OFFLOAD OF MACHINE LEARNING ALGORITHMS TO PROGRAMMABLE NETWORKS

UNIVERSIDAD POLITÉCNICA DE MADRID

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

ADRIAN GARCIA ESPINOSA

MASTER THESIS

JULY 2019

Advisor: Arsany Guirguis
Lausanne, EPFL

Contents

Contents **iii**

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 2

1.3 State of the art and Problem Statement 2

2 Design **5**

2.1 General considerations 5

2.2 Roles 6

2.3 Operation 6

3 Implementation **9**

3.1 Limitations 9

3.2 Topologies 11

3.2.1 Parameter Server 11

3.2.2 In-Network Aggregation 12

3.3 Distributed Learning Protocol (DLP) 12

3.3.1 Header Design 13

List of Figures **i**

Bibliography **iii**

1. Introduction

1.1 Motivation

There is no doubt that Machine Learning (ML) today is changing the world enabling new ways of analysing, training and processing data in a way never explored before. For the last few years the remarkable success of ML has lead to important improvements and discoveries, both in hardware and software.

One of the emerging trends is Distributed Machine Learning (DML), where new applications are being continuously explored [1, 2, 3]. DML allows training of ML models in parallel, allocating both process and storage in multiple nodes. Training time dramatically increases with the sophistication of the models and the size of the data sets, thus DML training has become almost a standard practice for large setups[4]. In this context, large-scale clusters are used with hundreds of nodes equipped with multiple GPUs or other hardware accelerators. Existing ML algorithms are designed for highly controlled environments (such as datacenters) where the data is distributed among machines and high-throughput networks are available. At this point, it is quite clear that network performance at this day has a substantial impact on overall training time.

Flexible networking hardware and data plane programming languages have produced networks that are deeply programmable. The functionality of networks can now be enriched without hardware modifications while having the capability of processing packets at "wire rates", even above Terabits per second. The latest developments in networking hardware allow taking one step further on SDN setups by providing programable data planes with certain computing capabilities in the switch's ASIC [5].

Programmable networks create the opportunity for in-network computation, i.e., offloading a set of operations from end hosts into network devices such as switches and smart NICs. In-network computation can offer substantial performance benefits and flexibility, as it has been shown in recent proposals with consensus protocols and in-network monitoring [6, 7, 8].

1.2 Objectives

This project aims to explore all the benefits of In-Network computation applied to DML and considers the extension of current proposals by adding Bizantine Fault Tolerance to the distributed learning process. For this purpose, the following objectives have been defined.

Analysis of state of the art proposals both for DML and Programable Networks.

Design the protocol and architecture needed to comply the requirements.

Implementation of the actual system and its different scenarios.

Evaluation of the implemented system and compare the different results.

1.3 State of the art and Problem Statement

Distributed Machine Learning can be addressed from different aspects; relaying on one of the standard proposals [9] the main methods of distributed computing are:

- **Local training:** The model and the dataset is stored on a single machine.
- **Distributed training:** When it is not possible to store the whole dataset or a model on a single machine, it becomes necessary to do it so across multiple machines. In this thesis, this is the method chosen to explored. In this setup, there are 2 parallelization approaches:
  - Model Parallelism: When the model is splitted across multiple machines.
  - Data Parallelism: When data is distributed across multiple machines. This can be used in the case of data being to large to be stored on a single machine or to achieve faster training by paralellizing the processing. Again, this is the approach that will be considered in this thesis.

Regarding In-Network computing, this thesis relays on the P4 environment [5] as the language to define, implement and deploy programable networks that allow certain computing inside the switches. P4 works in conjunction with Software Defined Networking (SDN) control protocols like OpenFlow [10], enriching the capabilities and flexibility towards the *Active Network* concept [11]. SDN allows control over networks by separating the control and the data (forwarding) planes, being one single control plane able to control multiple forwarding devices. In this context, the P4 language has 3 main goals:

- **Reconfigurability:** The controller should be able to redefine packet parsing and processing in the field.
- **Protocol Independence:** The switch should not be tied to specific packet formats. Instead, the controller should be able to specify a packet parser for extracting header fields with particular fields and types and a collection of typed match-action tables to process these headers.
- **Target Independence:** Just as a C programmer does not need to know the specifics of the underlying CPU, the controller programmer should not need to know the details of the underlying switch.

At this point, it is important to note that In-Network computation applied to DML is not a new thing, there have been several proposals since last year [6, 12]. These proposals state the importance of the network overhead during the learning process and how hardware and algorithm improvements cannot bypass the *network bottleneck*. This same year, implementations and more realistic proposals where published which addressed this challenge using P4 as programable hardware language:

- *Scaling Distributed Machine Learning with In-Network Aggregation*[13]
  This is a generic implementation of In-Network aggregation using P4, they address multiple challenges like data quantization, packet loss and aggregation itself. The actual implementation of this paper doesn't seem to be publicly available.
- *Accelerating Distributed Reinforcement Learning with In-Switch Computing*[14]
  This paper goes a little deeper into the implementation of the aggregation process and focuses on applying it to Distributed Reinforcement Learning.
- *P4BFT: Hardware-Accelerated Byzantine-Resilient Network Control Plane*[15]
  This paper focuses on developing a BFT distributed learning environment by implementing a Bizantine Resilient control plane over P4 switches. Unlike the other proposals, this one does not address aggregation algorithms on switches.

These proposals have successfully supported that In-Network Aggregation is a topic currently explored and with great expectations among the community. Despite this thesis covers some of the problems stated in these papers, it is important to note that it was started before their publication. With no doubt, these papers have been inspiring and have provided valuable resources for the realization of this thesis.

The key points that this thesis addresses are:

- **Implementation and Evaluation of In-Network Aggregation:** Since, there was no actual implementation found, basic DML setups have been implemented and evaluated.
- **BFT in-network aggregation:** Up until now, it has never been approached yet in any proposal found. The 3rd proposal mentioned above [15] has an orthogonal approach focusing on the control plane, whereas this thesis uses BFT Aggregation but in the data plane.

2. Design

The previous chapter contains an introduction to the technologies used along the thesis, as well as a brief analysis on current proposals. Having already stated the problem statement, this chapter explains the design and considerations relevant for the later implementation.

2.1 General considerations

In order to efficiently design the system, the following considerations are worth noting:

- The method used for the distributed learning is Data Parallelism, where the dataset is distributed across the workers.
- The learning process is synchronous, thus the aggregation step is done with the weights of all the workers. This means that each worker waits for the aggregated response in order to update their local model, resulting in all workers learning eventually at the same time.
- It is assumed that no packet will be lost during the learing, meaning that the implementation doesn't handle packet loss algorithms (resend, CRC, etc). This assumption is not far from reality considering that modern datacenter setups consider realistic to have under 1% of packet loss [16].
- The system being designed will be deployed in an emulated environment, this is due to limited access to actual programable switches. In any case, thanks to P4's target independence, execution in a real environment should be exactly the same.
- The ML models valid for this setup have to use Gradient Descent as optimization algorithm, in this case will be used Stochastic Gradient Descent (SGD). It has been shown that Gradient Descent based learning is highly parallelizable and allows simple aggregation methods [9].

2.2 Roles

In all topologies (3.2), the aggregation process consists of 2 types of roles:

- **Worker(W)**: who has the computing power and calculates the gradients locally in every iteration before sending it to the aggregator. Each of the worker has a part of the dataset used during the learning process and updates its local model when receiving the aggregated gradients.

- **Aggregator(A)**: there is only one instance in each scenario. It takes care of the initial setup of the workers, the synchronous learning, and the parameter aggregation.

2.3 Operation

The learning process, starts by setting up the learning parameters in **A**, these parameters will be defined in detail in the protocol definition (3.3). Each of the workers $\mathbf{W_i}$ is registered in **A** and gets the learning parameters in order to start the operation. From now on, every iteration (defined in the learning parameters), each $\mathbf{W_i}$ will calculate its gradients, send them to **A** and wait for an answer. As soon as all gradients are aggregated, **A** sends back the new weights for $\mathbf{W_i}$ to update its local model and start again with the gradient computation. When all iterations are done, the trained model is evaluated.

This operation is described from a global perspective in Algorithm 1.

Algorithm 1: Distributed Learning Global Operation

Agents:
A: Aggregator
W_i: Worker instance

Operation:
P = {iterations, learing_rate, input_size, ...};
A.*setupLearningParameters(P)*;
A.*registerWorkers([*W_1*,* W_2*,* W_3*, ...])*;
A.*sendParametersToWorkers()*;

for *every iteration* **do**
  for *every* W_i **do**
    W_i.*computeLocalGradient()*;
    W_i.*sendWeightsToAgg()*;
    W_i.*waitForAggregation()*;
    A.*aggregateWeights(*W_i*)*;
    A.*sendAggregatedWeights(*W_i*)*;
    W_i.*updateLocalModel()*;
  end
end

W_i.*evaluateModel()*;

In order to implement the global operation presented above, both roles have been implemented using a state machine model, allowing a stateful learning process. The state machine definition needs to be consistent across all roles participating in the learning, this is why it is part from the protocol specification (3.3).

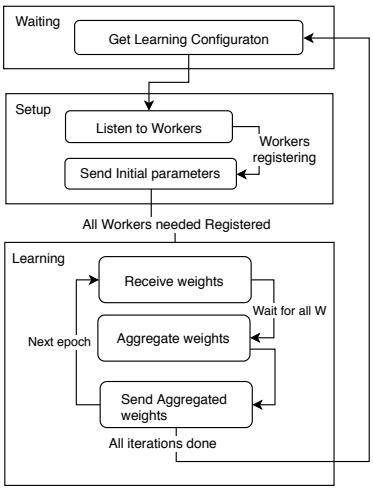
State Machine

The state of the nodes is stored in both roles in the *state* field. This field is also used for error handling in any operation between any **W** and **A**. Having this in mind, the following state types have been defined:

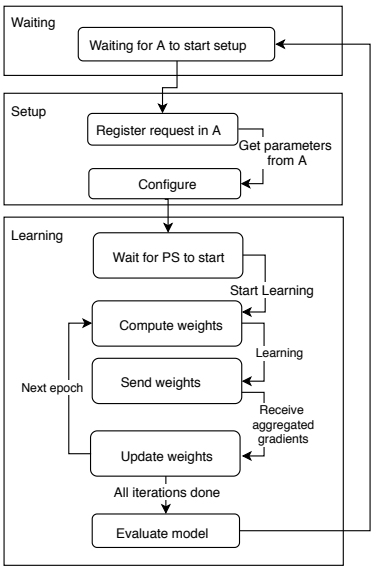
- STATE_WAITING: This is the default state where no learning process is ongoing.
- STATE_CONFIGURE: Used during the configuration of **A** with the initial parameters.
- STATE_SETUP: Used during the setup phase, where the parameters are exchanged and workers are registered to **A**.
- STATE_LEARNING: Used during the learning process, when exchanging the gradients between **A** and **W**s.
- STATE_WRONG_STEP: This is an error state used when the step of the gradients doesn't match the overall iteration step in the learning.
- STATE_ERROR: This is a generic error state.
- STATE_RESET: Used when the learning process will be reset.

The functioning of the state machine can be explained in the following diagrams. Some of the states don't appear on the flow diagrams in order to keep the representation simple and consider the most usual use case.

State Machine in Aggregator



State Machine in Workers



3. Implementation

This chapter will focus on the actual implementation of the designed system by considering the limitations and topologies that will be evaluated.

3.1 Limitations

Implementing the designed system addresses some limitations due to the technologies used and the use cases evaluated. Most of the limitations are imposed by the use of P4, being an extremely low level language with limited programming structures:

- **Limited Storage**: Gradient updates are usually large. In each iteration, each worker may supply hundreds of megabytes of gradient updates. This far exceeds the capacity of on-switch storage, which is limited to a few tens of MB and must be shared with other resources like forwarding tables and other core switching functions. This limitation is given by speed considerations since the strive to compute at wire-rate forces using the switch's on-die SRAM.

  Solution: this limitation is currently addressable by establishing some upper thresholds in the systems functioning. Thus, after considering current hardware capabilities the thresholds are:

  - The maximal number of workers used during the learning is 32, this valid as a first approach being considered also by other proposals [13].
  - The maximal size of gradient vectors generated by each worker that has been considered as a realistic first approach is 10.

- **Limited Computation**: Most common parameter aggregation algorithms are very simple, being almost standard the use of gradient averaging [17, 14, 13]. While a seemingly simple operation, it exceeds the capabilities of today's programmable switches. As they must maintain line rate processing, the number of operations they can perform on each packet is limited and the operations themselves can only be simple integer arithmetic/logic operations; neither floating point operations nor complex programming structures are possible like loops or recursive functions.

Solution: Partitioning of aggregation algorithms. This means, that the aggregation computation is split into both of the roles according to their limitations. In the implementation addressed by this thesis, 2 aggregation methods are used for evaluation:

- Arithmetic Mean: also known as Average, it is the most common algorithm for aggregation being defined as:

$$A = \frac{1}{n}\sum_{i=1}^{n} w_i = \frac{w_1 + w_2 + \cdots + w_n}{n}$$

- Marginal Median: it is a generalization of the one dimensional median. Median is considered a resistant statistic and has been proven to be Bizantine Resilient when applied to aggregation in gradient based learning [18].

  The median is the value separating the higher half from the lower half of a data sample, in this case, a sorted list of gradient values from each W_i.

It's important to note that due to the impossibility of looping in P4, for both methods, the use of loop unrolling was needed.

- **Parameter Handling**: Gradient aggregation, independently of the operation, is done over floating-point vectors. Currently, data types handled by P4 are very limited and the use of floating point numbers is ot feasible.

  Solution: The use of quantization for the parameters. The simplest approach is the use of a scaling factor during the learning. This scaling factor allows the workers to scale up the gradients and cast them to integers before sending them for aggregation. After getting the aggregation result,gradients are scaled back down and used for updating their local model. The estimation of this scale factor should be based on the maximal and minimal bounds of the gradient values.

- **Concurrency:** Modern switches support parallel processing of packets, it is the same case for programable switches. As mentioned in the design chapter 2, the operation is based in a state machine, which by it's nature needs to have atomic operations to it's state.

  Solution: The best approach to solve this was the use of P4's atomic tag in certain parts of the pipeline definition as well as structuring the code in a "concurrency aware" manner to avoid an unwanted behaviour.

3.2 Topologies

In order to be able to compare the performance and functioning of the problem statement, 2 topologies will be tested trying to use in both the closest environment setup possible. These topologies are:

3.2.1 Parameter Server

The Parameter Server setup has proven to be widely used in Distributed Machine Learning environments, not only by its performance but by its relative simplicity to other approaches [17].

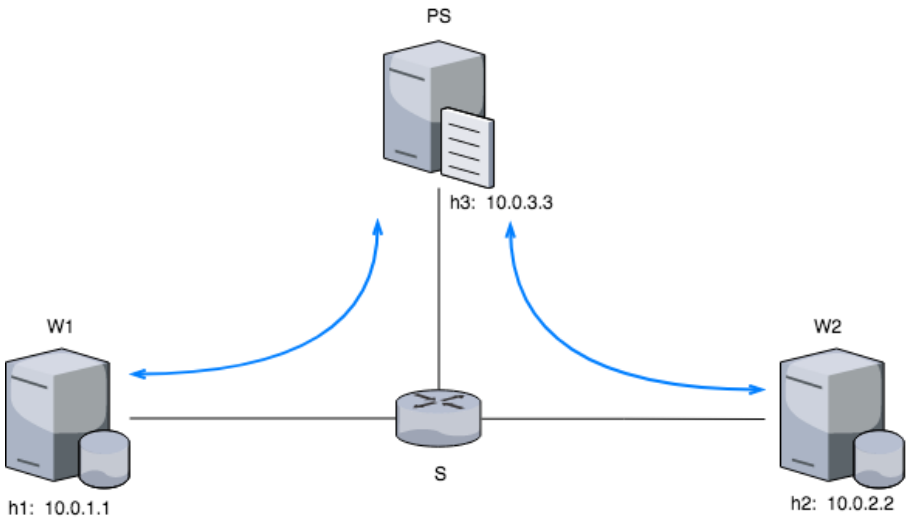


Figure 3.1. Parameter Server Topology

This topology is consists of the following elements:

- **W**: Workers run as separate processes and subscribe to the PS in order to do the aggregation synchronously with the other workers. Workers have their own data. During the learning, in each of the iterations, after computing their own gradients, they share the gradients with PS and wait until they get a response.

- **PS**: The parameter server handles the setup during the initial phase as well as the aggregation during the learning. In the learning process, for each iteration, it receives the weights from the workers, it makes sure to receive them synchronously and sends back the aggregation result. In this topology, the PS has the **A** role.

- **S**: In this setup, this is a normal switch.

3.2.2 In-Network Aggregation

This topology is proposed in order to illustrate the simplicity of the system and the lack of need of a centralised server unlike the previous topology.

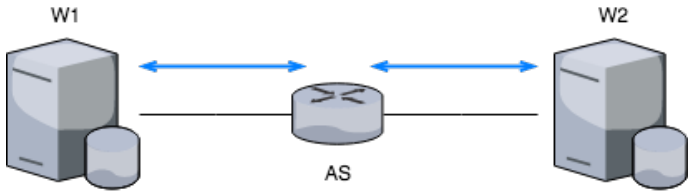


Figure 3.2. In-Network Aggregation Topology

This topology consists of the following elements:

- **W**: Workers behave exactly the same as in the previous topology but communicating with the AS instead of the PS. The learning algorithm will be the same but the communication with the AR may vary.
- **AS**: The aggregation switch has a basic routing purpose but with additional capabilities of aggregating the gradients sent from the workers. Thus, doing the work of the PS in the previous scenario. Using the AS imposes some limitations mentioned before (3.1) but offers aggregation at wire speed, therefore promising a better performance in the learning process. Before the setup phase, the control plane of the switch sets up the initial parameters to be sent to the workers. In this topology, the AS has the **A** role.

3.3 Distributed Learning Protocol (DLP)

In order to perform Distributed Machine Learning with a flexible and scalable implementation, it needs to integrate in both proposed topologies. A protocol has ben defined from scratch for this purpose, which has the following specifications:

- DLP sits on top of the transport layer (TCP/UDP), allowing the use of standard sockets in all the nodes involved on the learning. Being on top of the Ethernet and IP protocols also enables transparent use of standard routing and switching capabilities.
- Both the Workers and the Parameter Server will use raw sockets, striving to have the same environment in both topologies.
- The definition of this protocol allows the abstraction of any actual ML implementation, being the model and algorithms agnostic of the communication process.
- DLP has several limitations imposed by the In-Network Aggregation device. These limitations were be detailed in 3.1

The communication between the workers and the aggregators, both the AS and the PS, can be summarized with the following diagram:

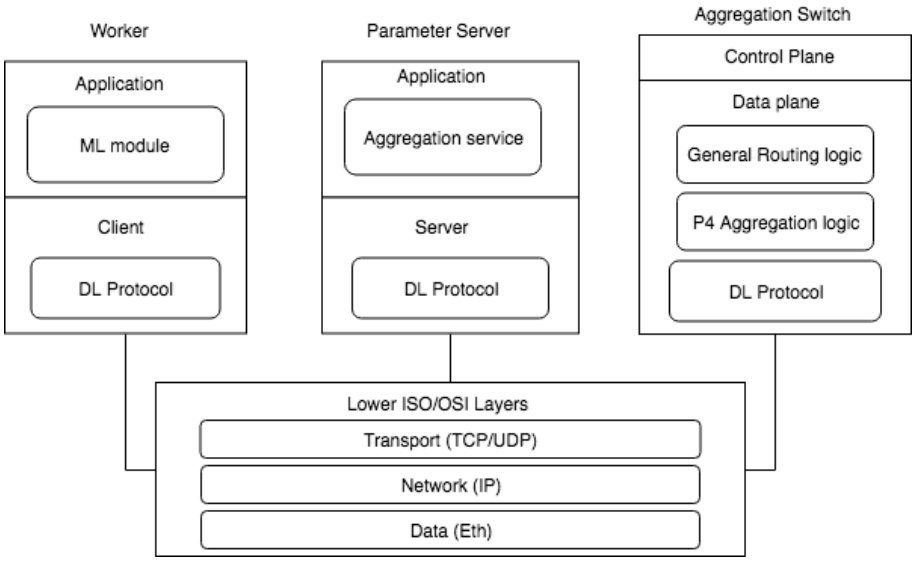


Figure 3.3. Communication using DLP

3.3.1 Header Design

As a part of the P4's programable data plane, the structure of the headers has to be defined. This structure needs to be consistent along workers and servers in oder to communicate in a transparent way.

Structure

- **bit** $<$ **8** $>$ state: This field contains the state of the aggregation process. It is modified by both the W and the A to exchange the state in every request. The value can be between 0-255.

- **bit** $<$ **8** $>$ node_count: It contains the nodes aggregated in each step. This field is filled by the A in the downstream to show progress to each W. The value can be between 255.

- **bit** $<$ **32** $>$ step: This field contains the current step to which the gradients belong to. Both the W and the A fill this field and use it to maintain synchronous learning. Max value can be between 0-4294967295.

- **bit** $<$ **32** $>$ param_0...N-1: These fields represent parameters that are sent between the W and the A. The value of these parameters depends on the state and the role of the sender. In the case of Figure 3.4, N is 6.

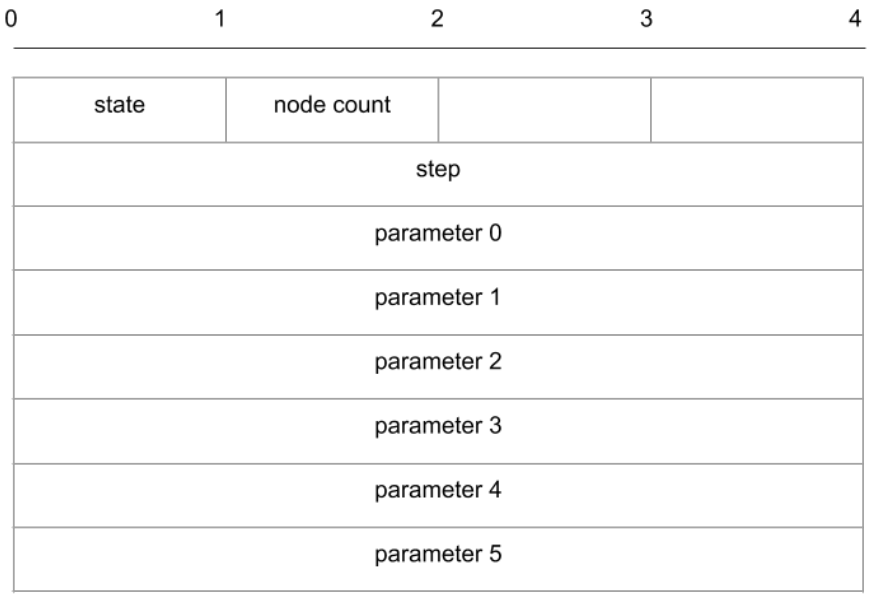


Figure 3.4. Header Structure

Content

The content of the fields depends on the state of the learning process as well as the role of the node that sends it:

- STATE_SETUP:
  W: send the initial state and the rest of the fields empty.
  A: Answers with the initial parameters, which are:
  - Iterations/steps
  - Learning rate/eta
  - Size of input data (number of entries)
  - Number of input features
  - Number of output classes
  - Scale factor
- STATE_LEARNING:
  W: sends the current state, the step, and the parameters are filled with the model's gradients.
  A: sends the current state (it could be an error state), the step, and the parameters are filled with the aggregated values from the other workers.
- Other states:
  Both **A:** and **W:** send their current state and decide what to do in case of errors.

List of Figures

3.1 Parameter Server Topology 11
3.2 In-Network Aggregation Topology 12
3.3 Communication using DLP 13
3.4 Header Structure 14

Bibliography

[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," dec 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[2] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A Distributed Framework for Emerging AI Applications," dec 2017. [Online]. Available: http://arxiv.org/abs/1712.05889

[3] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated Learning: Strategies for Improving Communication Efficiency," oct 2016. [Online]. Available: http://arxiv.org/abs/1610.05492

[4] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large Scale Distributed Deep Networks," Tech. Rep. [Online]. Available: https://www.cs.toronto.edu/{~}ranzato/publications/DistBeliefNIPS2012{_}withAppendix.pdf

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. Mckeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," Tech. Rep. [Online]. Available: https://www.sigcomm.org/sites/default/files/ccr/papers/2014/July/0000000-0000004.pdf

[6] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-Network Computation is a Dumb Idea Whose Time Has Come," pp. 150–156, 2017. [Online]. Available: https://doi.org/10.1145/3152434.3152461

[7] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, *NetChain: Scale-Free Sub-RTT Coordination*, 2018. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/jin

[8] C. Kim, P. Bhide, E. Doe, H. H. A, D. D. I, M. Hira, and B. D. V, "In-band Network Telemetry (INT)," no. September, pp. 1–28, 2016. [Online]. Available: https://p4.org/assets/INT-current-spec.pdf

[9] V. Hegde and S. Usmani, "Parallel and Distributed Deep Learning," Tech. Rep. [Online]. Available: https://stanford.edu/{~}rezab/classes/cme323/S16/projects{_}reports/hedge{_}usmani.pdf

[10] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 69, mar 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1355734.1355746

[11] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 2, pp. 5–17, apr 1996. [Online]. Available: http://portal.acm.org/citation.cfm?doid=231699.231701

[12] D. Sanvito, P. Di Milano, G. Siracusano, and R. Bifulco, "Can the Network be the AI Accelerator?" 2018. [Online]. Available: https://doi.org/10.1145/3229591.3229594

[13] A. Sapio, M. Canini, K. Chen-Yu Ho, J. Nelson Microsoft Panos Kalnis KAUST Changhoon Kim, A. Krishnamurthy, M. Moshref, and D. R. K Ports Microsoft Peter Richtárik KAUST, "Scaling Distributed Machine Learning with In-Network Aggregation," Tech. Rep. [Online]. Available: https://repository.kaust.edu.sa/bitstream/handle/10754/631179/main.pdf?sequence=1

[14] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating Distributed Reinforcement Learning with In-Switch Computing." [Online]. Available: https://doi.org/10.1145/3307650.3322259

[15] E. Sakic, N. Deric, E. Goshi, and W. Kellerer, "P4BFT: Hardware-Accelerated Byzantine-Resilient Network Control Plane," may 2019. [Online]. Available: http://arxiv.org/abs/1905.04064

[16] R. Bhardwaj, K. Chintalapudi, and R. Ramjee, "007: Democratically Finding the Cause of Packet Drops," 2017. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/arzanihttps://www.usenix.org/conference/nsdi17/technical-sessions/presentation/bhardwaj

[17] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. Schwing, H. Esmaeilzadeh, and N. S. Kim, "A network-centric hardware/algorithm co-design to accelerate distributed training of deep neural networks," in *Proc. Annu. Int. Symp. Microarchitecture, MICRO*, vol. 2018-Octob, 2018, pp. 175–188. [Online]. Available: https://www.cc.gatech.edu/{~}hadi/doc/paper/2018-micro-inceptionn.pdf

[18] C. Xie, O. Koyejo, and I. Gupta, "Generalized Byzantine-tolerant SGD," Tech. Rep. [Online]. Available: https://arxiv.org/pdf/1802.10116.pdf