SINGA: A Distributed Deep Learning Platform

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¶, Kaiping Zheng†
†National University of Singapore, §Zhejiang University, China, ‡NetEase, Inc., China,
¶ Singapore University of Technology and Design
†{ooibc, tankl, wangsh, wangwei, caiqc, jinyang.gao, zhaojing, atung, zhongle,
kaiping}@comp.nus.edu.sg, §cg@zju.edu.cn,
‡wangyuan@corp.netease.com, ¶meihui_zhang@sutd.edu.sg

ABSTRACT

Deep learning has shown outstanding performance in various machine learning tasks. However, the deep complex model structure and massive training data make it expensive to train. In this paper, we present a distributed deep learning system, called SINGA, for training big models over large datasets. An intuitive programming model based on the layer abstraction is provided, which supports a variety of popular deep learning models. SINGA architecture supports both synchronous and asynchronous training frameworks. Hybrid training frameworks can also be customized to achieve good scalability. SINGA provides different neural net partitioning schemes for training large models. SINGA is an Apache Incubator project released under Apache License 2.

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; Distributed training

1. INTRODUCTION

There has been a surge of interests, from both industry and academia, in deep learning. On one hand, deep learning has been shown to achieve (and even surpass) the accuracy of state-of-the-art algorithms in a variety of tasks, e.g., image classification [4] and multi-modal data analysis [9, 10]. On the other hand, to improve runtime performance, distributed training systems have also been proposed in recent years, e.g., Google's DistBelief [1], Torch used by Facebook [7], Baidu's DeepImage [11], Caffe [3] and Purine [6]. These studies showed that deep learning models can benefit from deeper structures and larger training datasets.

However, there are two major challenges to develop a distributed deep learning system. First, deep learning models have a large number of parameters, which would incur huge amount of communication overhead to synchronize nodes when these are updated. Consequently, the scalability in terms of training time to reach certain accuracy is a challenge. Second, it is non-trivial for programmers to develop and train models with deep and complex model structures. Distributed training further increases the burden of programmers, e.g., data and model partitioning, and network communication. This problem is exacerbated if data scientists with little deep learning background are expected to work with deep learning models.

In this paper, we present SINGA[1], a general distributed deep learning platform. SINGA is designed with an intuitive programming model that supports a variety of popular deep learning models, namely feed-forward models including convolution neural networks (CNN), energy models like restricted Boltzmann machine (RBM), and recurrent neural networks (RNN). Many built-in layers are provided for training popular deep learning models. SINGA architecture is sufficiently flexible to run synchronous, asynchronous and hybrid training frameworks. Synchronous training improves the efficiency of one iteration, and asynchronous training improves the convergence rate. Given a fixed budget (e.g., cluster size), users can run a hybrid framework that maximizes the scalability by trading off between the efficiency and convergence rate. SINGA also supports different neural net partitioning schemes to parallelize the training of large models, namely partitioning on batch dimension, feature dimension or hybrid partitioning. GPU support and integration with cluster management software like Mesos will be added in near future.

2. OVERVIEW

The stochastic gradient descent (SGD) algorithm is used in SINGA to train parameters of deep learning models. The training workload is distributed over worker and server units as shown in Figure 1. In each iteration, every worker calls *TrainOneBatch* function to compute parameter gradients.

[1]`http://www.comp.nus.edu.sg/~dbsystem/singa`

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

ACM 978-1-4503-3459-4/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2733373.2807410 .

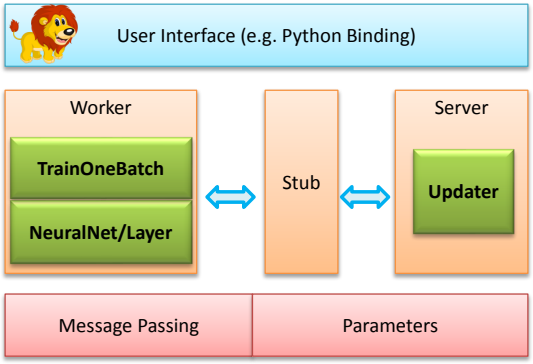

Figure 1: SINGA software stack.

TrainOneBatch takes a *NeuralNet* object representing the neural net, and visits layers of the *NeuralNet* in certain order. The resultant gradients are sent to the local stub that aggregates the requests and forwards them to corresponding servers for updating. Servers reply to workers with the updated parameters for the next iteration. To start a training job, users submit a job configuration consisting of four components: 1) a *NeuralNet* describing the neural net structure with the detailed layer setting and their connections (Section 3.1); 2) a *TrainOneBatch* algorithm which is tailored for different model categories (Section 3.2); 3) an *Updater* defining the protocol for updating parameters at the server side (Section 3.3); 4) a *Cluster Topology* specifying the distributed architecture (Section 4) of workers and servers.

3. PROGRAMMING MODEL

In this section, we describe the first three components of the job configuration. SINGA has provided many built-in implementations for these components. They are also modular and extensible for customization. Users can submit their training job on a single node after configuring these three parts. To conduct distributed training, users need to set the cluster topology as discussed in Section 4.

3.1 NeuralNet

Neural net representation. SINGA represents a neural net using a data structure called *NeuralNet*, which consists of a set of unidirectionally connected layers. Users configure the *NeuralNet* by listing all layers of the neural net and specifying each layer's source layer names. This representation is natural for feed-forward models, e.g., CNN and MLP. For energy models including RBM, DBM, etc., their connections are undirected. To represent these models using *NeuralNet*, users can simply replace each connection with two directed connections. In other words, for each pair of connected layers, their source layer field should include each other's name. For recurrent neural networks, users can remove the recurrent connections by unrolling the recurrent layer. For example, in Figure 2, the original layer is unrolled into a new layer with 4 layers managed inside the new layer. In this way, the model is like a normal feed-forward model, thus can be configured similarly. When SINGA creates the *NeuralNet* instance, it also partitions the original neural net according to user's configuration to support parallel training of large models. Partitioning strategies include:

1. Partitioning all layers into different subsets.

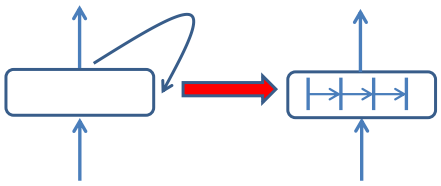

Figure 2: Unroll a recurrent layer into 4 (internal) layers.

```
Layer:
Vector<Layer> srclayer
Blob feature
Func ComputeFeature(phase)
Func ComputeGradient()

Param:
Blob data, gradient
```

Figure 3: Base layer class.

2. Partitioning each singe layer into sub-layers on batch dimension.
3. Partitioning each singe layer into sub-layers on feature dimension.
4. Hybrid partitioning of strategy 1, 2 and 3.

More details on partitioning are discussed in [8].

Layer. Layers of a neural net are represented by instances of Layer classes. Figure 3 shows the base layer class which includes two fields and two functions. The *srclayer* vector records all source layers. The *feature* blob consists of a set of feature vectors, e.g., one vector per image, computed from the source layers. For a recurrent layer in RNN, the feature blob contains one vector per internal layer. If a layer has parameters, these parameters are declared using type *Param* which consists of a *data* and a *gradient* blob for parameter values and gradients respectively. The *ComputeFeature* function evaluates the feature blob by transforming (e.g. convolution and pooling) features from the source layers. *ComputeGradient* computes the gradients of parameters associated with this layer. These two functions are invoked by the *TrainOneBatch* function during training (Section 3.2). SINGA has provided many built-in layers, which can be used directly to create neural nets. Users can also extend the layer class to implement their own feature transformation logics as long as the two base functions are overridden to be consistent with the *TrainOneBatch* function. Besides the common fields like name and type, layer configuration contains some specific fields as well, e.g. file path for data layers. The layers in SINGA are categorized as follows according to their functionalities,

- *Data layers* for loading records (e.g., images) from disk, HDFS or network into memory.
- *Parser layers* for parsing features, labels, etc. from records.
- *Neuron layers* for feature transformation, e.g., convolution, pooling, dropout, etc.
- *Loss layers* for measuring the training objective loss, e.g., cross entropy-loss or Euclidean loss.

- *Output layers* for outputting the prediction results (e.g., probabilities of each category) onto disk or network.
- *Connection layers* for connecting layers when the neural net is partitioned.

3.2 TrainOneBatch

For each SGD iteration, every worker calls the *TrainOneBatch* function to compute gradients of parameters associated with local layers (i.e., layers dispatched to it). SINGA has implemented two algorithms for the *TrainOneBatch* function. Users select the corresponding algorithm for their model in the configuration. Algorithm 1 shows the back-propagation (BP) algorithm [5] for feed-forward models and recurrent neural networks. It forwards (Line 1-3) features through all local layers and backwards gradients in the reverse order (Line 4-6). Since RNN models are unrolled into feed-forward models (*ComputeFeature* and *ComputeGradient* functions compute for all internal layers), Algorithm 1 runs as the back-propagation through time (BPTT) algorithm for them. Algorithm 2 illustrates the contrastive divergence (CD) algorithm [2] for energy models. Parameter gradients are computed (Line 7-9) after the positive phase (Line 1-3) and negative phase (Line 4-6). *kCD* controls the number of Gibbs sampling iterations in the negative phase. In both algorithms, the *Collect* function is blocked until the parameters are fetched from servers. The *Update* function returns immediately after sending the gradients.

Algorithm 1: BPTrainOneBatch

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

3.3 Updater

SINGA comes with many popular protocols for updating parameter values based on gradients. If users want to implement their own updating protocols, they can extend the base Updater to override the *Update* function.

4. DISTRIBUTED TRAINING

4.1 System Architecture

The logical system architecture is shown in Fig. 4. There are two types of execution units, namely workers and servers. Workers compute the parameter updates (e.g., the gradients), while servers maintain up-to-date parameters and handle get/update requests from workers. In each iteration, workers collect fresh parameters from servers and issue update requests to servers after the computation. A number of workers/servers are logically grouped as a worker/server group. A *worker group* loads a subset of the training data and computes the parameter gradients for a complete model replica, denoted as ParamShard. SINGA provides different strategies (i.e., data parallelism, model parallelism and hybrid parallelism) to distribute the workload within a worker group. Workers in the same group run synchronously, while

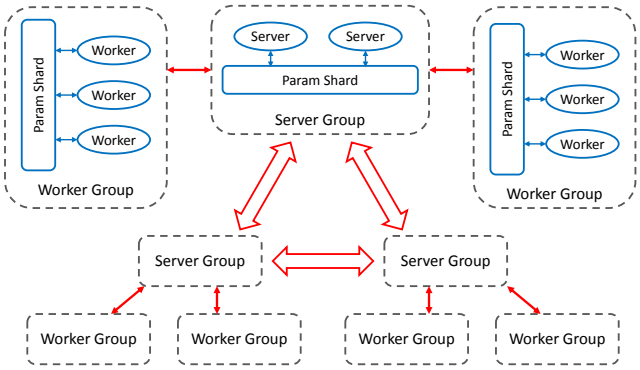

Figure 4: Logical architecture of SINGA.

different worker groups run asynchronously. A *server group* maintains one replica of the full model parameters (i.e., a ParamShard), handling requests from multiple worker groups. Neighboring server groups synchronize their parameters periodically.

Algorithm 2: CDTrainOneBatch

```
   Input: net, kCD
1  foreach layer in net.local_layers do
2      Collect(layer.params()) // receive parameters
3      layer.ComputeFeature(kPostive)// pos phase
4  foreach k in 1...kCD do
5      foreach layer in net.local_layers do
6          layer.ComputeFeature(kNegative) // neg phase
7  foreach layer in net.local_layers do
8      layer.ComputeGradient();
9      Update(layer.params())// send gradients
```

4.2 System Implementation

In SINGA implementation, each execution unit (worker or server) is a thread. As a process may contain multiple threads, there could be multiple (worker/server) groups in one process. It is also possible that one group spans across multiple processes. When starting up a SINGA process, after all execution units are launched, the main thread runs as a stub thread as shown in Figure 1, which aggregates local requests and sends them to remote stubs. Hence, each unit only sends and receives messages from its local stub. SINGA defines general communication APIs on top of ZeroMQ[2] and MPI. Users can choose the underlying implementation (ZeroMQ or MPI) at compile time. If two execution units manage the same parameter partition and they are in the same process, SINGA can leverage the shared memory to reduce communication cost.

4.3 Training Frameworks

SINGA supports various synchronous and asynchronous training frameworks. Users can change the cluster topology configuration to run different frameworks. Here we illustrate how users can train with SINGA using popular distributed training frameworks.

[2]http://zeromq.org/

Sandblaster. This is a synchronous framework used by Google Brain [1]. The training dataset is partitioned into multiple nodes. In one iteration, all nodes fetch up-to-date parameters from parameter servers and sends requests back to update parameters. To run this framework with n nodes, we configure SINGA as follows:

- A single worker group with x workers.
- A single server group with $n - x$ servers.

AllReduce. This is a synchronous framework used by Baidu's DeepImage [11]. There is no concept of server. Each worker computes gradients of one model replica and maintains a subset of parameters. In one iteration, every node fetches up-to-date parameters from all other nodes and sends parameter gradients back to corresponding nodes. To run this framework with n nodes, we configure SINGA as follows:

- A single worker group with n workers.
- A single server group with n servers.
- One worker and one server in each node/process.

Downpour. This is an asynchronous framework used by Google Brain [1]. The training process is similar to Sandblaster. One major difference is that there are multiple groups running asynchronously. Each group of nodes run without awareness of other groups. To run this framework with n nodes, we configure SINGA as follows:

- x worker groups, each with y workers.
- A single server group with $n - x \cdot y$ servers.

Distributed Hogwild. This is an asynchronous framework used by Caffe [3]. Each node maintains a local replica of parameters. In an iteration, each node computes gradients for one model replica and updates them locally. To ensure accuracy and convergence, nodes must exchange updates with others periodically. To run this framework with n nodes, we configure SINGA as follows:

- n worker groups, each with one worker.
- n server groups, each with one server.
- One worker and one server in each node/process.

All training frameworks have their own advantages and limitations. Synchronous training can accelerate the speed for one iteration as the workload is distributed to multiple workers. However, it is limited to small or medium size clusters, because the synchronization delay is large as the cluster size increases. Asynchronous training can improve the convergence rate to some degree, but the improvement becomes small when there are too many model replicas.

SINGA provides users a unified platform for verifying the performance of different frameworks (e.g., convergence rate, computation time) under the same environmental setting. Moreover, users can run hybrid training by launching multiple server groups and worker groups: using multiple server (groups) can alleviate the communication bottleneck at server sides; using multiple worker groups can improve the convergence rate; using multiple workers in a group can accelerate each training iteration. Given a fixed budget (e.g., number of nodes in a cluster), there lies the opportunity to find the optimal hybrid training framework that trades off between the convergence rate and efficiency to achieve the minimal training time.

5. AVAILABILITY

The code and documentation of SINGA is available on the incubator website[3] under Apache License 2. Installation guide and quick start examples are also provided. Sample applications and extensive performance evaluations are presented in [8].

6. 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.

7. REFERENCES

[1] 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.

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

[3] 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.

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

[5] Y. LeCun, L. Bottou, G. B. Orr, and K. Müller. Effiicient backprop. In *Neural Networks: Tricks of the Trade*, pages 9–50, 1996.

[6] M. Lin, S. Li, X. Luo, and S. Yan. Purine: A bi-graph based deep learning framework. *CoRR*, abs/1412.6249, 2014.

[7] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. Fast convolutional nets with fbfft: A GPU performance evaluation. *CoRR*, abs/1412.7580, 2014.

[8] W. Wang, G. Chen, T. T. A. Dinh, J. Gao, B. C. Ooi, K.-L. Tan, and S. Wang. SINGA: Putting deep learning in the hands of multimedia users. *MM*, 2015.

[9] 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.

[10] 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.

[11] R. Wu, S. Yan, Y. Shan, Q. Dang, and G. Sun. Deep image: Scaling up image recognition. *CoRR*, abs/1501.02876, 2015.

[3] `http://singa.incubator.apache.org/`