Theoretical Computer Science 1003 (2024) 114611

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Learned load balancing ☆

Brian Chang [a,*], Kausik Subramanian [b], Loris D'Antoni [b], Aditya Akella [a]

[a] University of Texas at Austin, United States of America
[b] University of Wisconsin-Madison, United States of America

ARTICLE INFO

Keywords:
Load balancing
Data centers
Networking
Optimization
Supervised learning
Deep learning
Traffic engineering

ABSTRACT

Effectively balancing traffic in datacenter networks is a crucial operational goal. Most existing load balancing approaches are handcrafted to the structure of the network and/or network workloads. Thus, new load balancing strategies are required if the underlying network conditions change, e.g., due to hard or grey failures, network topology evolution, or workload shifts. While we can theoretically derive the optimal load balancing strategy by solving an optimization problem given certain traffic and topology conditions, these problems take too much time to solve and makes the derived solution stale to deploy. In this paper, we describe a load balancing scheme Learned Load Balancing (LLB), which is a general approach to finding an optimal load balancing strategy for a given network topology and workload, and is fast enough in practice to deploy the inferred strategies. LLB uses deep supervised learning techniques to learn how to handle different traffic patterns and topology changes, and adapts to any failures in the underlying network. LLB leverages emerging trends in network telemetry, programmable switching, and "smart" NICs. Our experiments show that LLB performs well under failures and can be expanded to more complex, multi-layered network topologies. We also prototype neural network inference on smartNICs to demonstrate the workability of LLB.

ACM Reference Format:
Brian Chang, Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2023. Learned Load Balancing. In *24th International Conference on Distributed Computing and Networking (ICDCN 2023), January 4-7, 2023, Kharagpur, India*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3571306.3571403

1. Introduction

Network load balancing has received a lot of attention in the datacenter context [3,4,14,22,25]. Load balancing is central to effectively utilizing the increased path diversity found in modern datacenter fabrics. An efficient fabric is key to different applications' performance, including both latency- and throughput-sensitive ones, and for effectively supporting emerging use cases such as serverless computing, low-latency inference, and large-scale machine learning.

☆ This article belongs to Section A: Algorithms, automata, complexity and games, Edited by Paul Spirakis.

* Corresponding author.
E-mail addresses: bchang@cs.utexas.edu (B. Chang), sskausik08@cs.wisc.edu (K. Subramanian), loris@cs.wisc.edu (L. D'Antoni), akella@cs.utexas.edu (A. Akella).

https://doi.org/10.1016/j.tcs.2024.114611
Received 31 August 2023; Received in revised form 27 February 2024; Accepted 28 April 2024

Available online 4 May 2024

Most state-of-the-art datacenter load balancing approaches thus focus on finding a "handcrafted" load balancing strategy that caters to a specific datacenter setting. For example, they may target a specific symmetric topology [4,14] or slowly changing network demands [3,23]. Many of these approaches probe local performance metrics, such as queue lengths at a given switch [14] or congestion on a path [4,22,25], to make locally-optimal load balancing decisions. While these approaches may work well for a specific network setting, they cannot provide any guarantees for other unforeseen settings.

In particular, due to failures, link capacity changes [38], and shifts in application mix and workload patterns, modern datacenters operate in an uncertain and constantly evolving environment. Today, when the network setting changes, operators are forced to live with suboptimal performance and severe application impact of handcrafted strategies. Alternately, network designers may try to craft a new niche load balancing strategy in an attempt to best serve the "new normal", often requiring or leveraging new hardware support (notable examples over time include [3,4,22]); this approach cannot keep up with datacenters' rapid pace of evolution.

Furthermore, ideally the optimal load balancing strategy can be computed by formulating the strategy into optimization problems such as the multi-commodity flow (MCF) problem, solving it, and installing it on datacenter switches. While these optimization problems allow us to discover the best strategy given certain workloads and topologies, solving these problems is typically much slower compared to the pace traffic pattern changes in datacenters: by the time an optimal solution is derived, the traffic pattern would have already changed, making the solution stale and unusable, which we later show in Section 2 with a simple analysis on real datacenter traces.

The limitations of existing approaches raise natural questions: *Can we automate the generation of load balancing strategies to keep up with the constantly evolving nature of datacenters?*

Instead of shooting for a "one-size-fits-all" strategy, we advocate using machine learning to automatically learn specific strategies that are *instance-optimal*: at every instant, the learned strategy provides close to optimal performance for the current traffic pattern and topology. Our learned load balancing approach, LLB, builds on recent successes of similar learned strategies in improving various aspects of networking [5,9,30,32,43,44] and systems [27,33] that utilize recent advances in machine learning. Machine learning has been pervasive in tackling computational problems across various fields, and in recent times, has found its way to solving networking problems [5,9,30,32,33,43]. At a high level, machine learning leverages real-world data to learn high quality solutions for complex problems.

First, modern datacenter networks have rolled out sophisticated telemetry frameworks [2,17,34,36] that offer detailed instantaneous views into traffic patterns, enabling *data-driven* load-balancing; the instantaneous view can enable the learning-based framework to select the right strategy that fits the current workload and topology. Second, programmable smartNICs are increasingly making their way into datacenters and provide the needed computation to run sophisticated models that support learned load balancing; in particular, advances in smartNIC architectures are pushing custom accelerators onto the NIC [39,41] offering the promise of even faster learned model inference. Third, recent advances in programmable switching enable both global (network-wide) enforcement of learned load balancing decisions (e.g., the load balancing weights to use at different switches along network paths) as well as rapid detection of events (e.g., link failures or capacity changes) that influence load balancing decisions. Fourth, by leveraging machine learning techniques to 'learn' a close-to-optimal solutions, running inference now has predictable latency for each observed traffic instance, which originally would have taken too long for a solver to compute. Given rich datasets on fine-grained traces within production clusters, we can derive traffic distributions of across spacial and temporal axes. By training on these datasets, machine learning model to make near-optimal decisions under particular traffic distributions. We would only have to re-train the network if the distribution shifts drastically, which occurs much less often compared to traffic demand changes within a distribution.

In this paper, we discuss modern factors that are driving the strong need for a learned load balancing approach, why a learned load balancing approach is plausible today, and the challenges to realizing its promise (§2). We present the architecture of LLB [8], our learned load balancing framework (§3). We discuss the algorithms we leverage, and the design choices we make to overcome the challenges to support both training learned models and deploying them in modern data centers using emerging smartNIC acceleration (§4). Finally, we present a comparison of our approach to the state-of-the-art (§5).

Our main contribution lies in arguing that a confluence of technologies has created a ripe setting with the right building blocks to deploy learned load balancing today (§2), and providing a learning-based solution based on these blocks. We make a case for using deep learning for a learned load balancing framework which can be used to generate load balancing strategies that adapt to different traffic failure, and topology scenarios. A second contribution is to show an approach to train the learned load balancing framework, and show how model inference can be deployed in the network using programmable/smart network interface cards.

2. Background and motivation

Despite years of effort, we still do not have load balancing techniques that can cope with the rapidly changing needs of datacenters. We make a case for *learned* network load balancing: a general framework to devise adaptive instance-optimal load balancing strategies for different kinds of topologies and traffic demands. We first outline how and why the datacenter environment changes rapidly, engendering the need for learned load balancing. We then discuss shortcomings of existing approaches. Finally, we outline technological advances that make the time ripe for learned load balancing and outline the challenges that must be overcome in realizing it.

Applications workloads are changing, leading to a constant evolution in the nature of traffic in the datacenter. Moreover, recent technological advances are enabling changes in topology at much quicker timescales than before (minutes/hours instead of months). State-of-the-art network load balancing frameworks are based on strategies that are customized for specific topologies and traffic characteristics, and are not well suited to adapt to evolving topology and traffic in the datacenter.

Table 1
Impact of delay on optimality.

Delay(s)	Max Opt. Gap	Average Opt. Gap
0	1	1
1	10.29	1.58
2	14.05	1.65
5	12.98	1.74
10	17.98	2.01

2.1. Why learned load balancing?

Evolution in Traffic Patterns. Datacenter network traffic mix changes based on the active application mix and the applications' intrinsic send/receive patterns (e.g., ON/OFF, diurnal, random burstiness, connection patterns such as out/incast, etc.). We analyzed the fbflow packet traces [36] and found that pod-to-pod traffic has largely uniform rates over time (determined by the applications in the pod), with the occasional microbursts interspersed over time. Luckily, with rapid advances in network telemetry due to improved in-network [2] and end-host [34] capabilities, operators can now accurately measure current traffic patterns at fine granularities (both in terms of time and at the level of the traffic grouping, e.g., individual flows). We argue that this capability should be leveraged to design a network load balancing framework that makes *high quality global load balancing decisions* commensurate with the *high quality of data*.

Consider Conga [4], a state-of-art distributed load-balancing scheme. Conga indirectly estimates the current load on the network by measuring congestion metrics on various paths. It uses these measurements to determine how to re-balance load to equalize congestion among multiple source-destination paths with the implicit hope that this local readjustment results in global load balance. As we show later (§5), comparing Conga with a "optimal" oracle-based load balancing solution that uses instantaneous network-wide input traffic patterns and demands to solve a multi-commodity flow (MCF) problem [12] to determine load balancing weights, we find that Conga has 50-90% higher maximum link utilization than optimal. This shows that instead of an one-size-fits-all, indirect approach, a direct *data-driven* strategy that *explicitly* optimizes a *network-wide* objective for the *current* observed workload can be highly beneficial. Using machine learning, we can learn the load balancing strategy to use as a function of the current workload.

Delay in Optimal Strategy Computation. Theoretically, it is possible to find an optimal load-balancing decision given the observed network and traffic by repeatedly collecting traffic demands, then transform the demands into an optimization problem such as the MCF problem. However, in practice there are several obstacles that need to be overcome: First, on larger networks, solving the MCF problem for each observed traffic demand constantly could potentially become a bottleneck. Second, after solving the MCF problem, the network would still need to update it's configurations to reflect the derived optimal solution constantly. This process might take up to several seconds to complete, and by the time all the solving/updating are completed, the derived solution would have gone stale and no longer optimal.

To demonstrate this, we use the traces from fbflow [36], which consists traces from Facebook's production clusters. Traces contain information such as (source, destination) pairs, flow size, and timestamps at the granularity of seconds. We aggregated flows across each (source, destination) pair, and used the aggregate traffic demand as input to a MCF problem solver to derive an optimal solution S. Delay is defined as the amount of time it takes before we can observe input at time t. For instance, if there is a 3 second delay, we can only observe traffic on $t = 10$ when $t = 13$. Optimality gap is calculated as S/S_o, where S_o is the optimal solution. Optimality gap captures how *close* the current solution is to the optimal.

Table 1 shows how delay in traffic observation impacts optimality. We can observe that even a few seconds of delay has a big impact on the derived solution, with some instances having 17+ times worse performance. We also calculated how long it takes to solve each MCF instance of the aforementioned traffic demands on a machine with an 2.2 GHz 6-core Intel Core i7 processor. On average each instance takes around 75 ms to solve.

Given the high variance of datacenter traffic, where flows can arrive every 1000 to 10000 us based on recent studies [36], delayed derivation of the solution (large optimality gap) and time taken to solve optimization instances (75 ms, 3-4 orders slower than flow inter-arrival times) makes repeatedly computing optimal solutions impractical to deploy. Our learning-based load balancer should be able to *derive near-optimal solutions with low latency*, which would overcome the performance overhead caused by delays.

Evolution and Fine-grained Changes in Topology. Datacenter topologies are constantly evolving due to a flood of compelling new network structures [16,37,40] many of which are being rolled out in practice. Although symmetric network topologies are still popular in the datacenter, such topologies are constantly subject to asymmetries: new switching technologies, e.g., optical networks [13] and Flyways [24], allow dynamically adding/removing links; recent advances [38] allow near-instantaneous changes to optical network link capacities; and, link failures are frequent in datacenters [15] and can cause topology asymmetry.

An ideal load balancing approach must be able to optimally support all possible types of topologies, including ones that are symmetric or asymmetric at the topological or the link-capacity level or both. In addition, the load balancing strategy must be able to optimally adapt to failures.

Finally, our network balancer must be able to handle the volatility in the network caused due to changes in the topology due to failures and changes in traffic demands. With the advent of standardised programmable switches like Barefoot Tofino and off-the-shelf programmable SmartNICs like Broadcom Stingray, we can potentially implement complex load balancing algorithms in hardware at line rates. Thus, argue that a load balancer should *be able to be implemented using off-the-shelf technologies to provide line rate processing throughput.*

Table 2
State-of-art load balancing strategies.

Type	Name	TD	NT	HW
Centralized	B4		✓	✓
	Hedera			✓
Distributed Local	ECMP/WCMP			✓
	DRILL	✓		
Distributed Global	Conga	✓		
	Hula	✓		
	Contra	✓		
	LLB	✓	✓	✓

2.2. Limitations of state-of-the-art

We classify the state-of-art load balancing strategies into three categories: (1) centralized controllers, which use topology and traffic demands to generate optimal network-wide paths, (2) distributed approaches, which are either stateless or rely only on local knowledge at a switch/end-host, e.g., local queue statistics, and (3) distributed approaches, which use network-wide statistics (e.g., congestion metrics on paths) to make load balancing decisions. Table 2 summarizes these different strategies.[1]

Today's deployed *centralized approaches*, e.g., Hedera [3] and Google's wide-area network load balancing controller B4 [23], take globally optimal decisions that can be used on different kinds of topologies. But existing designs and implementations render them incapable of handling traffic variations at millisecond-timescales (B4 can take ∼1 second to calculate new paths that optimize global metrics, and cannot react quickly to traffic variations).

Today's completely *distributed approaches* (ECMP [21] and WCMP [45]) are tailormade for specific network topologies and traffic demands – ECMP works best for symmetric topologies, while WCMP uses a central controller to compute the weighted load distribution for different switches. Other switch-local distributed approaches like DRILL [14] use local switch queue statistics to adapt to changing traffic demands. However, these distributed local load balancing approaches cannot deal with changing topologies and asymmetries as they do not have visibility into network-wide statistics. For instance, let's consider a switch has three outgoing links with WCMP weights 1:2:1. When the third link fails, WCMP does not adapt and still distributes load among the active links using 1:2 weights, which could lead to congestion on the second link due to increased demand.

Finally, today's *distributed approaches* with deeper visibility into network statistics (e.g., Conga [4], Hula [25] and Contra [22]) change how traffic is sent on paths by dynamically probing paths and measuring load on them. For instance, Hula [25] is a performance-aware load balancer that leverages enhanced data-plane capabilities to collect link utilizations. Hula's algorithm does not depend on the topology or traffic demands, it sends flowlets on the current least utilized hop (determined by probes). However, Hula probing mechanism requires certain structure to the topology (notion of upstream and downstream switches), and thus, Hula cannot be used for arbitrary topologies. In general, these approaches do not use current network-wide traffic as input and thus cannot perform global optimization. These approaches also often make assumptions about the topology and cannot be easily used for arbitrary topologies, or they break under topology asymmetries. Probing network path metrics requires support for custom hardware (as is the case with Conga) which may not be available.

In this paper, we seek to address the limitations of prior work and answer the following questions: With a wide variety of datacenter designs and workloads, is there a general approach to find an optimal load balancing strategy given a network topology and workload? How should the strategy change under different failure scenarios? Can we leverage new programmable network devices such as standardised P4 programmable data plane switches and off-the-shelf programmable SmartNICs to implement our load balancing algorithm in hardware to run at line rates?

2.3. Why learned load balancing now?

We envision a *learned network load balancing framework* which optimizes *network-wide objectives* while leveraging *evolving topology and traffic information*, and can *run on off-the-shelf hardware with low latency* for easy deployment. We see four emerging opportunities that make We discuss these and associated challenges next.

Opportunity 1: Advances in Telemetry. Recent advances in programmable switches and scalable monitoring infrastructures have made fine-grained traffic measurement feasible [34]. In addition to what applications are running and their fine-grained communication patterns and demands, it is possible to track in-network statistics such as current and historical queue lengths at each switch [2]. Operators therefore have a plethora of traffic data, which can be used as input to infer load balancing strategies.

Challenge 1: Data Collection and Overfitting. Like any system based on learning, our approach requires collecting representative training data and avoiding overfitting. Constant innovation in telemetry [2,17,18] is enabling ever-richer instantaneous views into network traffic, but we need a traffic-pattern-modeling scheme to craft suitable training data that allows our deep learning-based approach to effectively learn strategies that are optimal for traffic and topology patterns that are yet to be seen.

[1] We show if the load balancer is adaptive to traffic demands (TD), can work on different network topologies (NT), and can run on off-the-shelf hardware (HW).

Table 3
List of Variables of the MLU Optimization Problem.

List of Variables		
Input	$G(E,V)$	Topology Graph Representation with edges E and vertices V
Input	$\mathcal{D}(s,t)$	Traffic Demand between source s and destination t
Input	$\mathcal{P}(s,t)$	Set of paths between source s and destination t
Input	$\mathcal{C}_e$	Link capacity of edge e
Input	$\mathcal{F}$	Set of failed links
Intermediate	$\mathcal{U}_e$	Link Utilization of edge e
Output	$\mathcal{W}_p$	Path weight for path p

Opportunity 2: Deep Learning. Recent advances in Machine Learning (ML), in particular Deep Learning (DL), have opened the door to exciting ML-based approaches to improving various large scale systems. Examples include fast learned indices in distributed databases [27], big data scheduling [33], routing strategies [9,43] and adaptive video streaming algorithms [32]. Inspired by the success of these approaches, given rich traffic traces and metadata recent telemetry frameworks have allowed operators to collect, we aim for a similar ML-based framework that automatically learns the best instance-optimal network-wide load balancing strategy for the current traffic pattern and network topology. If an ML network can be trained to *adjust to network failures and traffic changes*, we could obtain a load balancing framework that is performance-aware, failure-tolerant, and is not overfitted to a certain traffic scenario.

Challenge 2: Learning Challenges. Since load balancing is distributed, each switch ideally must have an individual learned model. However, in the optimal case, each switch requires global information on the traffic, failures and topology in order to make an optimal decision. Thus, we formulate the load balancing problem as a *global* optimization problem–i.e., a single ML agent makes load balancing decisions for the entire network. Moreover, it is not practical to re-learn load-balancing every time the current traffic pattern shifts or topology changes due to link failures. Our deep learning-based approach must be able to infer the load balancing strategy when specific topological or traffic changes are seen.

Opportunity 3: SmartNICs. Load balancing decisions can be performed at different granularities: packet, flowlet or flow-level. Of these, per-packet load balancing can be most effective in distributing load across paths. While it can lead to re-ordering of packets, modern fabrics and transports are increasingly robust to reordering [39] and can be easily combined with learned per-packet load-balancing. However, this means that we may need to perform inference using a learned load balancing model at high line rates (100G and beyond). While modern switches with P4 capabilities can offer line-rate custom processing of packets, their limited computational models restrict the nature of learned models that can be executed on them. Here, a new opportunity is offered by programmable *network interface cards* (NICs), which are being increasingly deployed in datacenters today. Such smartNICs' embedded on-board cores and FPGAs can support complex neural network computations and contain dedicated cores, custom ASICs [1] or packet processing at end-hosts at high throughput and without incurring additional latency.

Challenge 3: Inference at Line Rate. While smartNICs are more computationally capable than programmable switches, model inference must be carefully engineered [31] to achieve line rate throughput and low packet latency. In particular, this may need co-design of the model/parameters (e.g., input, output, number of hidden layers) with the NIC hardware specifications to ensure sufficiently rich models can run with the needed performance.

Opportunity 4: Programmable Switches. Modern programmable switches provide the opportunity of encoding custom headers and parsers using high-level languages [6]. This can be powerful, when combined with smartNICs: the latter can compute the inferred weights to use at multiple on-path switches and encode them into packet header fields; each switch can then parse the headers to determine the weights to use locally for load balancing. Crucially, this decouples inference from load balancing, and avoids having to update switch rules when the model provides a new set of inferred load balancing weights.[2] Programmable switches also allow for enhanced in-band network telemetry [2] and rapid remote link failure detection [20].

Challenge 4: Interaction with Routing, and Telemetry Latency. While programmable parsers provide the nice advantage outlined above, careful design is necessary to account for the fact that network routing may change independently: without co-design with routing, the inferred weights may result in poor load balancing performance when routes change. Furthermore, it is important to gather network-wide statistics and provide them quickly to NICs for inference.

Instead of designing a custom load balancing strategy for every topology, we want an approach that is robust for different network designs, workloads, and failure scenarios, and is extremely agile, i.e., the strategy can adapt to changes in the network quickly. This is our primary motivation for using deep learning. The deep learning model will learn to make decisions similar to an optimization solver, where the model will produce *Weighted ECMP (WCMP)* weights for each leaf switch, and adjust these weights dynamically based on traffic demands and global network status. Along with the utilization of stateful programmable devices such as smartNICs, this enables the opportunity to deploy machine-learning models on end-hosts that can be used to perform guide network load balancing.

[2] In contrast, when WCMP [45] is used, for example, we may have to reprogram weights at each switch.

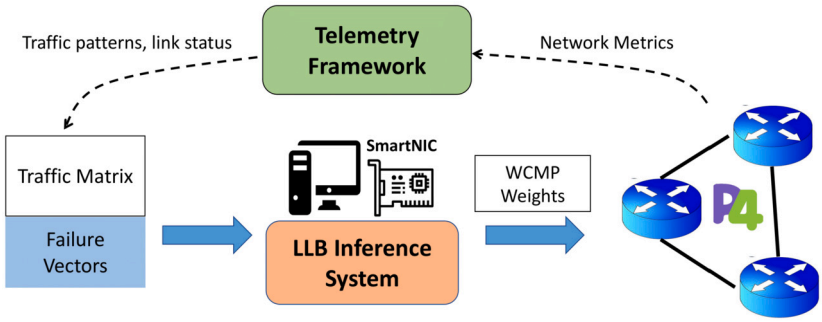

Fig. 1. Learned Load Balancing (LLB) Architecture.

3. Learned load balancing

The architecture of LLB is illustrated in Fig. 1. LLB collects traffic demand and link status data from the network's telemetry framework and runs inference on the trained deep learning model. The output of the DL model at each inference step is a set of WCMP weights that is then deployed onto the network. In the rest of this section, we first describe the optimization problem tackled by LLB to achieve load balancing (§3.1) and how LLB uses deep learning to solve such a problem (§3.2).

3.1. Minimizing maximum utilization

In this paper, we focus on load balancing strategies that aim at reducing link utilization. Given a set of *WCMP weights*, and a *traffic demand*, the utilization of a link is defined as the portion of aggregated traffic sent across a link.

Link utilization can be used as the recipe of a load balancer's objective. One of the most commonly used objectives is minimizing the *maximum link utilization* (MLU) across all links in the network.

LP Formulation. One aim of our formulation is to make sure it is generalizable enough to work on different topologies. We use a variation of the Multi-Commodity Flow (MCF) problem, where we incorporate the min-max fairness objective into the original MCF formulation. The formulation is shown below and details are provided in Table 3.

Objective: minimize $max(\mathcal{U})$
Subject to:
c1. $\sum_{p \in \mathcal{P}_{(s,t)}} \mathcal{W}_p = 1 \ \forall (s,t) \in G(E,V)$
c2. $\mathcal{U}_e = \sum \mathcal{D}_{(s,t)} * \mathcal{W}_p / C_e \ \forall e \in p \wedge p \in \mathcal{P}_{(s,t)}$
c3. $\mathcal{W}_p \geq 0 \ \forall p$
c4. $\mathcal{W}_p == 0$ iff $e \in \mathcal{F} \wedge e \in p$

Given a graph $G(E,V)$ that represents the network topology, traffic demand, a set of paths $\mathcal{P}$ for each (source, destination) pair, and topology information (failures and capacity), constraint **c1.** ensures weights of paths with the same (source, destination) add up to 1, **c2.** calculates each link's utilization, **c3.** ensures all weights are positive, and **c4.** makes sure no traffic is assigned to a failed link.

3.2. How LLB learns

In this section we describe how LLB captures the aforementioned LP formulation in a deep learning setting. We briefly describe how we use deep learning to approximate the behavior of the optimizer.

Deep Learning and Supervised Learning.

LLB uses Deep supervised learning (DL) to train an agent that makes load balancing decisions. Deep learning (DL) models have acquired huge success in various classification and regression tasks such as image classification, speech recognition and recommendation engines [29]. On the other hand, supervised learning is a machine learning method that trains a function that maps inputs to outputs based on a given dataset of (input, output) pairs. A simplified objective of supervised learning is as follows:

$$f(x) = \hat{y} \tag{1}$$

$$min(\mathcal{L}(y, \hat{y})) \tag{2}$$

Given input, output pairs (x, y), supervised learning methods try to find an approximation function f that aims to minimize a loss function $\mathcal{L}$ with respect to output y and prediction $\hat{y}$.

Given the nature of our problem's background described in Section 2, we can present the deep learning model with abundant (input, output) examples of the optimization formulation in Section 3.1, and let it learn to approximate the optimizer's behavior [7]. Deep learning models have the ability to capture complex linear and non-linear relationships between inputs and labels to the

network, which is a good fit for our problem formulation of capturing the MCF solver's behavior and serve as the predictor to the given (input, output examples).

We train LLB by drawing traffic from real traces along with different failure/capacity settings to generate a rich dataset, which the deep learning model then is trained on. Concretely, given the dataset of traffic demands and topological data, we want the deep learning model to predict the best WCMP weights under the input.

4. Training LLB

In this section we present LLB's formulation of the learned network load balancing problem as a deep learning problem. We describe how LLB trains a load balancing policy and how it uses the learned policy to modify the state of the network at runtime. To model the load balancing problem as a deep learning problem we first have to make a set of choices: Where should we draw the dataset from? What is the output of the deep learning model? How should we train the model? We also discuss the challenges and potential solutions for LLB to be deployed in the network.

4.1. LLB formulation choices

Modeling Traffic as a Distribution. The ultimate goal of a learning-based solution is to leverage historical data for training a solution that generalizes to unseen traffic and topology patterns. Besides modeling real traffic traces, a learning-based solution should also be able to predict for unseen data. In order to achieve this, we assume traffic matrices follow a certain *distribution*, which captures the characteristics of collected traffic traces.

We define the LLB's traffic distribution $\mathcal{D}$ as follows: $\mathcal{D}(C, \Delta)$. Data points generated using this distribution is termed as a *traffic matrix* that describes the traffic volume between a source and destination pair Traffic matrices drawn from this distribution are sampled from a high-dimensional ball with center matrix C and radius Δ, where C's dimension is the number of host pairs in the topology.

This methodology models **micro-changes** in traffic patterns by creating noisy versions of traffic matrices appearing in the given set of traffic patterns. We call this sampling space a Δ-ball of T.

Modeling failures. Besides $\mathcal{D}(C, \Delta)$, in order to model failures during dataset generation and training, we model link failures inside the network as a binary *failure vector* variable $\mathcal{F} = [l_1, l_2, ..., l_n]$, where $l_i = 1$ denotes a failure at link i, and $l_i = 0$ denotes that link i is working. This vector, along with the aforementioned traffic modeling yields $\mathcal{D}(C, \Delta, \mathcal{F})$, a distribution that produces various scenarios of topology and traffic changes. Traffic matrices and failure vectors generated with the methodology above is then passed as input to LLB's prediction network, which is encoded as a 3-layer fully-connected neural network.

Network-wide Objective. In this paper, we train LLB to minimize the utilization of the most congested link in the network. To minimize the maximum utilization, we define our optimization objective as $max(\mathcal{U})$ (where $\mathcal{U} = max_l \mathcal{U}_l$). This objective is used in the multi-commodity flow (MCF) solver to derive the best set of WCMP weights, where the weights are then learned by LLB's deep learning model. An advantage of a learning-based approach is that it is programmable, i.e., DL models can be easily trained on different objectives by generating datasets derived from different optimization objectives. For instance, one can train the model to minimize *average utilization* and therefore distribute traffic evenly across all links. We could also potentially train the agent to take multiple sub-objectives into account by using *composite objectives* that include network metrics such as queuing status (Q), flow completion time (F), and throughput (T) in the objective function. For instance, one can define an objective function $a/Q - bF + cT$, where a, b, and c are non-negative constants, which asks LLB to minimize queuing and flow completion time while maximizing throughput. This programmable aspect gives learning-based load balancing schemes a promising key advantage: the ability to customize to different policies while taking multiple sub-objectives into account.

Choosing the Training Model. One of the common architectures used in deep learning is the fully-connected neural network architecture. LLB uses fully connected neural networks for training. Two main reasons for this choice is that fully-connected networks are cheap to compute, and their simplicity to implement make them more straightforward to offload onto network devices such as smartNICs. Also, while the approximating nature of machine learning prevents our model from perfectly predicting optimal solutions for all data points, we will see that this simplification still yields good results in Section 5.

Representing complex topologies. Besides capturing the adaptive nature of the LLB problem statement, another design challenge is how we can generalize LLB to large and complex topologies. One issue that arises from the formulation is that in larger topologies, the number of possible paths between host pairs increases exponentially, resulting in huge and sparse (link weights are thin-spread) solution spaces in the dataset. This aspect makes training impractical because *1.* a much larger neural network is needed to train the agent and *2.* the ultimate goal is to run inference on programmable devices like smartNICs, and increasing the neural network size to fit a large topology would make inference on the NIC not fast enough for line-rate processing.

Therefore to tackle this challenge, we provide a solution with the use of ***tunnels*** in order to keep the neural network size fairly moderate and run inference at acceptable speed. Tunneling protocols in computer networks allow data between private networks to be sent across a public network, and are typically 'point to point' that connects a user to a remote resource/user at the other end of the tunnel. LLB adapts tunnels' *point to point* characteristics and **pre-generate n tunnels for each (source, destination) pair**, where n is a magic number that the developer determines.

The LP formulation of minimizing maximum utilization changes slightly when we abstract possible paths as separate tunnels. However, this abstraction actually simplifies the formulation into a much more general form. Assume that each tunnel

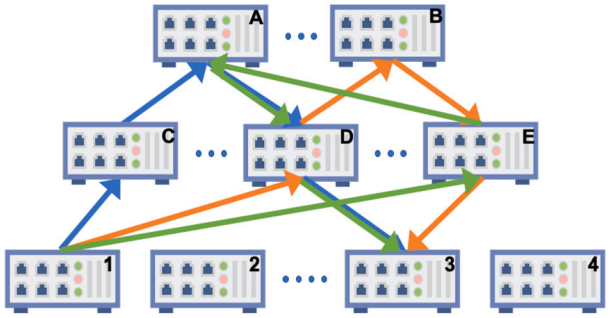

Fig. 2. Tunneling and Path Selection. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

Algorithm 1 LP Formulation for Tunneling Abstraction.

Notations:
$\mathcal{T}_{s,d}$: Traffic demand between source s and destination d
$\mathcal{W}_{i,j}$: WCMP weight of tunnel i, j
$\mathcal{U}_l$: Link utilization of link l
C_l: Link capacity of link l
$\mathcal{F}_{i,j}$: Failure vector ($\mathcal{F}_{i,j} = 1$ iff link i, j has failed)
$tunnels[s, d]$: Set of tunnels from source s to destination d
Algorithm:
$\mathcal{U}_l \leftarrow 0$ for every link l
for all pairs (s, d) **do**
  for all tunnels t in $tunnels[s, d]$ **do**
    if No links in tunnel t have failed **then**
      for all links l in tunnel t **do**
        $\mathcal{U}_l \mathrel{+}= \mathcal{T}_{s,d} * \mathcal{W}_{s,d}$
      end for
    end if
  end for
end for
Objective: $minimize\ max_l(\mathcal{U}_l)$
Subject to:
1. $\forall i. \sum_j \mathcal{W}_{i,j} = 1$
2. $\forall i, j. \mathcal{U}_{i,j} \leq C_{i,j}$
3. $\forall i, j. \mathcal{F}_{i,j} = 1 \Rightarrow \mathcal{U}_{i,j} = 0$

$t_i = \{l_1, l_2, ..., l_n\}$ is a set of links that form tunnel i. Given a traffic matrix entry $\mathcal{T}_{s,d}$ for a host pair, $\mathcal{U}_l$ is simply the summation of WCMP weights of tunnels l appears in multiplied respectively by the traffic matrix entry of the host pairs each tunnel belongs to. In particular, the new LP formulation for tunneling is described in Algorithm 1. By abstracting the feasible paths in the form of tunnels, the optimization problem reduces to simply aggregating utilization across working links and minimizing the maximum link utilization.

Path Selection. It is not practical to utilize *all* paths between each source, destination pair since switches have limited table space to store these forwarding rules, and maintaining this huge amount of information on larger topologies might not be deployable in practice. When solving the optimization problem on these topologies, we enforce a *path budget* t, which limits the number of paths each source, destination pair is allowed to route traffic through. This adaptation keeps the neural network size moderate and able to run inference at acceptable speed. When solving the optimization problem, LLB currently tries to find the maximum number edge-disjoint paths with best effort given the budget t. If t paths which have complete disjoint edges cannot be found, LLB tries to minimize the number of overlapped edges. It is also straightforward to let LLB learn under different path selection policies by simply changing how the optimization problem derives its set of paths $\mathcal{P}$. Fig. 2 demonstrates such tunneling and path selection on a Clos topology, where there are three tunnels for every given (source, destination) pair. One color represents one tunnel for traffic with source 1 and destination 3, where one of the three tunnels can be chosen as the path.

A note on failures. When a link l fails, the WCMP weight predicted by LLB for l should ideally be zero—i.e., traffic should not be sent over l. It is almost impossible for a complex, non-linear function such as a neural network to output exactly zero every time under a certain input. Therefore, we simply ignore the weight of the failed link and simply *split* the traffic on the active links based on the weights for the active links.

4.2. Efficient inference for LLB

A key requirement is that a learning-based load balancing framework should be able to perform inference at line rate. For every packet/flowlet/flow, we need to "run" the learned model to perform inference and decide the next hop. In high-speed networks (40/100Gbps), a software inference solution cannot process packets at line rate and will greatly increase latency of each

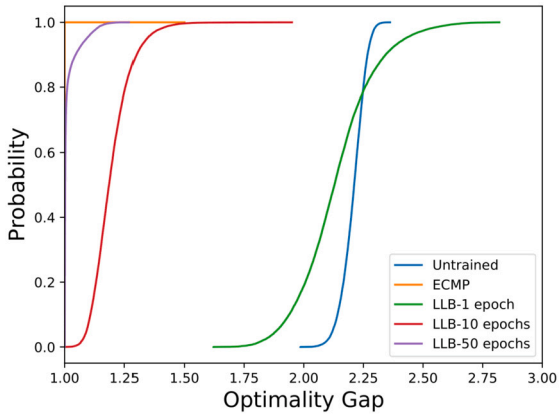

Fig. 3. Clos Topology Without Failures.

packet/flowlet/flow. We can leverage recent advances in building fabrics and transport protocols which are robust to reordering—e.g., 1RMA [39] proposes providing ordering guarantees to applications which desire it in software via a shim layer.

One option is to perform inference using programmable switches like Barefoot Tofino. While programmable switches support complex processing capabilities in the network at line-rate, however, their architectures are not suited for complex floating point operations and do not have the memory required to store large models. We attempted a design of a fully-connected neural network in P4, but it was not scalable due to the fixed numbers of pipelines and stages of a hardware switch.

We propose to implement the LLB's inference system on a programmable smartNIC. Programmable smartNICs [1], which now offer increasing computing resources, have brought up the opportunity for data centers to offload general applications [31] and stateful network functions [35]. With the combination of distributed inference techniques, these progressions have made it possible to run inference systems on smartNICs at a large scale while maintaining low latency. The programmable NICs would collect the input traffic demands through a telemetry framework and run inference to generate WCMP weights for each switch. The NIC would then encode these weights in the packet headers. The programmable switches can parse these headers (using P4's support for custom parsers), and then perform load balancing according to the encoded WCMP weights [42].

Another benefit that comes with running inference on programmable smartNICs is *performance predictability*. When running on hardware, we can obtain an upper bound of the number of instruction cycles the inference of a fix-sized neural network takes to run. Hardware predictability provides a latency guarantee to the system that a host-based solution would fail to achieve due to unexpected software latency.

5. Evaluation

Our evaluation answers the following questions:
Q1. Can LLB adapt to unseen traffic demands?
Q2. Can LLB react to asymmetry and link failures?
Q3. Can LLB run on more complex, multi-tiered topologies?
Q4. Can LLB inference run on state-of-art smartNICs?

5.1. Experiment settings

Implementation Details. We implemented LLB using PyTorch the Gurobi optimizer to generate optimal WCMP weights for each traffic matrix/capacity/failure triplet in the dataset. The optimal weights are then later used to calculate the optimality gap against the LLB-predicted WCMP weights. All our experiments are run on Cloudlab [11] machines.
Topologies and Traffic Patterns. We consider two topologies for our experiments. The first topology is a two layer Clos topology with 8 leaves and 6 spines, The second topology is a 8-ary 2-flat flattened-butterfly [26] topology with 10 edge devices. Clos topologies utilize abundant link connectivity between the leaf and spine layers, and enjoy high path diversity and low latency. Flatten butterfly topologies are another class of topologies commonly seen in modern datacenters. Compared to Clos topologies, which have better fault tolerance, flattened butterfly topologies are less expensive to construct and are more scalable. For experiments with an asymmetric topology we set the capacity of the optical link from capacities studied in RADWAN [38]. In the asymmetric setting we consider one-link failures across the topology, which is the most commonly observed scenario in data centers [15]. For each topology, we sample 50000 samples from their respective distribution $D(C, \lambda, \mathcal{F})$, and partitioned the samples into train/test sets with an 80%/20% split. All the test data points were not observed during training.

To approximate the flow statistics reported in the Fbflow [36] network traces, we design distributions of the form discussed in Section 4.1 by using fbflow's average flow size as the center traffic matrix C. The two layer topology samples from a center with 8 hosts, denoted C_8, and similarly, the flattened-butterfly with 10 edges draws from a center matrix denoted C_{10}. LLB's learning network has three fully-connected layers, with each hidden layer consisting of 128 hidden units.

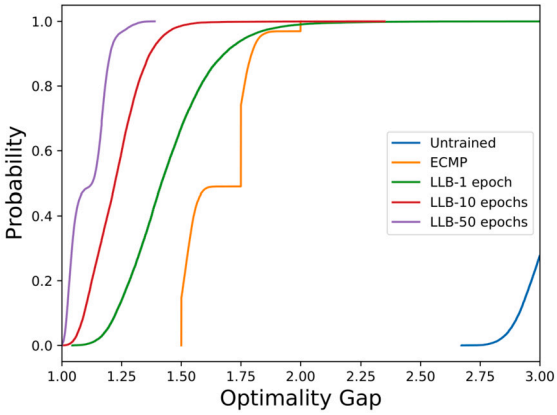

Fig. 4. Clos Topology with Failures and Capacity Asymmetry.

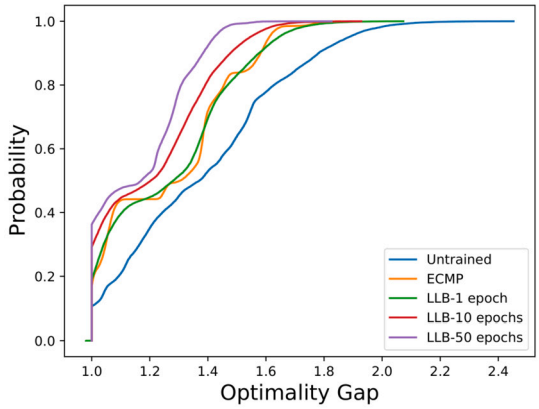

Fig. 5. Flattened Butterfly with Path Budget = 2.

5.2. Adapting to unseen traffic demands

We first consider topologies with no failures and equal link capacities. Data points used in Fig. 3 were drawn from $\mathcal{D}(C_4, \lambda, \vec{0})$, where λ is the average of C_4. The zero vector indicates that no failures are present in the sampled points. Data points used in Fig. 4 were drawn from $\mathcal{D}(C_4, \lambda, \mathcal{F})$, and points in Figs. 5 and 6 were drawn from $\mathcal{D}(C_8, \lambda, \mathcal{F})$. Fig. 3 shows the optimality gap of LLB under symmetric topologies—i.e., how far is LLB from the optimal WCMP weights for each sampled configuration. When LLB's DL model is untrained, as anticipated the optimality gap is bad: all samples are over 200% more worse than the optimal solution. As training progresses, after around 50 epochs LLB converges to a solution close to the optimal solution, which is equivalent to weights derived in ECMP under this setting, since all paths are equally lengthed and have identical capacity. Results for Figs. 3, 4, 5 and 6 were also derived from unseen traffic in the test dataset, which shows the generalizability of LLB. We answer **Q1** based on these results: LLB can adapt to unseen traffic demands in the distribution.

5.3. Adapting to asymmetry and failures

We also analyze how LLB adapts to asymmetry and failures with the asymmetric topology. During dataset generation for results in Figs. 4, 5 and 6, we combined various failure and link capacity constraints with traffic matrices drawn from the referenced distribution 5.1. Fig. 4 shows that LLB converges to a close-to-optimal solution even under failures and capacity asymmetry, and outperforms static methods like ECMP. Figs. 5 and 6 shows LLB trained on a flattened-butterfly topology, with Fig. 5 showing results with a path budget of 2, and Fig. 6 with a budget of 4. We can observe that under these more complicated topologies, LLB is still able to learn a policy for the underlying traffic distribution. In both cases, after training for more than 10 epochs LLB outperforms ECMP with a better optimality gap. The key insight is that LLB detects capacity asymmetry among paths caused by either the inherent topological structure or link failures, and adjusts the WCMP weights of those paths accordingly. We draw the conclusion for **Q2** from these results: LLB can adapt to asymmetry and link failures in the topology, and can generalize to other more complex topologies.

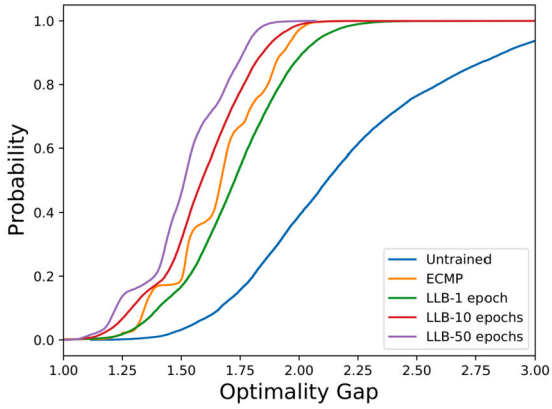

Fig. 6. Flattened Butterfly with Path Budget = 4.

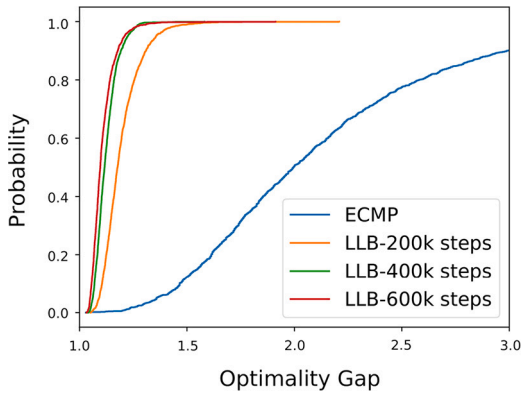

Fig. 7. Test Matrices Sampled from Traffic Matrix ball with 1Δ.

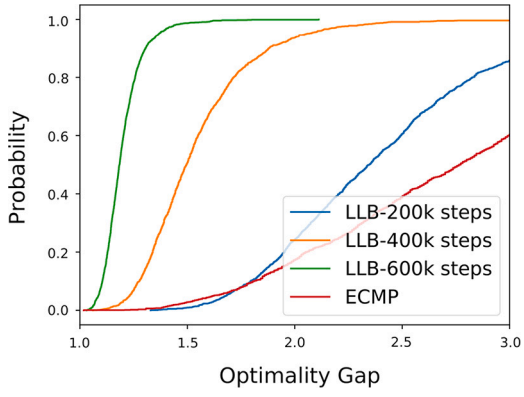

Fig. 8. Test Matrices Sampled from Traffic Matrix ball with 2Δ.

5.4. Multi-tier topology with tunnels

We also train LLB on a three layer Clos topology with [8, 4, 4] nodes from the leaf layer to spine layer respectively to demonstrate the use of tunnel selection. For the three layer topology, 15 tunnels are selected for each (source, destination pair). Fig. 7 shows the optimality gap for a three layer Clos topology with failures and capacity asymmetry. As anticipated ECMP's performance degrades drastically when there is asymmetry and failure present in the network (Fig. 8). LLB converges after around 600,000 steps, and over 80% of the sampled test traffic matrices have an optimality gap of less than 1.25. These results provide answers to **Q3**: LLB is able to react to topology changes and asymmetry even under multi-tier topologies.

LLB's performance can be further improved if we can dynamically adjust it's inferred WCMP weights at a finer granularity. If we can detect intermediate traffic demands at a finer timescale, potentially even capturing the existing of flowlets, we can expect LLB's enhanced performance since WCMP weights are re-calculated more frequently.

5.5. LLB inference on smartNICs

We implemented our 3-layer neural network on the AlphaData ADM-PCIE-9V3 Programmable FPGA smartNIC [1]. Profiling is done via the hls4ml [10] package, where the package translates deep learning models written in Keras or Pytorch into FPGA code, and simulates how many cycles inference takes to complete. Our microbenchmarks showed that running at 300 MHz, this programmable NIC can sustain per-packet inference at 120 Gbps, and it imposes a 85 ns per inference latency. Compared to runtimes of the multi-commodity flow (MCF) solver reported in Section 2, inference on NICs is 6 orders of magnitude faster to execute. Alternately, inference can potentially be accelerated further if it was implemented on a custom ASIC on the NIC (with a limited API for programmability to allow for model updates) optimized specifically for funcionalities like inference. Programmable NIC platforms with such embedded ASICs for custom functions have been proposed recently [39,41], and it is conceivable that such NICs may have an inference ASIC on them in the future.

To avoid potential high latency overhead on existing programmable smartNICs, we can conduct inference, and correspondingly load balancing, at the granularity of flowlets. If we set the inter-flowlet inactivity timeout to 120 ns, then we can prefetch the load balancing decision for an upcoming flowlet by conducting inference 60 ns after the previous flowlet to the same destination ended. Typical flowlet inactivity timeouts are set to 500 ns; a 120 ns timeout results in smaller flowlets and thus effective load balancing [19]. To answer **Q4**: LLB can be deployed on state-of-art smartNICs.

We now further evaluate the computational overhead of the LLB's load balancing solution by comparing it with centralized approaches such as Hedera and B4. Hedera's performance in moderate to large-scale networks shows a runtime variation from 1 millisecond to 200 milliseconds [3], whereas B4's traffic engineering (TE) solution averages a runtime of 0.3 seconds, peaking at 0.8 seconds daily [23]. In contrast, LLB's proposed per-solution 85 ns runtime, is faster by 6 to 7 orders of magnitude. This improved efficiency underscores LLB's viability for practical deployment.

Other distributed approaches are stateless and don't rely on a centralized strategy. While this eliminates the computational overhead associated with centralized systems, it introduces challenges in achieving optimal results, adaptability, and generalizability, as discussed in Section 2.2.

6. Related work

Data Center Network Load Balancing. Load balancing is a crucial component in data center networks. Efficient load balancing is essential for applications to deliver performance guarantees such as high throughput and low latency. Host pairs in data centers typically consist of multiple paths, and the load balance algorithm is responsible for distributing traffic among these paths to reach applications' performance requirements. Well-known flow-based load balancing techniques include methods like equal-cost multi-path (ECMP) and weighted-cost multi-path (WCMP), where flows are routed through different paths based on path weights.

Programmable Network Devices. Programmable network devices that emerged in recent years have allowed the implementation and deployment of new functionalities such as custom network protocols, custom header parsing and network function offloading. Theses programmable devices allow developers to write complex controlling logic and packet processing pipelines that a lot of times can be run at line rate. A few classes of programmable network devices include programmable switches like the Barefoot Tofino P4 switch and the Broadcom Stingray smartNIC. The Tofino switches allow custom header parsing and match-action logic to be run at line rate in the dataplane, and smartNICs like Stingray enable the support of complex offloads and in-network computation tasks.

7. Conclusion and future work

We presented a learned load balancing framework that can adapt to evolving traffic patterns and topologies. While we showed that such a framework can be made practical, the concept of learned load balancing raises interesting questions, some of which we describe here.

When traffic characteristics are unpredictable, it may be beneficial to learn online using real traffic. Continuously refining the model can yield better strategies. However, convergence in online learning is slow and the learned model performance can be poor before convergence if we re-learned constantly. We are exploring addressing this challenge using transfer learning, i.e., we will reuse the offline learning models to bootstrap the online learner.

LLB leverages simple aggregate traffic characteristics during training. With richer telemetry frameworks, deeper statistics about network data might become available, such as instantaneous queue lengths and packet drop/marking statistics. How to leverage these network statistics and combine them with other application-level data sources, e.g., from cluster schedulers and L7 load balancers, to further improve the quality of load balancing is another important challenge.

Another promising direction of future work is to have LLB learn optimization formulations that incorporate different path selection paradigms. The current formulation of LLB assumes that paths are statically pre-selected before training. As path selection also has great influence on the robustness of load balancing and traffic engineering [28], including such mechanisms could further improve the performance of learning-based solutions like LLB.

CRediT authorship contribution statement

Brian Chang: Software, Validation, Visualization, Writing – original draft, Writing – review & editing, Formal analysis, Investigation, Methodology, Conceptualization, Data curation. **Kausik Subramanian:** Data curation, Formal analysis, Investigation, Methodology, Validation, Visualization, Writing – original draft, Conceptualization, Writing – review & editing. **Loris D'Antoni:** Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing. **Aditya Akella:** Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Project administration, Supervision, Validation, Visualization, Writing – original draft, Writing – review & editing, Resources, Software, Methodology.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] [n. d.]. ADM-PCIE-9V3 - high-performance network accelerator, https://www.alpha-data.com/dcp/products.php?product=adm-pcie-9v3 ([n. d.]).

[2] [n. d.]. improving network monitoring and management with programmable data planes, https://p4.org/p4/inband-network-telemetry/ ([n. d.]).

[3] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al., Hedera: dynamic flow scheduling for data center networks, in: Nsdi, vol. 10, 2010, pp. 89–92.

[4] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, et al., CONGA: distributed congestion-aware load balancing for datacenters, in: Proceedings of the 2014 ACM Conference on SIGCOMM, 2014, pp. 503–514.

[5] Mahmoud Bahnasy, Fenglin Li, Shihan Xiao, Xiangle Cheng, DeepBGP: a machine learning approach for BGP configuration synthesis, in: Proceedings of the Workshop on Network Meets AI & ML, 2020, pp. 48–55.

[6] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al., P4: programming protocol-independent packet processors, ACM SIGCOMM Comput. Commun. Rev. 44 (3) (2014) 87–95.

[7] Rich Caruana, Alexandru Niculescu-Mizil, An empirical comparison of supervised learning algorithms, in: Proceedings of the 23rd International Conference on Machine Learning, 2006, pp. 161–168.

[8] Brian Chang, Aditya Akella, Loris D'Antoni, Kausik Subramanian, Learned load balancing, in: Proceedings of the 24th International Conference on Distributed Computing and Networking, 2023, pp. 177–187.

[9] Li Chen, Justinas Lingys, Kai Chen, Feng Liu, AuTO: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18), Association for Computing Machinery, New York, NY, USA, 2018, pp. 191–205.

[10] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, Z. Wu, Fast inference of deep neural networks in FPGAs for particle physics, J. Instrum. 13 (07) (jul 2018) P07027–P07027, https://doi.org/10.1088/1748-0221/13/07/p07027.

[11] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, Prabodh Mishra, The design and operation of CloudLab, in: 2019 USENIX Annual Technical Conference (USENIX ATC 19), USENIX Association, Renton, WA, 2019, pp. 1–14, https://www.usenix.org/conference/atc19/presentation/duplyakin.

[12] Bernard Fortz, Mikkel Thorup, Increasing Internet capacity using local search, Comput. Optim. Appl. 29 (2004) 13–48.

[13] Monia Ghobadi, Ratul Mahajan, Amar Phanishayee, Nikhil Devanur, Janardhan Kulkarni, Gireeja Ranade, Pierre-Alexandre Blanche, Houman Rastegarfar, Madeleine Glick, Daniel Kilper, ProjecToR: agile reconfigurable data center interconnect, in: Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16), Association for Computing Machinery, New York, NY, USA, 2016, pp. 216–229.

[14] Soudeh Ghorbani, Zibin Yang, P. Brighten Godfrey, Yashar Ganjali, Amin Firoozshahian, Drill: micro load balancing for low-latency data center networks, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, 2017, pp. 225–238.

[15] Phillipa Gill, Navendu Jain, Nachiappan Nagappan, Understanding network failures in data centers: measurement, analysis, and implications, in: Proceedings of the ACM SIGCOMM 2011 Conference, 2011, pp. 350–361.

[16] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, Songwu Lu, BCube: a high performance, server-centric network architecture for modular data centers, in: Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM '09), Association for Computing Machinery, New York, NY, USA, 2009, pp. 63–74.

[17] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, Varugis Kurien, Pingmesh: a large-scale system for data center network latency measurement and analysis, in: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15), Association for Computing Machinery, New York, NY, USA, 2015, pp. 139–152.

[18] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, Walter Willinger, Sonata: query-driven streaming network telemetry, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18), Association for Computing Machinery, New York, NY, USA, 2018, pp. 357–371.

[19] Keqiang He, Eric Rozner, Kanak Agarwal, Wes Felter, John Carter, Aditya Akella, Presto: edge-based load balancing for fast datacenter networks, in: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15), Association for Computing Machinery, New York, NY, USA, 2015, pp. 465–478.

[20] Thomas Holterbach, Edgar Costa Molero, Maria Apostolaki, Alberto Dainotti, Stefano Vissicchio, Laurent Vanbever, Blink: fast connectivity recovery entirely in the data plane, in: 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), USENIX Association, Boston, MA, 2019, pp. 161–176, https://www.usenix.org/conference/nsdi19/presentation/holterbach.

[21] Christian Hopps, et al., Analysis of an equal-cost multi-path algorithm, Technical Report. RFC 2992, November 2000.

[22] Kuo-Feng Hsu, Ryan Beckett, Ang Chen, Jennifer Rexford, David Walker, Contra: a programmable system for performance-aware routing, in: 17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20), 2020, pp. 701–721.

[23] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al., B4: experience with a globally-deployed software defined WAN, ACM SIGCOMM Comput. Commun. Rev. 43 (4) (2013) 3–14.

[24] Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, Flyways to de-congest data center networks, in: Proceedings of the 8th ACM Workshop on Hot Topics in Networks, 2009.

[25] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, Jennifer Rexford, Hula: scalable load balancing using programmable data planes, in: Proceedings of the Symposium on SDN Research, 2016, pp. 1–12.
[26] John Kim, William J. Dally, Dennis Abts, Flattened butterfly: a cost-efficient topology for high-radix networks, in: Proceedings of the 34th Annual International Symposium on Computer Architecture, 2007, pp. 126–137.
[27] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, Neoklis Polyzotis, The case for learned index structures, in: Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 489–504.
[28] Praveen Kumar, Yang Yuan, Chris Yu, Nate Foster, Robert Kleinberg, Petr Lapukhov, Chiun Lin Lim, Robert Soulé, Semi-oblivious traffic engineering: the road not taken, in: 15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18), 2018, pp. 157–170.
[29] Yann LeCun, Yoshua Bengio, Geoffrey Hinton, Deep learning, Nature 521 (7553) (2015) 436–444.
[30] Eric Liang, Hang Zhu, Xin Jin, Ion Stoica, Neural packet classification, in: Proceedings of the ACM Special Interest Group on Data Communication, 2019, pp. 256–269.
[31] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, Karan Gupta, Offloading distributed applications onto SmartNICs using IPipe, in: Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19), Association for Computing Machinery, New York, NY, USA, 2019, pp. 318–333.
[32] Hongzi Mao, Ravi Netravali, Mohammad Alizadeh, Neural adaptive video streaming with pensieve, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17), Association for Computing Machinery, New York, NY, USA, 2017, pp. 197–210.
[33] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, Mohammad Alizadeh, Learning scheduling algorithms for data processing clusters, in: Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19), Association for Computing Machinery, New York, NY, USA, 2019, pp. 270–288.
[34] Masoud Moshref, Minlan Yu, Ramesh Govindan, Amin Vahdat, Trumpet: timely and precise triggers in data centers, in: Proceedings of the 2016 ACM SIGCOMM Conference, 2016, pp. 129–143.
[35] Salvatore Pontarelli, Roberto Bifulco, Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, et al., Flowblaze: stateful packet processing in hardware, in: 16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19), 2019, pp. 531–548.
[36] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, Alex C. Snoeren, Inside the social network's (datacenter) network, in: Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, 2015, pp. 123–137.
[37] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, Amin Vahdat, Jupiter rising: a decade of clos topologies and centralized control in Google's datacenter network, SIGCOMM Comput. Commun. Rev. 45 (4) (Aug. 2015) 183–197, https://doi.org/10.1145/2829988.2787508.
[38] Rachee Singh, Manya Ghobadi, Klaus-Tycho Foerster, Mark Filer, Phillipa Gill, RADWAN: rate adaptive wide area network, in: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, 2018, pp. 547–560.
[39] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F. Wenisch, Monica Wong-Chan, Sean Clark, Milo M.K. Martin, Moray McLaren, Prashant Chandra, Rob Cauble, Hassan M.G. Wassel, Behnam Montazeri, Simon L. Sabato, Joel Scherpelz, Amin Vahdat, 1RMA: re-envisioning remote memory access for multi-tenant datacenters, in: Proceedings of the 2020 ACM SIGCOMM Conference, 2020, pp. 708–721.
[40] Ankit Singla, Chi-Yao Hong, Lucian Popa, P. Brighten Godfrey, Jellyfish: networking data centers randomly, in: Presented as Part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12), 2012, pp. 225–238.
[41] Brent Stephens, Aditya Akella, Michael M. Swift, Your programmable NIC should be a programmable switch, in: Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets '18), Association for Computing Machinery, New York, NY, USA, 2018, pp. 36–42.
[42] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, Aditya Akella, D2R: dataplane-only policy-compliant routing under failures, arXiv:1912.02402, 2019.
[43] Asaf Valadarsky, Michael Schapira, Dafna Shahaf, Aviv Tamar, Learning to route, in: Proceedings of the 16th ACM Workshop on Hot Topics in Networks (HotNets-XVI), Association for Computing Machinery, New York, NY, USA, 2017, pp. 185–191.
[44] Keith Winstein, Hari Balakrishnan, TCP ex machina: computer-generated congestion control, SIGCOMM Comput. Commun. Rev. 43 (4) (Aug. 2013) 123–134, https://doi.org/10.1145/2534169.2486020.
[45] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, Amin Vahdat, WCMP: weighted cost multipathing for improved fairness in data centers, in: Proceedings of the Ninth European Conference on Computer Systems, 2014, pp. 1–14.