Optimizing Network Performance in Distributed Machine Learning

Luo Mai
Imperial College London

Chuntao Hong
Microsoft Research

Paolo Costa
Microsoft Research

Abstract

To cope with the ever growing availability of training data, there have been several proposals to scale machine learning computation beyond a single server and distribute it across a cluster. While this enables reducing the training time, the observed speed up is often limited by network bottlenecks.

To address this, we design MLNET, a host-based communication layer that aims to improve the network performance of distributed machine learning systems. This is achieved through a combination of traffic reduction techniques (to diminish network load in the core and at the edges) and traffic management (to reduce average training time). A key feature of MLNET is its compatibility with existing hardware and software infrastructure so it can be immediately deployed.

We describe the main techniques underpinning MLNET and show through simulation that the overall training time can be reduced by up to 78%. While preliminary, our results indicate the critical role played by the network and the benefits of introducing a new communication layer to increase the performance of distributed machine learning systems.

1 Introduction

Over the last decade, machine learning has witnessed an increasing wave of popularity across several domains, including web search, image and speech recognition, text processing, gaming, and health care. A key factor causing this trend is the availability of large amounts of data that can be used for training purposes. This has led to the appearance of several proposals aiming at scaling out the computation by distributing it across many servers [1, 13, 16, 24, 34, 35].

Typically, these systems adopt an approach referred to as *data parallelism* [16]. Rather than training a single model with all the available input data, they replicate the model across many servers and feed each replica with a subset of the input data. Since the model replicas are trained using different input data, their model parameters will typically diverge. To reconcile these parameters and ensure that all model replicas eventually converge, each replica periodically pushes its set of parameter values to a centralized server, called the *parameter server* [24]. The latter aggregates all the received updates for each parameter (e.g., by averaging them) and then sends back to all replicas the newly computed set of values, which will be used at the beginning of the next iteration. As the total numbers of parameters can be very high (up to 10^{12} [9]), multiple parameter servers are used, with each one being responsible for a subset of the parameters.

A major challenge of this approach is the high communication cost. Model replicas must frequently read and write global shared parameters. This generates a large amount of network traffic and, due to the sequential nature of many of the machine learning algorithms used, it may also stall the computation if the synchronization latency is high. Therefore, the network is often regarded as one of the main bottlenecks for distributed machine learning systems [13, 16, 25]. To alleviate this issue, these systems are often deployed on high-performance network fabrics such as Infiniband or RoCE [13, 23], while others have proposed to trade-off algorithm training efficiency for system performance by introducing asynchronous communication [10, 24], thus removing some of the barriers. Unfortunately, neither of these approaches is completely satisfactory as the former significantly increases infrastructure costs while the latter reduces overall training efficiency.

In this paper, we explore a different yet complementary point of the design space. We argue that network bottlenecks can be greatly reduced through a customized communication layer. To demonstrate this, we designed MLNET, a novel communication layer for distributed machine learning. MLNET uses tree-based overlays to implement distributed aggregation and multicast and reduce network traffic, and relies on traffic control and prioritization to improve average training time.

A key constraint underlying our design is that we wanted MLNET to be a drop-in solution for existing machine learning deployments. Therefore, we implemented MLNET as a user-space process running on hosts without requiring any changes in the networking hardware, in the OS stack, or in the training algorithm code. Further, by sitting in between workers and parameter servers, it decouples these two classes of servers, enabling scaling each one independently and efficiently masking network and server failures. We evaluate its effectiveness in Section 4 by means of large-scale simulations with 800 servers and 50 to 400 parameter servers. The results show that MLNET reduces the training time by a factor of up to 5x.

2 Background

In this section, we provide a brief introduction to machine learning, motivate the need for its distributed execution, and discuss the use of parameter servers for scaling distributed machine learning.

2.1 Machine Learning

The goal of a machine learning algorithm is to construct a prediction model that extracts useful knowledge from training data, and uses it to make inferences about future arrival data. This can be formalized as an optimization problem: Given a set of training data X, it tries to find a model W that minimizes the error of a prediction function $F(X,W)$. Typically, a machine learning algorithm approaches this problem iteratively, starting from a randomly generated W and then refining its solution gradually as more and more data are processed.

Complex models are usually able to capture the knowledge hidden in training data. To an extreme, a sufficiently complex model can "memorize" all the information contained in the data. In this case, it can give the correct prediction for any sample it has seen before, but may perform poorly for unseen samples. This is called *over-fitting*: a model fits its training data well, but does not generalize to others. This is why a large amount of training data is necessary for machine learning. By using more data, a model can generalize sufficiently, reducing the risk of over-fitting.

2.2 Distributed Machine Learning

As the size of training data can significantly affect prediction accuracy, it has become common practice to train models with large datasets. To speedup these training tasks, they are often distributed across many servers.

In a distributed setting, a server iteratively refines a shared model by learning from a local data partition, and periodically synchronizes this model with the other servers. More specifically, after each iteration, it calculates a refinement ΔW_i to the model W. To make sure that all servers eventually converge to the same model, they can synchronize every iteration, every n iterations, or completely asynchronously. When machine learning algorithms are implemented on traditional distributed platforms such as Hadoop [38] or Spark [39], servers have to synchronize every iteration. This requires placing a barrier at the end of a iteration, incurring increasing overhead as the system scales.

Parameter Server [24] is another approach to implement synchronization in distributed machine learning. It outperforms the aforementioned platforms thanks to domain-specific engineering and algorithmic optimizations. In this approach, a set of servers act as *parameter servers* that store the model W. The other servers process the training data and act as *workers*. After ΔW_i are calculated, workers do not communicate with each other directly, but push ΔW_i to the parameter servers and then pull a new W to be used in the next iteration. By tuning push/pull frequencies, programmers can balance the training efficiency and system performance. For example, in the *Stale Synchronous Parallel (SSP)* model [12, 15], workers are allowed to cache W and use it in the next iteration while sending the ΔW_i of the previous iteration to servers, as long as the cached version is within a staleness threshold s. In this way, communication can be overlapped with computation. Nevertheless, as workers still have to periodically synchronize the model, the network can quickly become a bottleneck. In the next section we show how MLNET can alleviate this problem using a customized communication layer.

3 MLNET Design

We begin the section by describing the MLNET architecture and then we show how this makes it easy to implement our two techniques to optimize network performance, namely *traffic reduction* and *traffic prioritization*. While we believe that this is a contribution per se, we also see this a first step towards a deeper rethinking of the network design for distributed machine learning. We will elaborate on this point in Section 5.

3.1 Architecture

MLNET is a communication layer, running as a local process on workers and parameter servers, similar to Facebook `mcrouter` setup [30]. These local processes behave like proxy, intercepting all exchanges between workers and parameter servers. To push training results, a worker initiates a normal TCP connection to a parameter server that is actually emulated by a local MLNet process. Regardless of the actual numbers of parameter servers and workers, MLNet maintains a single parameter server abstraction to workers, and symmetrically a single worker abstraction to parameter servers. This design decouples workers and parameter servers and allows them to scale in and out independently. Further, it makes it easy to change the communication logic, e.g., to which server and when send the traffic to, without requiring modifications in the worker's or parameter server's code.

To achieve transparency w.r.t. both workers and parameter servers, MLNet inherits the standard communication APIs from the Parameter Server [24] where data is sent between nodes using *push* and *pull* operations:

- *weights = pull(modelId, staleness)*: Pull the *weights* of a model within a *staleness* gap.
- *push(modelId, gradients)*: Push the *gradients* of the weights of a model.
- *clock()*: Increment the clock of a worker process.

We give an example of using this interface. The training of a shared model consists of multiple iterations. At the end of an iteration, a worker *pushes* the newly calculated gradients of model weights to a parameter server and increases its local clock c_{worker} by calling `clock()`. The parameter server aggregates all pushes to update model weights and maintains a vector of the clocks of all workers. A model clock c_{model} is defined as the minimum worker clock in the vector. To start the next iteration, the worker then *pulls* the model weights that have to be within a staleness threshold s (see Section 2.2) by checking if $c_{model} \geq c_{worker} - s$.

3.2 Distributed Aggregation and Multicast

The push and pull phases are the primary sources of network traffic in distributed machine learning. In the push phase, workers send the gradients of model weights to parameter servers, while in the pull phase they receive the new weights generated after aggregating the gradients from all model replicas. These two operations can generate high congestion at servers and, if the network fabric is over-subscribed, in the core of the network too.

Adding more parameter servers would reduce the edge congestion by spreading the traffic (albeit at the expenses of increasing the overall server count) but it is of little help to reduce congestion in the core. We propose, instead, a different yet complementary approach that aims at reducing, rather than simply re-routing, network traffic by exploiting its domain-specific characteristics.

Aggregation and multicast tree. The aggregation functions used by parameter servers are typically *associative* and *commutative*, e.g., an `average` function is often used. This means that gradients can be aggregated incrementally and the final result is still correct. We leverage this property to reduce traffic during the push phase. For each parameter server, MLNet uses it as the root and builds a spanning tree connecting all workers. The workers in the leaves send gradients to their parent nodes. The latter aggregate all received gradients and push the results upstream towards the root where the last step of aggregation is performed.

Assuming that a node has an in-degree d. For each parameter, this node receives d values and transmits only one value (i.e., the aggregated result), thus drastically reducing the network traffic at each hop. This not only helps alleviating the load on parameter servers (avoiding the so-called *in-cast* effect), but also reduces the load in the network core too, which is beneficial in case of over-subscribed networks. As shown in Section 4, depending on the network fabric characteristics and the system load, different values of d may be preferable.

A dual technique is also used during the pull phase. In this case, rather than aggregating values on path, we use the same tree to *multicast* the weights to all replicas. By using a multicast tree, only d values per server are transmitted, which reduces the load on the outbound link of a parameter server (we expect $d \ll W$, where W is the number of workers in the system).

Synchronous vs. asynchronous operations. MLNet use the staleness threshold s to determine how push/pull operations are performed. In a synchronous setting, i.e., $s = 0$, a worker needs to wait for all its child nodes on the spanning tree in each iteration. In this case, a MLNet process uses its position on the tree to figure out the number of pushes in an aggregation window as well as the pull responses to multicast.

When $s > 0$, a MLNet process needs to decide when to perform push/pull operations. If a pull request can be satisfied with the cached weights ($c_{model} \geq c_{worker} - s$), the process responds it immediately, without incurring extra upstream network traffic. If not, the request is forwarded upstream, until it is satisfied with a cached weight, or it waits on the parameter server. When receiving a push message, the process first updates its own model with this message. It then pushes this message upstream if and only if the gradients carried by this message are not within the staleness thresholds compared to its parent node. This means that the degree of asynchrony is controlled by MLNet and it does not have to be hard-coded in a training algorithm.

Fault tolerance. To detect failures, a MLNet process uses heartbeats to check the liveness of its parent node on the spanning tree. If a node fails, the downstream nodes are connected to the parent of this failed node. They then exchange model information to ensure that staleness bounds are not broken. If the root of the tree, i.e., a parameter server, fails, its children wait for it to be replaced and then re-initiates a connection. If a failed node is back, it asks its neighbors to restore its local model and re-enters the tree. In case of multiple concurrent failures, the above mechanism may incur high overhead. MLNET then tears down the tree entirely and reverts to the traditional setup where workers directly communicate with parameter servers.

3.3 Network Prioritization

The second technique used by MLNET is network prioritization. Mainstream congestion control protocols such as TCP strive to provide *per-flow fairness*. This can be particularly detrimental in a multi-tenant environment where different models are trained in parallel because contention on network resources will delay the training time of *all* models sharing that bottleneck.

To address this issue, we implemented a network prioritization mechanism in MLNET to arbitrate access to network resources. We do not constrain how priorities are defined. For example, a model with relatively smaller communication cost could be given a high priority in order to complete earlier, leading to a shorter average model training time. Hereafter, we only assume that the MLNET process has a way to extract the priority from a flow (e.g., this can be encoded in the first transmitted byte).

A key challenge of implementing this feature is how to achieve this functionality without requiring any change in existing network infrastructure. Recent proposals, e.g., [3, 18, 36], require custom switch hardware and, hence, do not fulfill our requirements.

In contrast, we opted for a software only solution. If the network fabric provides full bisection bandwidth, e.g., through a fat-tree design [2,20], contention only occurs at the edge [22], i.e., at either the worker's or parameter server's uplink. When the destination machine, either a worker or a parameter server, receives a new TCP connection, it can inspect the relative priority (e.g., by looking at the first byte) in order to decide whether to reject the connection or to accept it by possibly dropping some of the existing ones if they have lower priority than the new one. As we show in the Section 4, this simple mechanism is effective in reducing the median training time without hurting the tail performance.

If the network is over-subscribed, the above mechanism is not sufficient as congestion can occur in the network core. If switches support enough priority queues, we can extend the above mechanism to take advantage of them, similar to recently proposed solutions [6, 28]. For the cases in which switch queues are not available, an alternative approach is to extend our previous work on bandwidth guarantees in a multi-tenant data center [8] to allocate bandwidth to flows according to their priority. This, however, would require knowledge of worker and parameter server locations and, hence, it might not be suitable for public cloud deployments. We are currently working on a decentralized solution that does not suffer from this limitation, possibly reusing some ideas from recent work on coflow scheduling [11, 18].

4 Preliminary Results

We use simulations to evaluate the performance of MLNET. While preliminary, the results indicate that our design can *i)* reduce the end-to-end training time (*makespan*), by performing distributed aggregation and multicast, and *ii)* shorten the median makespans of concurrent machine learning tasks by prioritizing flows.

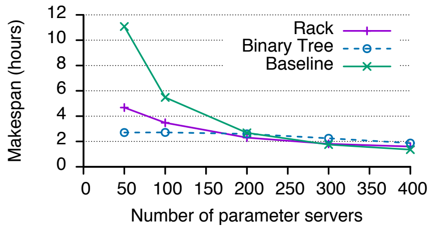
We use OmNET++ [44], a discrete event simulator, to model a mid-size cluster with a three-tier, multi-rooted network topology based on popular scalable data center architectures [2,20]. The cluster comprises 1,024 servers connected by 320 16-port switches via 10 Gbps links. We model the TCP max/min fairness model and we use the standard Equal Cost Multi Path (ECMP) routing protocol.

We adopt a synthetic workload modeled after a recently published machine learning algorithm for sparse logic regression [24]. The model has 65 billion parameters and is trained with a 141 TB dataset. The dataset is evenly partitioned and consumed by 800 workers. We vary the number of parameter servers between 50 and 400. The parameter space is equally divided across the parameter servers. Both workers and parameter servers are randomly deployed on the 1,024 servers.

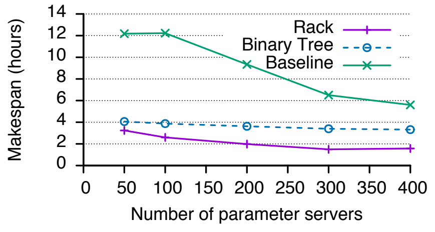
Based on our experience in training models with Minerva [35], we assume that workers process training data at a rate, uniformly randomly chosen between 100 MB/s and 200 MB/s. They use synchronized push and pull, and communicate with the parameter servers every 30 s.

Distributed Aggregation and Multicast. We begin our analysis by focusing on the benefits of using our aggregation and multicast mechanism. To show the impact of the node in-degree d, we compare the performance of today's approach (BASELINE) against two strategies that use a different value of d. The first one, RACK, uses a single aggregator per rack, i.e., $d = 7$, as in our experiments we assumed eight servers per rack. The other one, BINARY, instead, uses binary trees, i.e., $d = 2$.

Figure 1a shows the makespan of these configurations against the number of parameter servers together with 800 workers in a *non-oversubscribed* network. When there are 50 parameter servers, BINARY and RACK outperform BASELINE, taking only 22% and 42% of the time used by BASELINE. This shows the benefits of reducing the inbound and outbound load at the param-



(a) No over-subscription.



(b) 1:4 over-subscription.

Figure 1: Training makespan against the number of parameter servers.

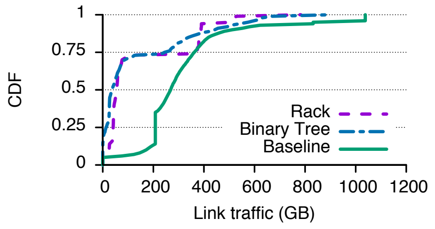


Figure 2: CDF of link traffic for different configurations.

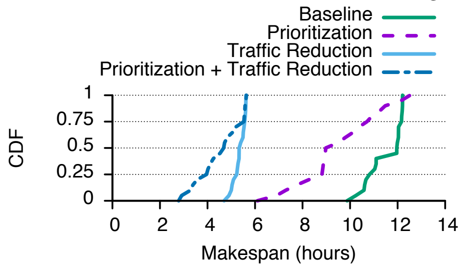


Figure 3: CDF of model makespans for different configurations.

eter servers. However, with a large number of parameter servers, the load is already sufficiently spread and our solutions become less effective compared to BASELINE. Finally, when the number of parameter servers is very high (≥ 300), the cost of re-directing traffic through the tree dominates and the performance of both BINARY and RACK gets worse than BASELINE. This shows the trade-off between the number of parameter servers and traffic reduction. Having more parameter servers helps to spread the load across the fabric and alleviate network bottlenecks. However, this comes at the cost of using more machines and is only effective when the network is not oversubscribed.

We then repeated the same experiment assuming a more realistic network setup with an over-subscription ratio of 1:4. In this case, congestion occurs in the network core too and, hence, unlike in the non-oversubscribed case, just increasing the number of parameter servers does not help in removing the main bottlenecks. In contrast, by reducing the overall network traffic, MLNET is also able to reduce the congestion in the core. This explains why, as shown in Figure 1b, RACK and BINARY outperform BASELINE in *all* configurations. For example, with 50 parameter servers, RACK reduces the makespan compared to baseline by 73% (resp. 66% for BINARY) and with 400 parameter servers, the makespan is reduced by 71% (resp. 40% for BINARY). Interestingly, in this configuration, BINARY achieves a worse performance than RACK. The reason is that since the height of BINARY's trees is higher, the path length increases accordingly and, hence, BINARY consumes more bandwidth than RACK. While this is not an issue in a non-oversubscribed network in which the core bandwidth is abundant, this becomes problematic in this setup where the core bandwidth is scarce.

In Figure 2, we show the CDFs of traffic across network links. In this experiment, we consider a non-oversubscribed network with 800 workers and 200 parameter servers. As expected, both RACK and BINARY exhibit a lower traffic per link, which explains the overall improvement in performance. We observed a qualitatively similar result in the over-subscribed case too (omitted for space reasons).

Flow Prioritization. Next, we explore the impact of network prioritization. We consider a workload in which 20 different models are trained concurrently. To reduce the computation load of our simulations, we scale down each mode to use only 200 workers and 50 parameter servers, and reduce the input data size proportionally. We randomly assign priority to each model and we start all the models at the same time. In Figure 3, to understand the impact of using prioritization alone and combined with traffic reduction, we show the CDF of the makespan for these configurations. As a comparison, we also report the CDF achieved when using traffic reduction alone and when using our TCP-based baseline.

Just using network prioritization reduces the median make span by 25% while only increasing the tail by 2%. Combining traffic reduction and prioritization together further improves the performance by reducing the median by 60% (resp. 54% for the 95^{th}-percentile) compared to baseline, and reducing the median by 13.9% compared to using traffic reduction only.

5 Discussion and Research Directions

We conclude the paper by highlighting current limitations and future research directions.

Model parallelism. The current design of MLNET targets the *data parallelism* model, in which multiple model replicas are trained concurrently, each using a different subset of input data. A complementary form of parallelism is the so-called *model parallelism* [16] in which a large neural network is partitioned across multiple workers. This exhibits a different communication pattern than the data parallelism model being explored in this paper. While some of the techniques discussed here might also be beneficial in this context, our next step is to understand the peculiarity of these patterns and design tailored solutions. One possibility is to extend recently proposed solutions for graph processing [19,32]. However, despite some similarities, deep neural networks differ from ordinary graphs as they are more structured and constrained but their sizes can be 10-100x larger.

Adaptive communication. Existing frameworks, e.g., [12,15,24,25], trade-off training efficiency for network performance by tolerating some degree of asynchrony through the staleness threshold s. The value of s, however, is usually determined a priori and cannot be easily changed at runtime. In contrast, we are currently exploring the feasibility of dynamically tuning this value based on *i)* the network load and *ii)* the convergence rate of individual parameters / models. This is particularly important for long-running training models, executing on public cloud infrastructure, in which the network performance is highly variable [8]. For example, in case of a highly loaded network, higher values of s (i.e., higher asynchrony) can be selected while, when network utilization drops, more synchronous configurations can be used. A similar approach can also be adopted to identify the best value of the node in-degree d.

This direction is complementary to recent work investigating similar trade-offs in the context of computation resources [21,29] and memory bandwidth [37].

Network infrastructure. A key design goal in our design was to maintain compatibility with the existing network deployments. Next, however, we want to explore the trade-offs of customizing the hardware as well. A first step in this direction is to investigate the feasibility of offloading our distributed aggregation to the switch hardware, e.g., using recently available programmable switch platforms [17,40,42,43] or middleboxes [4,26,27,33]. In the longer term, we intend to assess the potential benefits of designing a machine learning rack-scale appliance from the ground up, including chip design, network fabric, and storage infrastructure, following the example of recently proposed rack-scale architectures for computation and storage [5,7,14,31,41].

References

[1] AHMED, A., ALY, M., GONZALEZ, J., NARAYANAMURTHY, S., AND SMOLA, A. J. Scalable inference in latent variable models. In *WSDM* (2012).

[2] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM* (2008).

[3] ALIZADEH, M., YANG, S., SHARIF, M., KATTI, S., MCKEOWN, N., PRABHAKAR, B., AND SHENKER, S. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM* (2013).

[4] ANDERSON, J. W., BRAUD, R., ET AL. xOMB: Extensible Open Middleboxes with Commodity Servers. In *ANCS* (2012).

[5] ASANOVIC, K. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *FAST* (2014). Keynote.

[6] BAI, W., CHEN, L., CHEN, K., HAN, D., TIAN, C., AND WANG, H. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *USENIX NSDI* (2015).

[7] BALAKRISHNAN, S., BLACK, R., DONNELLY, A., ENGLAND, P., GLASS, A., HARPER, D., LEGTCHENKO, S., OGUS, A., PETERSON, E., AND ROWSTRON, A. Pelican: A Building Block for Exascale Cold Data Storage. In *OSDI* (2014).

[8] BALLANI, H., COSTA, P., KARAGIANNIS, T., AND ROWSTRON, A. Towards Predictable Datacenter Networks. In *SIGCOMM* (2011).

[9] CANINI, K., CHANDRA, T., IE, E., MCFADDEN, J., GOLDMAN, K., GUNTER, M., HARMSEN, J., LEFEVRE, K., LEPIKHIN, D., LLINARES, T. L., MUKHERJEE, I., PEREIRA, F., REDSTONE, J., SHAKED, T., AND SINGER, Y. Sibyl: A system for large scale supervised machine learning, 2012. Machine Learning Summer School, Santa Cruz, CA.

[10] CHILIMBI, T., SUZUE, Y., APACIBLE, J., AND KALYANARAMAN, K. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *OSDI* (2014).

[11] CHOWDHURY, M., ZHONG, Y., AND STOICA, I. Efficient Coflow Scheduling with Varys. In *SIGCOMM* (2014).

[12] CIPAR, J., HO, Q., KIM, J. K., LEE, S., GANGER, G. R., GIBSON, G., KEETON, K., AND XING, E. Solving the Straggler Problem with Bounded Staleness. In *HotOS* (2013).

[13] COATES, A., HUVAL, B., WANG, T., WU, D., CATANZARO, B., AND ANDREW, N. Deep learning with COTS HPC systems. In *ICML-13* (2013).

[14] COSTA, P., BALLANI, H., RAZAVI, K., AND KASH, I. R2C2: A Network Stack for Rack-scale Computers. In *SIGCOMM* (2015).

[15] CUI, H., CIPAR, J., HO, Q., KIM, J. K., LEE, S., KUMAR, A., WEI, J., DAI, W., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *ATC* (2014).

[16] DEAN, J., CORRADO, G. S., MONGA, R., CHEN, K., DEVIN, M., LE, Q. V., MAO, M. Z., RANZATO, M., SENIOR, A., TUCKER, P., YANG, K., AND NG, A. Y. Large Scale Distributed Deep Networks. In *NIPS* (2012).

[17] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., IANNACCONE, G., KNIES, A., MANESH, M., AND RATNASAMY, S. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *SOSP* (2009).

[18] DOGAR, F. R., KARAGIANNIS, T., BALLANI, H., AND ROWSTRON, A. Decentralized Task-aware Scheduling for Data Center Networks. In *SIGCOMM* (2014).

[19] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012).

[20] Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM* (2009).

[21] Huang, B., Boehm, M., Tian, Y., Reinwald, B., Tatikonda, S., and Reiss, F. R. Resource Elasticity for Large-Scale Machine Learning. In *SIGMOD* (2015).

[22] Jeyakumar, V., Alizadeh, M., Mazières, D., Prabhakar, B., Kim, C., and Greenberg, A. EyeQ: Practical Network Performance Isolation at the Edge. In *NSDI* (2013).

[23] Li, H., Kadav, A., Kruus, E., and Ungureanu, C. MALT: Distributed Data-parallelism for Existing ML Applications. In *EuroSys* (2015).

[24] 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 *OSDI 14* (2014).

[25] Li, M., Andersen, D. G., Smola, A., and Yu, K. Communication Efficient Distributed Machine Learning with the Parameter Server. In *NIPS* (2014).

[26] Mai, L., Rupprecht, L., Alim, A., Costa, P., Migliavacca, M., Pietzuch, P., and Wolf, A. L. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *CoNEXT* (2014).

[27] Martins, J., Ahmed, M., et al. ClickOS and the Art of Network Function Virtualization. In *NSDI* (2014).

[28] Munir, A., Baig, G., Irteza, S. M., Qazi, I. A., Liu, A. X., and Dogar, F. R. Friends, Not Foes: Synthesizing Existing Transport Strategies for Data Center Networks. In *SIGCOMM* (2014).

[29] Narayanamurthy, S., Weimer, M., Mahajan, D., Condie, T., Sellamanickam, S., and Keerthi, S. S. Towards Resource-Elastic Machine Learning. In *NIPS 2013 BigLearn Workshop* (2013).

[30] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling Memcache at Facebook. In *NSDI 13* (2013).

[31] Putnam, A., Caulfield, A., Chung, E., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P. Y., and Burger, D. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. In *ISCA* (2014).

[32] Salihoglu, S., and Widom, J. Gps: A graph processing system. In *SSDBM* (2013).

[33] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M. K., and Shi, G. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI* (2012).

[34] Smola, A. J., and Narayanamurthy, S. An Architecture for Parallel Topic Models. In *VLDB* (2010).

[35] Wang, M., Xiao, T., Li, J., Zhang, J., Hong, C., and Zhang, Z. Minerva: A scalable and highly efficient training platform for deep learning. In *NIPS Workshop, Distributed Machine Learning and Matrix Computations* (2014).

[36] Wilson, C., Ballani, H., Karagiannis, T., and Rowtron, A. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM* (2011).

[37] Zhang, C., and Re, C. DimmWitted: A Study of Main-Memory Statistical Analytics. In *VLDB* (2014).

[38] Apache Hadoop. `https://hadoop.apache.org`.

[39] Apache Spark. `https://spark.apache.org/`.

[40] Arista EOS+ Platform for Network Programmability. `http://www.arista.com/en/solutions/eos-platform`.

[41] HP Moonshot System. `http://bit.ly/1mZD4yJ`.

[42] NetFPGA. `http://netfpga.org/`.

[43] Netronome FlowNICs. `http://netronome.com/product/flownics/`.

[44] OmNET++ Discrete Event Simulator. `https://omnetpp.org`.