RaPyDLI Deep Learning Framework Report 2016

Goals

Deep learning methods have led to significant progress in large-scale applications (e.g., speech, language, vision) and show promise in different application domains. As deep learning and its utilization continues to matures, so does the infrastructure and software needed to support it. Various frameworks have been developed in recent years to facilitate both implementation and training of deep learning networks. As deep learning has also evolved to scale across multiple machines, there's a growing need for frameworks that can provide full parallel support. While deep learning frameworks restricted to running on a single machine have been studied and compared, frameworks which support parallel deep learning at scale are relatively less well-known and studied. To bridge the gap, we survey, summarize, and compare frameworks which currently support distributed execution, including but not limited to Tensorflow, CNTK, Deeplearning4j, MXNet, H2O, CaffeOnSpark, Theano, and Torch.

Framework Comparison

Tensorflow, CNTK, Deeplearning4j, MXNet, H2O, CaffeOnSpark, Torch, and Theano were deep learning frameworks, chosen for their traction among other factors, for detailed study. They were compared according to a consistent set of characteristics, ranging from parallelism at the hardware and application level, to other information such as release date, core language, API, computation and synchronization models, programming paradigm, fault tolerance, and visualization. These findings have been summarized in Table 1.

The relevance of release date, core language, and user-facing APIs are self-explanatory. Synchronization model specifies the nature of data consistency through execution, i.e. whether updates are synchronous or asynchronous. In the context of optimization kernels like stochastic gradient descent (SGD), synchronous execution has better convergence guarantees by maintaining consistency or near-consistency with sequential execution. Asynchronous SGD can exploit more parallelism and train faster, but with less guarantees of convergence speed. Frameworks like Tensorflow and MXNet leave this tradeoff as a choice to the user.

The computation model tries to categorize the nature of across-machine execution according to well-known paradigms. There are three possible levels of parallelism at the hardware level: cores within a CPU/GPU device, across multiple de- vices (usually GPUs for deep learning), or across machines. Most lower-level library kernels (e.g. for linear algebra) are designed to use multiple cores of a device by default, so this is not a major point of comparison. At this point, all the frameworks also support parallelism across multiple GPUs. Theano and Torch do not yet support multi-machine parallelism.

Data and model parallelism are the two prevalent classes for parallelism in training deep learning networks at the distributed level. In data parallelism, copies of the model, or parameters, are each trained on its own subset of the training data, while updating the same global model. In model parallelism, the model itself is partitioned and trained in parallel.

Deep learning models can be categorized into three major types: deep-belief networks (DBNs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). CNNs and RNNs were briefly described in the introduction. DBNs are less domain-specific compared to CNNs and RNNs, and could be considered a precursor to CNNs, but are fundamental nonetheless.

Programming paradigm falls into the categories of imperative, declarative, or a mix of both. Conventionally, imperative programming specifies *how* a computation is done, whereas declarative programming specifies *what* needs to be done. There is plenty of gray area, but the distinction made here is based on whether the API exposes the user to computation details that require some understanding of the inner math of neural networks (imperative), or whether the abstraction is yet higher (declarative).

Fault tolerance is included for two reasons. Distributed execution tends to be more failure prone, especially at scale.

Furthermore, any failures (not necessarily limited to distributed execution) that interrupt training part-way can be very costly, especially if all the progress made on the model is simply lost as a result.

Finally, UI/Visualization is a feature supported to varying degrees across the frameworks studied. The ability to monitor the progress of training and the internal state of networks over time could be useful for debugging or hyperparameter tuning, and poses an interesting direction of research. Tensorflow and Deeplearning4j both support this kind of visualization.

Table 1: Deep Learning Open-source Frameworks

Platform	**Tensorflow**	**CNTK**	**Deeplearning4j**	**MXNet**	**H2O**	**CaffeOnSpark**	**Theano**	**Torch**
Release Date	2016	2016	2015	2015	2014	2016	2010	2011 (deep learning)
Computation Model	Parameter server	MPI	Iterative MapReduce	Parameter server	Distributed fork-join	MPI Allreduce	Single node	Single node
Parallelism	Data & Model	Data	Data	Data & Model	Data	Data	Data & Model	Data & Model
Synchronization Mechanism	Sync or async	Sync	Sync	Sync or async	Async	Sync	Async	Sync
Deep Learning Models	DBN, CNN, RNN	DBN,CNN, RNN	DBN, CNN, RNN	DBN, CNN, RNN	DBN	DBN, CNN, RNN	DBN,CNN, RNN	DBN,CNN, RNN
Core Language	C++	C++	Java	C++	Java	C++, Scala	C++	C
API	C++, Python	NDL	Java, Scala	C++, Python, R, Scala, Matlab, Javascript, Go, Julia	Java, R, Python, Scala, Javascript, web-UI	Python, Matlab, Scala	Python	Lua
Programming Paradigm	Imperative	Imperative	Declarative	Both	Declarative	Declarative	Imperative	Imperative
Fault Tolerance	Checkpoint-and-recovery	Checkpoint-and-resume	Checkpoint-and-resume	Checkpoint-and-resume	N/A	N/A	Checkpoint- and-resume	Checkpoint- and-resume
Visualization	Graph (in-teractive), training monitoring	Graph (static)	Training monitoring	N/A	N/A	Summary Statistics	Graph (static)	Plots

Future Work

Scaling Deep Learning

Distributed frameworks operate under the basic premise that easily scaling up to more machines is important for scaling up to bigger problems. At heart the goal is to utilize parallelism effectively. While execution on more machines can help, there are also other important factors, namely the underlying hardware and application characteristics, which can escape the abstraction of a general-purpose framework. One example of this is the difference in Google's and Stanford's approaches in training a large-model convolutional auto-encoder network. Whereas Google's seminal billion-parameter model was trained using thousands of machines for several days, Stanford demonstrated training the 11 billion-parameter version of a similar model using a tiny fraction of the hardware in 3 days [1].

They used a HPC cluster of 16 machines, with 4 NVIDIA GPUs each. Communication was via MPI on top of fast Infiniband interconnects. The results showed that the application characteristics, namely a convolutional neural network (with auto-encoding) with 200x200 images as input, limited the amount of model parallelism that could be extracted from mapping partitions of the images to different GPUs. Therefore, it seems that a small cluster of 16 machines, or 64 GPUs, was optimal enough. Taking the hardware concept further, NVIDIA now has a state-of-the-art server (DGX-1) consisting of 8 Tesla P100 GPUs (over 28,000 CUDA cores), optimized for deep learning [2]. The communication network features a direct-connect topology for inter-GPU communication. It is certainly possible that such specialized hardware could sufficiently handle certain deep learning problems, at scale, without need of a second machine.

On the application side, it also is not necessarily the case that greater model size correlates with higher model accuracy. In fact, a fairly state-of-the-art convolutional neural network like GoogleNet [30] achieves similar accuracy to other networks that use far more parameters. So while software frameworks for deep learning provide helpful abstractions both for constructing networks and running them at scale, necessary attention must also be paid to underlying hardware and application-specific characteristics to in order to effectively utilize parallelism.

Tensorflow was released by Google Research as open source in November 2015, and included distributed support in 2016. The user-facing APIs are C++ and Python. Programming with Tensorflow leans more towards the imperative approach. While plenty of abstraction power is expressed in its library, the user will probably also be working with computational primitive wrappers such as matrix operations, element-wise math operators, and looping control. In other words, the user is exposed to some of the internal workings of deep learning networks. Tensorflow treats networks as a directed graph of nodes encapsulating dataflow computation and required dependencies [4]. Each node, or computation, gets mapped to devices (CPUs or GPUs) according to some cost function. This partitions the overall graph into subgraphs, one per device. Cross-device edges are replaced to encode necessary synchronization between device pairs. Distributed execution appears to be a natural extension of this arrangement, except that TCP or Remote Direct Memory Access (RDMA) is used for inter-device communication on separate machines. This approach of mapping subgraphs onto devices also offers potential scalability, because each worker can schedule its own subgraph at runtime instead of relying on a centralized master [4]. Parallelism in Tensorflow can be expressed at several levels, notably including both data parallelism and model parallelism. Data parallelism can happen both across and within workers, by training separate batches of data on model replications. Model parallelism is expressed through splitting one model, or its graph, across devices. Model updates can either be synchronous or asynchronous for parameter-optimizing algorithms such as SGD. For fault tolerance, Tensorflow provides checkpointing and recovery of data designated to be persistent, while the overall computation graph is restarted. In terms of other features, TensorBoard is a tool for interactive visualization of a user's network, and also provides time series data on various aspects of the learning network's state during training.

While deep learning frameworks such as Tensorflow provide abstraction and many are designed to scale up to many machines, there is evidence that some deep learning problems can be solved efficiently and accurately with a small to medium sized cluster of machines, given the right utilization of specialized hardware and attention to application-specific characteristics. In future work, we will look into utilizing HPC technologies to improve parallel computation and communication efficiency of Deep Learning frameworks, where we have demonstrated good results in modeling iterative computations for other machine learning applications [5] [6].

Acknowledgements

We gratefully acknowledge that this project is partially supported from NSF 14-39007 RaPyDLI grant and NSF 14-43054 SPIDAL grant.

References

[1] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. arXiv preprint arXiv:1512.01274, 2015.

[2] Gp100 pascal whitepaper. https://images. nvidia.com/content/pdf/tesla/whitepaper/ pascal-architecture-whitepaper.pdf. (Accessed on 07/27/2016).

[3] D. Yu, K. Yao, and Y. Zhang. The computational network toolkit [best of the web]. IEEE Signal Processing Magazine, 32(6):123–126, 2015.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. J´ozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Man´e, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Vi´egas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. CoRR, abs/1603.04467, 2016.

[5] B. Zhang, Y. Ruan, and J. Qiu, "Harp: Collective Communication on Hadoop," in IC2E, 2015.

[6] B. Zhang, B. Peng, and J. Qiu, "High Performance LDA through Collective Model Communication Optimization," in ICCS, 2016.