Network Evolution for DNNs

Michael Alan Chang
UC Berkeley

Aurojit Panda
NYU, ICSI

Domenic Bottini
UC Berkeley

Lisa Jian
UC Berkeley

Pranay Kumar
UC Berkeley

Scott Shenker
UC Berkeley, ICSI

1 INTRODUCTION

Deep neural networks (DNNs) power a wide range of modern applications, including autonomous vehicles, datacenter schedulers, search engines, etc. Many of these applications must meet responsiveness and accuracy bounds; consequently, training and serving performance for these models has become crucial. The community has addressed these performance requirements in several ways including: (a) building specialized data flow systems such as TensorFlow [1] and MxNet [2]; (b) deploying hardware accelerators such as GPUs, FPGAs, TPUs, etc. to accelerate training and serving; (c) developing and adopting distributed training algorithms that allow model training to be scaled across several physically separated servers in a data center; etc. These efforts are complimentary and are commonly deployed together, *e.g.,* most specialized ML frameworks implement distributed training algorithms which distribute training tasks across servers thereby parallelizing training, and make use of hardware accelerators to improve execution time for individual tasks.

Existing proposals for improving DNN training performance have however largely ignored the network fabric. This gap in the literature is particularly surprising given that most machine learning models are relatively large, ranging from the 120MB Inception-v3 [14] model to the 820MB VGG-16 [13] model, and must be transferred in entirety twice each training iteration. Furthermore, recent advances in software-defined networking and programmable networks have made it feasible to implement optimizations that target the network. Given these observations we ask two questions:

(a) **Does the network fabric affect DNN training performance?** We address this question in §2 and show through analytical arguments and empirical data that network performance impacts DNN training time, and that this impact increases as a function of the number of workers, limiting scalability.

(b) **Given the previous observation, what network services can most improve DNN training performance?** As we discuss in §3, distributed training algorithms make use of the network during an *aggregation* phase – where data from multiple workers is aggregated to compute an updated model – and a *distribution phase* – where the updated model is sent to each worker. We empirically measure the possible benefit from improving network overheads for each stage, and show that improving distribution leads to better gains than improving aggregation. Additionally we also find that while improving the aggregation phase requires adding application specific functionality to the network, the distribution phase can be improved using existing mechanisms, and requires minimal network changes.

Therefore our analysis indicates that optimizing the network fabric can improve DNN training time, and techniques targeting the distribution phase of learning algorithms provide greater improvements than ones targeting the aggregation phase.

2 DOES THE NETWORK AFFECT DNN TRAINING PERFORMANCE?

We begin by analyzing what impact the network has on DNN training time. In this section, and through the rest of this paper we focus our analysis on distributed training algorithms that make use of parameter servers [9] and are synchronous [1]. Each iteration of these training algorithms proceeds as follows: each worker randomly samples training data and uses this sample to train a model, then sends this updated model to one or more *parameter servers*. Each parameter server waits to receive updates from all workers, at which point it *aggregates* these updates to generate a new model. Finally once the parameter server has computed a new model it *distributes* this model to all workers, who then begin working on the next iteration. Note that in this computational model the coordination required is that the parameter server wait for updates from all workers. As a result this model is amenable to *pipelining* and in most current implementations, workers begin training (forward propagation) as soon as possible without waiting to receive the entire model, and send updates for each layer as soon as they are available (*i.e.,* as soon as back-propagation is complete through a layer). Pipelining in the backpropagation step has a more significant impact since this step typically takes longer [4]; as a result, our analysis only factors in pipelining during backpropagation.

We consider the following three effects when analyzing if and how the network affects training time: processing cycles spent transferring data, effects due to latency, and effects due to throughput.

Processing Cycles In existing ML frameworks there is a connection between each parameter server and worker. During aggregation each worker must send an updated model to each parameter server, requiring the ML framework to initiate $n \cdot w$ transfers (across nodes) in the case where there are n parameter servers, and w workers. Similarly during the distribution phase each parameter server must send to each of w workers, requiring $n \cdot w$ transfer. Initiating a transfer consumes CPU cycles (either to call into a message passing library, or to queue up messages in a message queue) and thus we find that the number of CPU cycles required for network transfers in current frameworks grows with both number of workers and parameter servers.

Latency Each parameter server must wait for an update from all workers, and the wait time is affected by network latency which therefore also affects iteration time. Network latency is determined both by distance between machines – which determines propagation delay – and the number of pending packets in the network

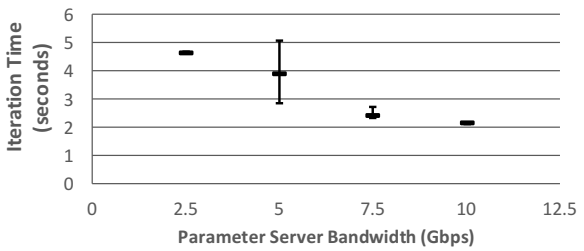

Figure 1: Running the Inception-v3 model shows how varying the parameter server link bandwidth affect training performance.

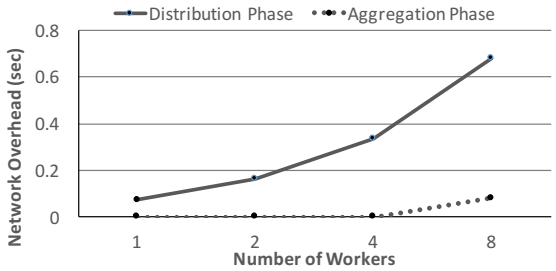

Figure 2: Remaining time to send network-queued parameters at conclusion of the respective phase, assuming 10Gbps parameter server network

– which dictates the time each packet spends in network queues. While propagation delay is unaffected by application, the number of pending packets is proportional (as shown above) to both the number of parameter servers and workers. Furthermore, queuing delays are also affected by the choice of congestion control and packet scheduling algorithms for the network.

Throughput Finally, the total amount of data transferred during each training iteration depends on both the model size (m) and number of workers, and is given by $2 \cdot m \cdot w$. Models can be hundreds of megabytes in size – *e.g.,* VGG-16 is 820MB, and Inception-v3 is 120MB – and hence several 100 megabytes of data need to be transferred during each iteration. Furthermore, the time required to transfer this data depends on network throughout, which consequently impacts training time. Furthermore, distributed training is not beneficial when network transfer times exceed computation time at each worker, and as a result network throughput imposes scaling limits on DNN training.

We validated the impact of network throughput on training performance through empirical evaluation where we measured training time for the *Inception-v3* model as we varied network network bandwidth. We ran our evaluation on Amazon EC2 [7], and used 8 *g3.4xlarge* instances as workers and one *c3.8xlarge* as a parameter server. Each workers was connected to the network using a 3.5Gbps link, while the parameter server was connected using a 10Gbps link. We implemented and trained the model using TensorFlow 1.4, and used Linux `tc` to vary the parameter server bandwidth. Our results, shown in Figure 1, show that throttling network throughput to 2.5Gbps results in a 2.5× increase in training time, validating the impact of network throughput on DNN training time.

3 BREAKING DOWN DNN COMMUNICATION

Having established that the network affects DNN training performance, we now take a closer look at how these jobs utilize the network and compare strategies for reducing network overhead. As seen above, each training iteration includes two communication phases: a *distribution phase* where the parameter server sends an updated model to all workers, and a *aggregation phase* where each worker sends updates to the parameter server. Additional network functionality can improve performance for both these phases. Network primitives such as *IP multicast* [11] that allow a sender to transmit data to multiple receivers can be used to reduce distribution overheads by reducing the number of messages sent by each parameter server (thus reducing CPU cycles required), reducing bandwidth requirements for the parameter server and reducing network congestion. Similarly programmable switches [8] and NFs [5] can be used to reduce aggregation overheads by offloading some aggregation functionality to the network [10, 12].

These techniques differ in the requirements they place on the network, ease of deployment and expected benefits:

Requirement: IP multicast is widely implemented in current switches [3], and adopting this technique requires minor changes to the network configuration and DNN implementation. On the other hand use of in-network aggregation requires adoption of new programmable switches, which are not yet widely deployed, and changes to the packet processing functionality implemented by these switches.

Deployability: Multicast based approaches can be deployed in any network fabric where multicast is enabled and requires no additional application specific configuration. In contrast, when using in-network aggregation network administrators must ensure that the network and application agree on aggregation semantics. This need for agreement adds significant complexity when deploying or modifying DNN training applications.

Benefits: We compare the benefit of improving aggregation and distribution by empirically measuring the network overheads for each phase. These overheads provide an upper bound on the maximum improvement that can be expected when addressing either of these phases. We used the same setup as in §2 to perform these measurements, and show our result in Figure 1. We find that network overheads are higher during the distribution phase, and these overheads increase as we increase the number of workers, showing that when training Inception-v3, the distribution phase is a scaling bottleneck. We observed similar trends for Resnet-200 [6] and VGG-16, and believe that this trend holds across a variety of models.

The above observations indicate that it is easier to improve the distribution phase for distributed learning algorithms, and improvements to this phase have a greater impact on training performance.

4 CONCLUSION

In this paper we have argued that networks should evolve to better support DNN training jobs, and that this evolution does not require adding new hardware or specializing the network. While the network is only one among several factors affecting DNN performance, optimizing the network is essential to ensuring DNN scalability and allowing the use of ever larger, and more complex models.

REFERENCES

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

[2] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR abs/1512.01274* (2015).

[3] Catalyst 6500: Configuring IP Multicast Layer 3 Switching. https://www.cisco.com/c/en/us/td/docs/routers/7600/ios/12-1E/configuration/guide/swcg/mcastmls.pdf, retrieved 01/05/2017.

[4] CNN-benchmarks. https://github.com/jcjohnson/cnn-benchmarks, retrieved 01/03/2017.

[5] ETSI. Network Functions Virtualisation. Retrieved 07/30/2014 http://portal.etsi.org/NFV/NFV_White_Paper.pdf.

[6] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. *CoRR abs/1512.03385* (2015).

[7] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, retrieved 01/03/2017.

[8] Barefoot Tofino. https://barefootnetworks.com/products/brief-tofino/, retrieved 01/03/2017.

[9] Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 583–598.

[10] Luo, L., Liu, M., Nelson, J., Ceze, L., Phanishayee, A., and Krishnamurthy, A. Motivating in-network aggregation for distributed deep neural network training. In *Workshop on Approximate Computing Across the Stack* (2017), ACM.

[11] Ratnasamy, S., Ermolinskiy, A., and Shenker, S. Revisiting ip multicast. In *SIGCOMM* (2006).

[12] Sapio, A., Abdelaziz, I., Canini, M., and Kalnis, P. Daiet: A system for data aggregation inside the network. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, ACM, pp. 626–626.

[13] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *CoRR abs/1409.1556* (2014).

[14] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. Rethinking the inception architecture for computer vision. *CoRR abs/1512.00567* (2015).