Large-scale Distributed Convolutional Neural Network — *a Model Parallel Scheme*

Longqi Cai
Andrew ID: longqic
longqic@andrew.cmu.edu

Lei Jin
Andrew ID: ljin1
ljin1@andrew.cmu.edu

Jiang Zhuang
Andrew ID: jzhuang
jzhuang@andrew.cmu.edu

Abstract

Large scale convolutional neural network (CNN) has shown its state-of-the-art accuracy in hard machine learning tasks such as image classification and speech recognition. However, training of CNNs on big data can be tremendously time consuming. In light of the good scalability of parallel computation in distributed systems, distributed CNN has been studied for higher training efficiency. Because of the large amount of data computation in convolutional layers, current distributed CNNs are primarily based on data parallelism, where distributed workers train the same model on different data samples in parallel. To fit large datasets and achieve more complicated abstractions on high dimensional data, large scale CNN becomes a necessary in deep learning, where billions of parameters could be included. As a result, parallel training of such large scale CNNs across the model dimension can also increase the efficiency considerably. In this study, we have proposed a model parallel scheme for distributed training of CNNs and have successfully implemented it in the deep learning framework of *Caffe*. We demonstrate the correctness of the model parallelism scheme on datasets MNIST and CIFAR-10, and the distributed CNN is also expected to achieve higher efficiency with large scale CNN models.

1 Introduction

With the advancement of computation power, deep learning (DL) in practice has shown its capability of learning multiple levels of feature or representations of data, where higher level of features can be derived from lower level ones and thus providing hierarchical abstraction of data. Different types of DL architectures, such as deep neural networks [1], convolutional neural networks [2] and deep belief networks [3], have been successfully applied to fields like image classification, audio recognition and myriads of other areas where they have produced state-of-the-art results on various tasks. In particular, convolutional neural network (CNN) is gaining its importance in high-dimensional tasks such as image classification and speech recognition owning to its special architecture [4, 5, 2]. A typical CNN is composed of one or more convolutional layers (may contain pooling layers) and fully connected layers; the convolutional layers allow CNNs to take advantage of the 2D structure of the input data. CNNs can be trained using standard backpropagation method, and compared to regular deep neural networks, CNNs are easier to train since weights are shared in the convolutional layers, which makes CNN an attractive architecture for applications.

However, large-scale CNNs are still extremely time consuming and computation heavy to train, not only because it requires a significant amount of training data to prevent such large models overfitting on the training dataset, but also due to its huge number of parameters to be trained. The surging volume of data keeps putting pressure on machine learning methods and model training techniques. For example, the Clueweb 2012 web crawl contains over 700 million web pages as 27 TB text, and photo sharing site like Instagram and Pinterest possess tens of billions of images. It could take more than tens of days for a single machine to train a moderate CNN on datasets at these

scales, if possible. Nonetheless, to obtain satisfactory prediction accuracy on such large-scale and high-dimensional tasks, we need to increase size of model and training dataset, which results in tremendous amount of computation cycles.

2 Related Work and Motivation

To increase the learning efficiency on the single-machine basis, GPU computation has been applied to CNN trainings. For example, Krizhevsky et al. trained a CNN across two GPUs using cross-GPU parallelization [2]. They used non-saturating neurons and a very efficient GPU implementation of the convolutional operation. Recently, using model parallel algorithms, Krizhevsky trained a CNN net with up to eight GPUs on a single machine [6]. However, a single machine can only accommodate a limited number of GPU cards, therefore, the single machine based approach has poor scalability with model size and thus limits the size of CNN that can be trained.

To further increase the training efficiency of large DL models such as CNNs, various platforms/frameworks have been applied or developed to parallelize the computation relying on distributed system. MapReduce-based platforms, such as Hadoop [7] and Spark [8], while easy to use, the simplicity of MapReduce abstraction makes them hard to implement the scheduling of computation and communication that are necessary for efficient and correct execution of computation tasks in machine learning [9]. Seunghak Lee et al. developed a system for model parallelism, called STRADS, which provides a programming interface for scheduling model-parallel machine learning algorithms [10]. They demonstrated the efficacy of model-parallel algorithms in topic modeling, matrix factorization, and Lasso based on STRADS. Another recent platform for general distributed machine learning on big data is Petuum, invented by Eric P. Xing et al. [11]. Since these two platforms aim to provide a general-purpose framework applying data and model parallelism to different machine learning models, they are less designed for computation and communication scheduling of distributed DL models like CNN.

Our goal in this project is to develop and implement a model parallel scheme which could exploit the scalability of distributed systems for massively parallel training of CNNs, aiming to optimize the computation and communication scheduling for efficiency and correctness of the distributed CNN. Observe that *Caffe* (developed by Yangqing Jia and BVLC) is an open source deep learning framework with (1) both CPU and GPU realizations, (2) optimized low level data representation, and (3) good extensibility, our implementation of distributed CNN is based on *Caffe*.

3 Method

In general, training of CNN can be parallelized in two dimensions for efficiency [6], namely, the data parallelism and model parallelism. In data parallelism, different workers train the same model on different data samples, and they synchronize on the model (weights). While model parallelism have different workers train different parts of a model, i.e., partial weights of a model, and they synchronized on the neuron activities, which includes data and diff (for backwardpropagation). Observe that in a typical CNN, data computation and model parameters are distributed in biased fashion. The convolutional layers contain around 90% of the whole data computation, but only 5% of the model parameters. As a result, the convolutional layers are data-heavy and thus favors more for data parallelism. On the other hand, 95% of the model parameters lie in the fully connected layers of a CNN, which only consume around 10% of the data computation. Apparently, the fully connected layers are parameter-heavy and hence benefits more if model parallelism is applied. Pioneering work of data parallelism of CNN nets has been done by a few groups including Sailing Lab at CMU. We focus on developing a model parallelism scheme and its implementation as well as analysis, described in detail under the following subsections.

3.1 Model parallelism scheme

The basic concept of model parallelism is splitting the weights of a layer into multiple partitions and distributing each partition to a single worker. For convenience, we define *bottom layer* be the layer fed into a forwardpropagation step and *top layer* be the outcome layer of the forwardpropagation step; the top layer of the previous forwardpropagation step acts as the bottom layer of the next

forwardpropagation step. When we split the top layer and weights as shown in Fig. 1, the weights, w and w', of the two partitions are independent during forward, backward and update computations. Therefore, it is safe to distribute them and the associated computations to different worker nodes. The only dependency in doing forward is the bottom layer, and that in doing backward is the sliced top layer. During the forward process, a complete top layer can be assembled by merging the sliced top layer computed on multiple worker nodes; similarly, a complete bottom layer can be merged from the distributed partitions of the bottom layer in the backward process. Fig. 2 depicts the details of the proposed model parallel scheme for the fully connected layers of a CNN, where the convolutional layers are unchanged.

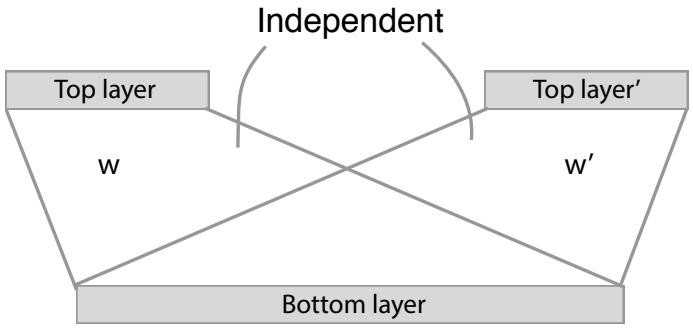

Figure 1: Weights independency across different model partitions.

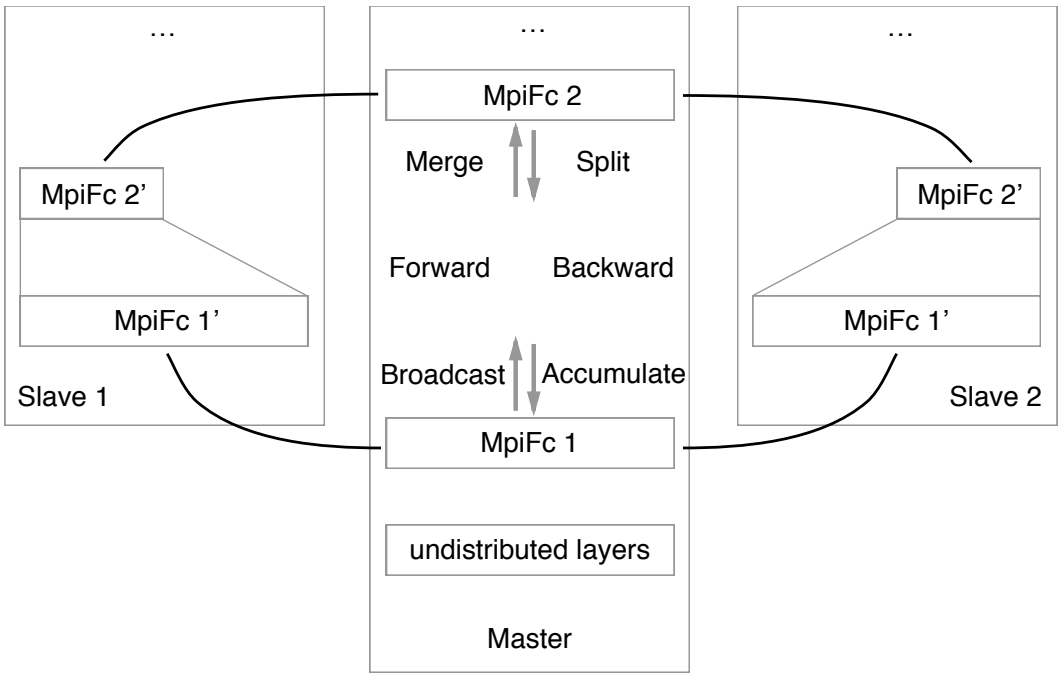

Figure 2: Forward and backward process in the distributed fully connected layers, *MpiFcLayer*.

A master node is assigned to do forward/backward for unmodified layers (e.g., convolutional layers), and to perform scheduling of the MPI layers (i.e., parallelized fully connected layers). Computations in forward/backward of the MPI layers are distributed to multiple slave nodes. When it comes to an MPI layer in the forward process, the master first do a broadcast to distributed the intermediate data to all the slaves. The slaves begin forward propagation on their own partitions of weights in parallel as soon as they received the data. The intermediate outputs on different slaves are then sent back

to the master, and the master merges them and forwards the merged data to the next layer. In the backward process, the master first splits the diff it holds and distributes the partitions of diff to their corresponding slaves. The slaves then perform backpropagation on their own partitions of diff, and send the intermediate diff back to the master. Finally, the master sums up the diffs from all slaves and continues to start the next backpropagating step. Upon the completion of a forward-backward process on a batch of data, the model is updated based on the sum of diff on this data batch.

In this scheme, the forward, backward and weights update are done in parallel on different slave nodes. Operation of data merge and diff accumulation are synchronized. It can be seen that the intermediate data are duplicated on master and slave nodes, but the weights of *MpiFc* layers are distributed on multiple slaves by a certain parallelism coefficient. Because the forward and backward process are synchronized from layer to layer, the overheads of this scheme are mainly from the idle slave nodes when synchronization happens on the master.

3.2 Implementation

This scheme is implemented based on *Caffe*[12] and MPI (Message Passing Interface). Low-level data representation (*Blob*) of *Caffe* is based on *Protobuf*, which is not only memory efficient, but also makes it easy for data serialization and deserialization. Neural network in *Caffe* is also designed to be easily configurable. However, the high level data structure, (*Net*), is designed and implemented on single machine-basis, where all data is located in a shared-memory, and thus the original *Caffe* is not able to run with distributed architectures.

We extended the *Caffe* framework with the class models shown in Fig. 3.

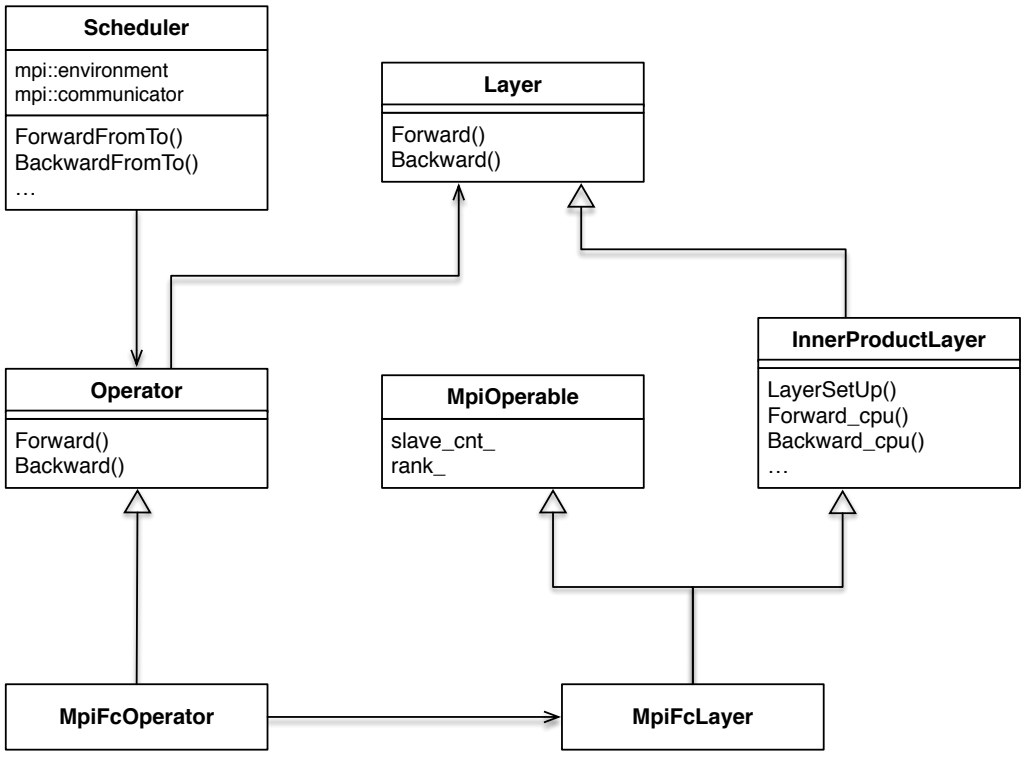

Figure 3: Object-oriented design for the MPI functionalities of the fully connected layers in the framework of *Caffe*.

The first refactorization is to extract *ForwardFromTo* and *BackwardFromTo* into a new class called *Scheduler*, along with any necessary member functions for communication, such as *Send*, *Recv*, and

Broadcast. The *Scheduler* is designed to be a singleton since MPI routines such as *rank* and *size* are called from many places. Secondly, we designed and implemented a new type of layer, called *MpiFcLayer*, mainly to differentiate different worker nodes when doing layer setup. The third major class is *Operator*, which serves as a wrapper class of *Layer* to do *Forward* and *Backward*, along with MPI operations such as *Send* and *Recv*. *Scheduler* delegates operation on each layer by the *Operator* class, and uses a factory method to construct this class.

The aforementioned distributed forward/backward process is implemented in *MpiFcOperator*. Detailed operations in these two procedures is shown in Algorithm 1 and 2.

Algorithm 1 Forward

procedure FORWARD
  if is master **then**
    Broadcast $bottom\_vecs$ to slaves
    Recv $sliced\_top\_vecs$ from slaves
    Merge $sliced\_top\_vecs$ into top_vecs
  else
    Broadcast $bottom\_vecs$ from master
    Call $Forward()$ on layer
    Send $sliced\_top\_vecs$ to master
  end if
end procedure

Algorithm 2 Backward

procedure FORWARD
  if is master **then**
    Split top_vecs into $sliced\_top\_vecs$
    Send $sliced\_top\_vecs$ to slaves
    Reset diff's on $bottom\_vecs$
    Recv & accumulate $bottom\_vecs$ from slaves
  else
    Recv $sliced\_top\_vecs$ from master
    Call $Backward()$ on layer
    Send $bottom\_vecs$ to master
  end if
end procedure

3.3 Cost analysis

Assume we have K slaves and a master. Batch size is B. Bottom layer has n_b outputs and top layer has n_t outputs. Table 1 summaries the cost of a fully connected layer before and after being distributed. The computation on a single worker node is apparently reduced. However, when the model is distributed to multiple machines under the real settings, the overall performance w.r.t the training efficiency of the CNN model is dependent on the relative cost of computation and communication per data size. Faster communication favors more for distributed training over single machine based training, and vice versa. Nonetheless, the distributed scheme is really designed for training large scale models. If you imagine a CNN with billions of parameters, distributing the parameters to different worker nodes becomes very meaningful and necessary. As the weights are distributed into multiple worker nodes, a single worker node can potentially keep all parameters in memory and likely to hold a larger batch size for higher throughput.

4 Experiment and Results

4.1 Correctness

Datasets MNIST and CIFAR-10 have been used train CNNs in order to test the correctness of the model parallelism scheme and its implementation. By plotting the testing accuracy along the training

	Before		After	
	Comp.	Comm.	Comp.	Comm.
Forward	$B \times n_b \times n_t$	0	$B \times n_b \times \frac{1}{K} n_t$	$B \times (K \times n_b + n_t)$
Backward	$B \times n_b \times n_t$	0	$B \times n_b \times \frac{1}{K} n_t$	$B \times (K \times n_b + n_t)$
Update	$n_b \times n_t$	0	$n_b \times \frac{1}{K} n_t$	0

Table 1: Computation and communication cost of a fully connected layer before and after being distributed. The cost is only about the data size being computed/communicated; the actual speeds of computation and communication may depend on real systems and thus not accounted.

process, Fig. 4 and Fig. 5 show the convergence time of the respective CNN models under different settings. For both datasets, the convergence trace of the distributed CNN is very similar to the single-process *Caffe* under a same batchsize of 64. In MNIST case, a testing accuracy close to 99.5% can be achieved within 250 seconds, while in CIFAR-10, the testing accuracy converged to 72.3% within 1400 seconds. The small discrepancies between distributed CNN and single-process *Caffe* under the same batchsize are mainly because of random error. Therefore, the their agreement on the convergence curve proves the correctness of our model parallelism scheme and its implementation in the framework of *Caffe*.

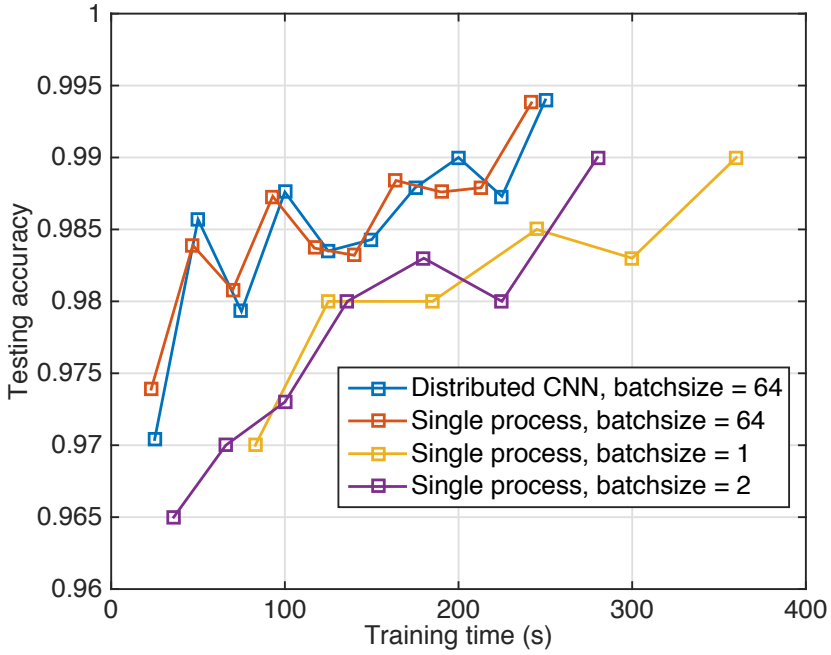

Figure 4: Training time of LeNet on dataset MNIST.

4.2 Effect of batchsize

Besides the basic correctness, we also studied the dependency of training time on the batchsize, which results corroborate that a reasonably large batchsize could help on increasing efficiency. It should be pointed out that, although we decreased the batchsize, the actual frequency of weight-update per data samples was kept constant. This means that smaller batchsize needs more iterations for a weight update. It can be seen that, for both cases shown in Fig. 4 and Fig. 5, when the batchsize decreased to 1, the efficiency drops considerably. One reason of the the efficiency decrease under small batchsize could be overheads of many iterations. For example, increased number of iterations increases the number of function calls, which is associated with register saving, pushing

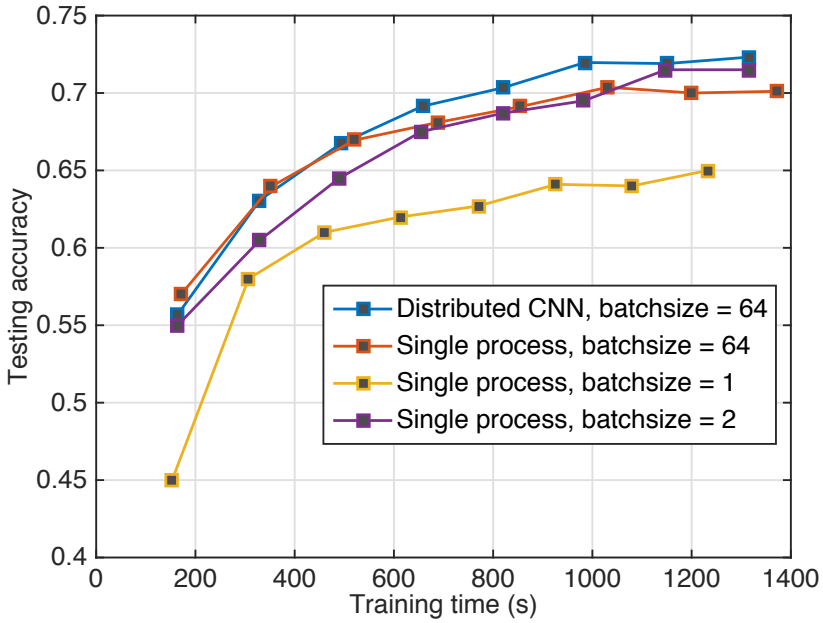

Figure 5: Training time of a CNN on dataset CIFAR-10.

arguments stack, etc. Another important reason is that larger batchsize could make better use of SIMD instructions included in the mordern CPU designs.

When the model scales up, in which case the model parameters could occupy a large portion of memory, there is not enough space to hold a reasonably large batch of data in memory, and therefore multiple iterations is the only choice. Since the model parallelism scheme could parallelize the model parameters to multiple worker nodes, less memory space would be occupied by the parameters, which saves more memory for data, and thus the whole batch of data is likely to be processed with a single iteration. Therefore, the degraded efficiency at lower batchsize indicates the potential of achieving higher throughput with distributed CNN when the model size is large.

5 Discussions

In our parallelism scheme, the master node coordinates with all the slaves for data synchronization. While this approach is straightforward, there might be associated issues when the parallelism becomes high. One possible problem is that the communication speed of the master could be a bottleneck of efficiency. Assuming the total size of data needs to be broadcasted by the master is S and there are k slave nodes, during a forward step as shown in Fig. 2, each slave node needs to send S/k data and receives S data, while the master node needs to receive S data and send kS data. Such master-slave configuration makes the master node communication-heavy. Fig. 6 illustrates a possible way to optimize the communication, where each worker node directly sends its partition of calculated data to all other workers. In this case, each worker sends S/k data and receives $(k-1)S/k$ data, making the communication cost of each node on the order of S. However, this method requires each worker nodes to keep information of all other nodes, which may introduce overhead on bookkeeping.

6 Conclusion and Future Work

A model parallel scheme for distributed CNN has been proposed and implemented successfully in the deep learning framework of *Caffe*. Based on the convergence comparison of CNNs trained by the distributed CNN and single-process *Caffe* on standard datasets, we have demonstrated the

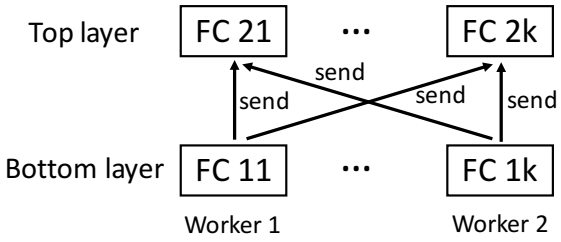

Figure 6: Another scheme of communication.

correctness of the proposed model parallelism scheme. In addition, dependency of training time on the batchsize indicates that the distributed CNN is potentially to achieve higher throughput in training of large scale CNN models.

We suggest the following future works on this project:

- Optimizing the communication and computation scheduling for higher efficiency
- Exploiting GPU computing in distributed CNN for higher speed

References

[1] G. Hinton, L. Deng, D. Yu, G. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *Signal Processing Magazine, IEEE*, vol. 29, pp. 82–97, Nov 2012.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), pp. 1097–1105, Curran Associates, Inc., 2012.

[3] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[4] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, 1995.

[5] S. Lawrence, C. Giles, A. C. Tsoi, and A. Back, "Face recognition: a convolutional neural-network approach," *Neural Networks, IEEE Transactions on*, vol. 8, pp. 98–113, Jan 1997.

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

[7] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st ed., 2009.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[9] W. Dai, A. Kumar, J. Wei, Q. Ho, G. A. Gibson, and E. P. Xing, "High-performance distributed ML at scale through parameter server consistency models," *CoRR*, vol. abs/1410.8043, 2014.

[10] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing, "On model parallelization and scheduling strategies for distributed machine learning," in *Advances in Neural Information Processing Systems 27* (Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, eds.), pp. 2834–2842, Curran Associates, Inc., 2014.

[11] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, (New York, NY, USA), pp. 1335–1344, ACM, 2015.

[12] 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.