one group, and configured the distributed Downpour framework to have 32 worker groups and 32 servers per server group (one server thread per node). We can see from Figure 18(c) that with more workers, the training is faster because each worker processes fewer images. However, the training is not as stable as in the single-node setting. This may be caused by the delay (staleness) of parameter synchronization between workers, which is not present in single-node training because parameter updates are immediately visible on the shared memory. The final stage of training (i.e., last few points of each line) is stable because there is only one worker group running during 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 [Dean et al. 2012].

6.3. Training Performance Evaluation on GPU

We evaluated the training performance of SINGA running on GPUs. We first analyzed the two optimization techniques discussed in Section 5.4, then we compared SINGA with other open source, state-of-the-art systems.

6.3.1. Methodologies. We used the on-line benchmark model from Soumith[12] as the training workload. The model is adapted from the AlexNet [Krizhevsky 2014] model with some layers omitted. Two sets of hardware are used in our experiments, whose specs and software configurations are shown in Table III.

Table III: Specs of hardware and software.

Type	CPU	Memory	GPU	CUDA	cuDNN
Single node	Intel i7-5820K	16 GB	GTX 970 (4GB)	7.0	4.0
GPU cluster (4 nodes)	Intel i7-5820K	64 GB	GTX TITAN-X (12GB)	7.5	4.0

6.3.2. Overlapping Communication and Computation. In Section 5.4.2, we analyzed the optimization technique for hiding the communication overhead by overlapping it with the computation. Here we evaluate the effect of this technique using the single node. Particularly, we compare the efficiency in terms of time per iteration for three versions of SINGA. No Copy version indicates that there is

[12] https://github.com/soumith/convnet-benchmarks

no communication between GPU and CPU, which is widely used for training with a single GPU, where all operations including parameter update are conducted on the single GPU. The other two versions conduct BP algorithm on GPU and parameter updating on CPU, differing only by whether data transferring is done synchronously or asynchronously.

Figure 19(a) shows the time per iteration with different mini-batch size. First, we can see that No Copy is the fastest one because it has no communication cost at all. Second, Async Copy is faster than Sync Copy, which suggests that the asynchronous data transferring benefits from the overlapping communication and computation. Moreover, we can see that when the mini-batch increases, the difference between Async Copy and Sync Copy decreases. This is because for large mini-batches, the BP algorithm spends more time doing computation, which increases the overlap area of computation and communication, effectively reducing the overhead. For mini-batch size = 256, Async Copy is even faster than No Copy, this is because Async Copy does not do parameter update, which is done by the server in parallel with BP. However, No Copy has to do BP and parameter updating in sequential.

6.3.3. Reducing Data Transferring. In Section 5.4.1, we discussed how hybrid partitioning is better than other strategies in terms of the overheads in transferring feature vectors between layers in different workers. To demonstrate its effectiveness, we ran SINGA on the single node using two partitioning strategies, i.e., data partitioning and hybrid partitioning for the first fully connected layer in AlexNet. 2 workers are launched (one per GPU). Figure 19(b) shows the time per iteration with different mini-batch sizes. We can see that hybrid partitioning has better performance over data partitioning. For data partitioning, only parameter gradients and values are transferred, which is independent of the mini-batch size, thus the time per iteration does not change much when mini-batchs size increases. For hybrid partitioning, when the mini-batch size increases, more feature vectors would be transferred, and then the time increases.

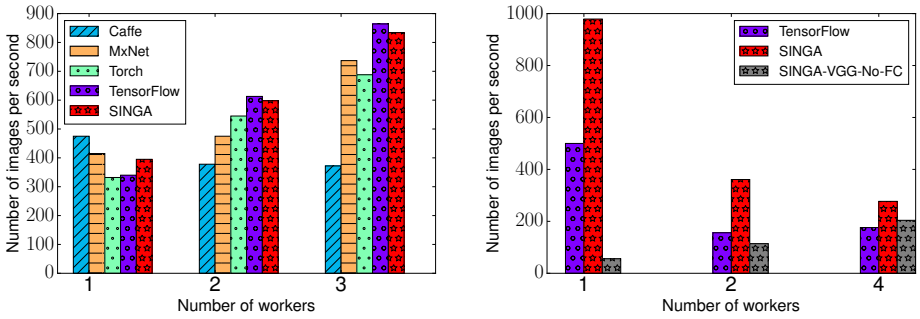
6.3.4. Comparison with Other Systems. We also compared SINGA with four other state-of-the-art deep learning systems namely, Caffe [Jia et al. 2014], MxNet [Chen et al. 2015], Torch7 [Collobert et al. 2011], and TensorFlow [et al. 2015] (TF). The performance is measured using the throughput, i.e., number of processed images per second. We used (or adapted) the scripts (or instructions) from each system's multi-GPU examples [13] [14] [15] [16]. We simply adapted the on-line code for the experiments, better performance could be achieved with further tuning.

We first compared the throughput of training on a single node with different number of workers (GPUs). We varied the number of workers from 1 to 3, where each worker ran on a GPU. Because Tensorflow ran out of memory with batch size = 128 (the setting used by Soumith's benchmark), we decreased the batch size to 96 for all runnings. The results are shown in Figure 20(a).

Caffe has the best single worker performance. This is because there is no parameter transferring between GPU and CPU, whereas others have parameter transferring, e.g., SINGA has to transfer the parameter from the worker (on GPU) to the server (on CPU). However, its performance decreases when more workers are added. This is because 1) the parameters have to be transferred among GPUs (via CPUs), which brings in communication cost; 2)its tree reduction communication pattern (See the link in the footnote) incurs more cost than the all-to-one or all-reduce communication pattern used by other systems when there are more than 2 workers.

For other systems, they have similar performance for the single worker case, as they all use the same cuDNN library for most of the computation. Note that their throughput is lower than Soumith's benchmark because the GPUs are slower than Soumith's and there is communication cost whereas Soumith's benchmark does not involve parameter transferring. Thanks to the optimization

[13] Caffe, https://github.com/BVLC/caffe/blob/master/docs/multigpu.md
[14] MxNet, https://mxnet.readthedocs.io/en/latest/how_to/multi_devices.html
[15] Torch7, https://github.com/soumith/imagenet-multiGPU.torch
[16] Tensorflow, https://www.tensorflow.org/versions/r0.8/tutorials/deep_cnn/index.html and https://github.com/tensorflow/models/tree/master/inception



(a) A single node with multi-GPUs. (b) A GPU cluster with 4 nodes.

Fig. 20: Performance comparison of open source systems.

techniques introduced in Section 5.4, SINGA has almost linear scalability. Tensorflow shows the best scalability among all tested system.

Next, we ran SINGA and Tensorflow in the GPU cluster using the synchronous training framework. We varied the number of nodes (one GPU worker per node) as shown in Figure 20(b). For Tensorflow, we launched one parameter server, which communicates with all workers via gRPC. For SINGA, we created a single server group with 1 server, which is in the same process as the first worker and communicates with other workers using ZeroMQ. We reduced the size of the first fully connected layer to 128, because this layer has a big parameter matrix whose size exceeds the limit of the Protobuf message used by Tensorflow. Consequently, the performance for single node (i.e., single GPU) is better than that in Figure 20(a). We can see that SINGA performs much better than Tensorflow in terms of throughput. It is likely caused by the network communication, which is not well optimized in Tensorflow. SINGA avoids some communication cost by running the first worker and the parameter server in the same process (they transfer messages via sharing memory). Both systems show poor scalability when there are more than two workers (nodes). On the one hand, node-to-node communication cost and synchronization cost are introduced when there are more than two workers. One the other hand, the Alexnet model has too many parameters that makes the communication the bottleneck. To verify this explanation, we conducted another set of experiments using the VGG model [Simonyan and Zisserman 2014] with fully connected layers omitted, denoted as VGG-No-FC. This model has 10 convolutional layers and a small amount of parameters, which make it more computation intensive than the Alexnet model. The result in Figure 20(b) shows that SINGA has good scalability for this model, which confirms our explanation. To conclude, distributed training is more suitable for models that are computation intensive and with a small amount of parameters.

7. 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. Optimization techniques are applied to improve the training performance. We demonstrated the use of SINGA for representative multimedia applications using a CPU cluster and a single node with multiple GPU cards, and showed that the platform is both usable and scalable.

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 Apache SINGA, the anonymous reviewers for their insightful and constructive comments, and NUS SeSaMe center for sharing the GPU cluster for our experiments. Meihui's work was funded under the Energy Innovation Research Programme (EIRP, Award No. NRF2014EWTEIRP002-026), administered by the Energy Market Authority (EMA). The EIRP is a competitive grant call initiative driven by the Energy Innovation Programme Office, and funded by the National Research Foundation (NRF).

REFERENCES

Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, and Yoshua Bengio. 2012. Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop. (2012).

Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Józefowicz. 2016a. Revisiting Distributed Synchronous SGD. *CoRR* abs/1604.00981 (2016). http://arxiv.org/abs/1604.00981

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv preprint arXiv:1512.01274* (2015).

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016b. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). http://arxiv.org/abs/1604.06174

Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI*. USENIX Association, 571–582. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi

Tat-Seng Chua, Jinhui Tang, Richang Hong, Haojie Li, Zhiping Luo, and Yan-Tao. Zheng. July 8-10, 2009. NUS-WIDE: A Real-World Web Image Database from National University of Singapore. In *CIVR'09*.

Dan Claudiu Ciresan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. 2010. Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition. *CoRR* abs/1003.0358 (2010).

Adam Coates, Brody Huval, Tao Wang, David J. Wu, Bryan C. Catanzaro, and Andrew Y. Ng. 2013. Deep learning with COTS HPC systems. In *ICML (3)*. 1337–1345.

R. Collobert, K. Kavukcuoglu, and C. Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.

Wei Dai, Jinliang Wei, Xun Zheng, Jin Kyu Kim, Seunghak Lee, Junming Yin, Qirong Ho, and Eric P. Xing. 2013. Petuum: A Framework for Iterative-Convergent Distributed ML. *CoRR* abs/1312.7651 (2013). http://arxiv.org/abs/1312.7651

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew W. Senior, Paul A. Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large Scale Distributed Deep Networks. In *NIPS*. 1232–1240.

John C. Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research* 12 (2011), 2121–2159. http://dl.acm.org/citation.cfm?id=2021068

Amit Agarwal et al. 2014. *An Introduction to Computational Networks and the Computational Network Toolkit*. Technical Report. Microsoft Technical Report MSR-TR-2014-112.

Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.

Fangxiang Feng, Xiaojie Wang, and Ruifan Li. 2014. Cross-modal Retrieval with Correspondence Autoencoder. In *ACM Multimedia*. 7–16. DOI: http://dx.doi.org/10.1145/2647868.2654902

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385* (2015).

Geoffrey Hinton and Ruslan Salakhutdinov. 2006. Reducing the Dimensionality of Data with Neural Networks. *Science* 313, 5786 (2006), 504 – 507.

Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

Dawei Jiang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Sai Wu. 2014. epiC: an Extensible and Scalable System for Processing Big Data. *PVLDB* 7, 7 (2014), 541–552. http://www.vldb.org/pvldb/vol7/p541-jiang.pdf

Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *CoRR* abs/1404.5997 (2014).

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS*. 1106–1114.

Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Jeffrey Dean, and Andrew Y. Ng. 2012. Building high-level features using large scale unsupervised learning. In *ICML*.

Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. 1996. Effiicient BackProp. In *Neural Networks: Tricks of the Trade*. 9–50. DOI : http://dx.doi.org/10.1007/3-540-49430-8_2

Tomas Mikolov, Stefan Kombrink, Lukás Burget, Jan Cernocký, and Sanjeev Khudanpur. 2011. Extensions of recurrent neural network language model. In *ICASSP*. IEEE, 5528–5531. DOI : http://dx.doi.org/10.1109/ICASSP.2011.5947611

Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *NIPS*. 3111–3119.

Beng Chin Ooi, Kian-Lee Tan, Sheng Wang, Wei Wang, Qingchao Cai, Gang Chen, Jinyang Gao, Zhaojing Luo, Anthony K. H. Tung, Yuan Wang, Zhongle Xie, Meihui Zhang, and Kaiping Zheng. 2015. SINGA: A Distributed Deep Learning Platform. In *ACM Multimedia*.

Thomas Paine, Hailin Jin, Jianchao Yang, Zhe Lin, and Thomas S. Huang. 2013. GPU Asynchronous Stochastic Gradient Descent to Speed Up Neural Network Training. *CoRR* abs/1312.6186 (2013).

Benjamin Recht, Christopher Re, Stephen J. Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*. 693–701.

Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In *INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014*. 1058–1062.

Heng Tao Shen, Beng Chin Ooi, and Kian-Lee Tan. 2000. Giving meanings to WWW images. In *ACM Multimedia*. 39–47.

Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014). http://arxiv.org/abs/1409.1556

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2014. Going Deeper with Convolutions. *CoRR* abs/1409.4842 (2014).

Kian-Lee Tan, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. 2015. In-memory Databases: Challenges and Opportunities From Software and Hardware Perspectives. *ACM SIGMOD Record* 44, 2 (2015), 35–40.

Ji Wan, Dayong Wang, Steven Chu Hong Hoi, Pengcheng Wu, Jianke Zhu, Yongdong Zhang, and Jintao Li. 2014. Deep Learning for Content-Based Image Retrieval: A Comprehensive Study. In *ACM Multimedia*. 157–166.

Wei Wang, Gang Chen, Tien Tuan Anh Dinh, Jinyang Gao, Beng Chin Ooi, Kian-Lee Tan, and Sheng Wang. 2015. SINGA: Putting Deep Learning in the Hands of Multimedia Users. In *ACM Multimedia*.

Wei Wang, Beng Chin Ooi, Xiaoyan Yang, Dongxiang Zhang, and Yueting Zhuang. 2014. Effective Multi-Modal Retrieval based on Stacked Auto-Encoders. *PVLDB* 7, 8 (2014), 649–660.

Wei Wang, Xiaoyan Yang, Beng Chin Ooi, Dongxiang Zhang, and Yueting Zhuang. 2015. Effective deep learning-based multi-modal retrieval. *The VLDB Journal* (2015), 1–23. DOI : http://dx.doi.org/10.1007/s00778-015-0391-4

Xinxi Wang and Ye Wang. 2014. Improving Content-based and Hybrid Music Recommendation using Deep Learning. In *ACM Multimedia*. 627–636. DOI : http://dx.doi.org/10.1145/2647868.2654940

Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. 2015. Deep Image: Scaling up Image Recognition. *CoRR* abs/1501.02876 (2015). http://arxiv.org/abs/1501.02876

Zuxuan Wu, Yu-Gang Jiang, Jun Wang, Jian Pu, and Xiangyang Xue. 2014. Exploring Inter-feature and Inter-class Relationships with Deep Neural Networks for Video Classification. In *ACM Multimedia*. 167–176.

Omry Yadan, Keith Adams, Yaniv Taigman, and Marc'Aurelio Ranzato. 2013. Multi-GPU Training of ConvNets. *CoRR* abs/1312.5853 (2013).

Quanzeng You, Jiebo Luo, Hailin Jin, and Jianchao Yang. 2015. Joint Visual-Textual Sentiment Analysis with Deep Neural Networks. In *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference, MM '15, Brisbane, Australia, October 26 - 30, 2015*. 1071–1074. DOI : http://dx.doi.org/10.1145/2733373.2806284

Ce Zhang and Christopher Re. 2014. DimmWitted: A Study of Main-Memory Statistical Analytics. *PVLDB* 7, 12 (2014), 1283–1294. http://www.vldb.org/pvldb/vol7/p1283-zhang.pdf

Hanwang Zhang, Yang Yang, Huan-Bo Luan, Shuicheng Yang, and Tat-Seng Chua. 2014. Start from Scratch: Towards Automatically Identifying, Modeling, and Naming Visual Attributes. In *ACM Multimedia*. 187–196.