MISTILL: DISTILLING DISTRIBUTED NETWORK PROTOCOLS FROM EXAMPLES

Anonymous authors
Paper under double-blind review

ABSTRACT

New applications and use-cases in data center networks require the design of Traffic Engineering (TE) algorithms that account for application-specific traffic patterns. TE makes forwarding decisions from the global state of the network. Thus, new TE algorithms require the design and implementation of effective information exchange and efficient algorithms to compute forwarding decisions. This is a challenging and labor and time-intensive process. To automate and simplify this process, we propose `Mistill`. `Mistill` distills the forwarding behavior of TE policies from exemplary forwarding decisions into a Neural Network. `Mistill` learns which network devices must exchange state with each other, how to process local state to send it over the network, and how to map the exchanged state into forwarding decisions. We show the ability of `Mistill` to learn distributed protocols with three examples and verify their performance in simulations. We show that the learned protocols closely implement the desired policies.

1 INTRODUCTION

Data center workloads are diverse and range from client-server applications, machine learning, and web-traffic to high-performance computing applications (Benson et al. (2010); Roy et al. (2015); Zhang et al. (2017)). Each workload has different requirements towards Traffic Engineering (TE). For example, optimizing the completion time of individual flows is desirable for client-server applications (Dukkipati & McKeown (2006)), but leads to decreased performance in machine learning workloads (Chowdhury et al. (2014)). To achieve the best performance, each workload needs a TE scheme that is tailored towards its specific requirements. Indeed, new applications regularly trigger the development of new TE schemes in the networking community (Katta et al. (2016); Chowdhury et al. (2014); Chen et al. (2018); Benson et al. (2011)).

Obtaining a deployable distributed protocol from an initial TE design is challenging. It requires the specification of the exchanged data, the processing of the data, and the algorithm to compute the forwarding decisions that result in the desired policy (Katta et al. (2016)). This is challenging, since the protocol must meet line-rate, exchange information at (sub)millisecond granularity, and cope with limited computational requirements. All steps must be efficient and the design holistic, e.g., integrates `CONGA` a substantial part of the calculation for the forwarding decision into the exchange of information (Hsu et al. (2020)). The complexity of the task makes customized TE schemes challenging for smaller enterprises, potentially resulting in competitive disadvantages.

To automate and simplify the translation of a TE policy to a distributed protocol, we propose `Mistill`. `Mistill` is a system that learns crucial components of distributed network algorithms with Machine Learning (ML). `Mistill` distills the forwarding behavior of a centralized approach from exemplary decisions into a Neural Network (NN). `Mistill` automates the process to obtain a protocol in two ways: 1) `Mistill` removes the need to manually design the logic that makes the forwarding decisions on the devices. 2) `Mistill` learns the processing and exchange of network state information jointly with the forwarding behavior.

Our main contributions in this work are: 1) The design of an ML model that can learn forwarding behavior and has the potential to be deployed on real hardware; 2) we show that `Mistill` can learn the forwarding behavior of three TE policies and evaluate the learned protocols in simulations. 3) We analyze the learned messages and their exchange. 4) We show that `Mistill` generalizes to previously unseen traffic patterns. We find that `Mistill` can learn distributed protocols that

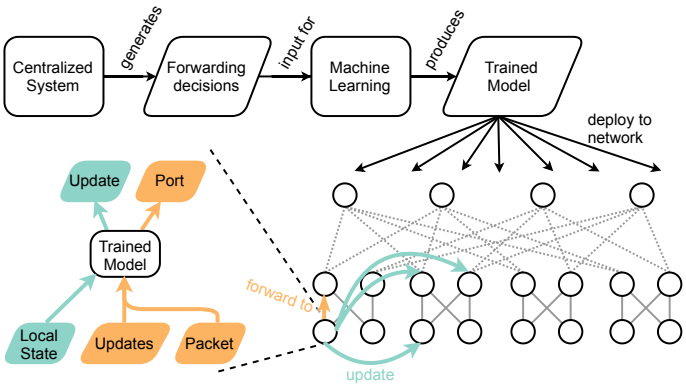


Figure 1: A high-level overview of `Mistill`. First, `Mistill` trains a NN with forwarding decisions. `Mistill` uses forwarding decisions from an in initial prototype of the TE scheme. Then, `Mistill` deploys the trained NN to the switches, which query the NN for forwarding decisions.

closely implement the desired policy, and is robust towards distributional shift in the input. Further, we show that the NN learns an information exchange that resembles an edge-cover, and analyze the content of learned messages. We thus show that `Mistill` can generate information to improve, e.g., the information exchange of other TE schemes.

The document is organized as follows. Sec. 2 describes the application scenarios and the requirements towards ML models. Sec. 3 describes the neural architecture and justifies design choices. Sec. 4 describes experiments and presents results and visualizations of the learned protocols. Sec. 5 discusses related work, and Sec. 6 concludes the paper.

2 Application Scenario

`Mistill` learns a distributed protocol from exemplary solution data. Fig. 1 illustrates the process and expected outcome of `Mistill`. Fig. 1 shows that the first component is a centralized implementation of a TE policy. For example, a packet or flow level simulation, where an Integer Linear Program or heuristic algorithm computes forwarding decisions based on the entire network state. The forwarding decisions specify how a switch forwards traffic to its neighbors, given the current state of the network. Usually, new TE schemes are evaluated in such a setting first to establish their performance and correctness.

Here `Mistill` hooks in and uses the forwarding decisions to train a ML model, i.e., a Neural Network (NN). `Mistill` trains the model offline, i.e., *not* on the network devices. The NN learns three aspects: first, how to encode local state in update messages, second how to exchange the update messages, and third, how to map the exchanged state into forwarding decisions.

After training, `Mistill` deploys the model to switches, e.g., with P4 (Siracusano & Bifulco (2018)). All switches in the network have *the same* model. `Mistill` distinguishes switches by conditioning the output of the model on a switch identifier. During training, the model learns how to vary the forwarding decisions based on this identifier, and thus learns the correct behavior for each switch. The switches use the model in inference mode, i.e., they do only forward passes and *no* backward passes. We stress that the network devices do *not* change the model.

The goal of this work is thus not to outperform existing TE strategies in the contexts for which they have been optimized. Instead, the goal of this work is to facilitate the automatic generation of novel TE schemes without asking network operators to take the time, or acquire the expertise for designing highly optimized protocols and implementing them, e.g., in P4.

2.1 DATA CENTER NETWORK REQUIREMENTS

The goal of `Mistill` is to distill network protocols for data-center networks from exemplary data. We focus on Clos-topologies due to their popularity and simple structure Zhang et al. (2019). See Appendix B for details. The network protocols must meet three requirements. They must react within milliseconds; they must be data efficient and computational tractable (Hsu et al. (2020)).

The learned network protocols must react to changes in the network within milliseconds, e.g., changes in link utilization (Alizadeh et al. (2014)).Thus, network devices must exchange state information frequently, and incorporate the state from distant parts of the network in their forwarding decisions (Alizadeh et al. (2014)). TE protocols exchange state updates at (sub-)millisecond scale (Katta et al. (2016)). Thus, the updates must be small to keep the resulting overhead low. To effectively utilize the frequent updates, switches must make forwarding decisions frequently, e.g., for each new flow. Thus, the decision logic should reside in the data-plane (Alizadeh et al. (2014)). This limits the computational demand the decision logic can express towards the switch. The learned decision logic must thus be computationally efficient to meet the limited computational capacities.

2.2 ML MODEL LIMITATIONS

To satisfy the computational requirements, `Mistill` uses a NN to learn a distributed protocol. We belief that NNs are a good choice for three reasons.

1) NNs are general function approximators (Glorot et al. (2011)). This is important because forwarding decisions can depend on the network state in a non-linear way. For example, minimizing the maximum link utilization results in forwarding decisions that do not depend linearly on the network state. A small change in the utilization can lead to a sudden change in the forwarding policy.

2) NNs can be executed with the limited computational resources in the data-plane. Here, the choice of activation function is important. The ReLU activation requires only a check for negativity (Glorot et al. (2011)). A NN with ReLU activations uses addition, multiplication and a check for negative values. Further, NNs can be quantized to integer or binary computations (Gholami et al. (2021)).

3) The execution of NNs is easy to parallelize and accelerators for NN inference already exist for network equipment (Sanvito et al. (2018)). Research efforts show that NNs can be executed in the data-plane at line rate already on today's ASICs (Siracusano & Bifulco (2018)). Similar results have been obtained on network devices with SmartNICs (Sanvito et al. (2018)). We thus believe that switches will be increasingly capable of executing NNs.

3 NEURAL ARCHITECTURE

The NN architecture consists of four modules: the Localization Module (LOCMOD), the HNSA Module (HNSAMOD), the Network State Module (NSMOD), and the Forwarding Module (FWD MOD). Fig. 2 shows how the modules interact and represents inputs and outputs with hashes. See Appendix C for sketches of technical realizations of `Mistill`.

Output. The NN has three outputs: the out-port, Hidden Node State Advertisements (`HNSA`s)[1], i.e., update messages that inform other network elements about the local state of the sending device, and State Interest (SI), signaling which nodes need to exchange information with each other.

Input. The NN has four inputs: `Link State`, `All Nodes`, `Destination` and `Location`. `Link State` corresponds to the switch local state, such as the availability and utilization of incident links. The NN transforms the `Link State` into `HNSA`s . `Location` represents the switch that executes the NN. The NN uses `Location` to condition the forwarding decisions on individual switches, Similarly, the NN uses `Destination` to condition the output on the the destination of a packet. Lastly, `All Nodes` contains the representation of all switches in the network. The NN uses `All Nodes` to learn which `HNSA`s are needed to make a forwarding decision for a specific destination on a specific switch.

[1] In analogy to Link State Advertisements (LSAs) in link-state routing protocols. *Hidden* refers to the fact that the content of the messages is the activations of a hidden layer of the NN. Further, `HNSA`s differ from LSAs in that they are not per link but node.

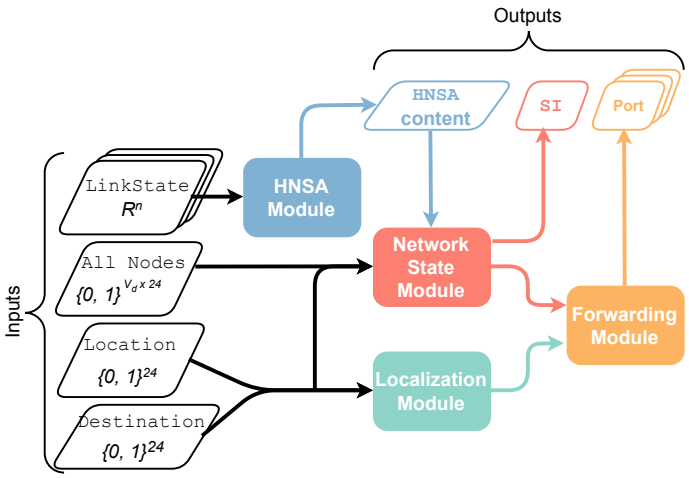


Figure 2: The NN architecture consists of four modules: The LocMod, the HNSAMod, the NSMod and the Fwd Mod.

Training. The NN is trained end-to-end. Training end-to-end tailors HNSAs and SI to a forwarding policy. The loss function that is optimized depends on the forwarding objective. Objectives can be interpreted as multi-label classification or as learning of a multinomial distribution. We use: $\mathcal{L}_{mn} = -\sum_{i=1}^{N} p(x_i) \log p(y_i)$ as loss function for the multinomial distribution, and $\mathcal{L}_{ml} = -\sum_{i=1}^{N} (p(x_i) \log p(y_i) + (1 - p(x_i)) \log(1 - p(y_i)))$ as loss function for the multi-label classification. N corresponds to the number of outputs, $p(x_i)$ is the ground truth and $p(y_i)$ is the prediction of the NN. The loss $\mathcal{L}_{mn}$ can learn distributional targets to split traffic over adjacent nodes. $\mathcal{L}_{ml}$ can learn policies that have one or multiple equally good paths.

Inference. After training finishes, the NN is deployed to the switches. The switches use the NN for three tasks: 1) to compute HNSAs from their local state, 2) to calculate the out-port for traffic, and 3) to select the HNSAs necessary to calculate the out-port. The switches do not change the weights of the NN. The switch also never execute the full NN. To compute HNSAs, the switch executes the HNSAMod. To compute the out-port, the switch executes the LocMod, NSMod and Fwd Mod, but not the HNSAMod. The switch uses the HNSAs it received from other nodes instead.

3.1 Data Format

The inputs Location, Destination and All Nodes correspond to the last three octets of IPv4 addresses, i.e., are binary vectors in $\{0, 1\}^{24}$. We use the addressing scheme for Fat-Trees Al-Fares et al. (2008). We experimented with different node representations. Due to brevity we enclose the results in Appendix D.

The Link State $\in \mathbb{R}^6_{\geq 0}$ is a real-valued vector and represents incident edges. The first four attributes are a one-hot encoding of the link availability in each direction. For example, 10 10 means that both directions are up, and 10 01 means that one direction is up and one is down. The fifth and sixth attributes are the weight of the link in each direction, representing, e.g., link utilization. The link state of a switch is the concatenation of the vectors of all its links, resulting in a vector from $\mathbb{R}^{6 \cdot \hat{d}}_{\geq 0}$, where $\hat{d}$ is the maximum degree of switches in the network. Vectors of switches with a smaller degree are zero-padded to this length.

3.2 Learning the HNSA content

The HNSAMod transforms the Link State of a switch into a binary vector. The binary vector is the content of a HNSA . The vector corresponds to a hidden layer with binary activations. Since switches send the HNSA over the network, binary activations are better than real-valued activations. A binary representation allows the NN to have more neurons compared to real-valued activations. For example, a binary vector of 128 bits corresponds to four floating-point numbers. Since NNs store

information in a distributed representation, having more neurons with low precision is better than having fewer neurons with high precision Glorot et al. (2011). We sample the binary activations from m independent, one-hot encoded categorical distributions of arity n: `HNSA` $\sim \mathrm{Cat}(n, \theta_1) \mid\mid \ldots \mid\mid \mathrm{Cat}(n, \theta_m)$. The NN calculates the parameters θ_i from the `Link State` by passing the `Link State` through a feed forward NN with one hidden layer and ReLU activation. To keep the NN differentiable, we use a re-parameterization trick to sample from the categorical distributions (Jang et al. (2017); Maddison et al. (2017)). During training, the NN learns how to parameterize the categorical distributions from the `Link State` to encode information in the binary hidden layer. The arity n and number of distributions m are hyperparameters of the NN architecture and optimized during training. We achieved the best results with an arity of 2. During operation, we set the neuron with the largest logit value in each block to one to keep inference cheap.

3.3 Learning to Communicate

The NSMod learns: 1) whose `HNSA`s are needed to make a forwarding decision on a specific switch for a given destination, i.e., the SI output, and 2) how to process `HNSA`s to make the necessary forwarding decision.

To learn the SI and process `HNSA`s , we use scaled dot-product attention (Vaswani et al. (2017)), which can be interpreted as a learned lookup table that retrieves `HNSA`s based on the switch that executes the NN and a given destination. The attention mechanism has three inputs: Keys K, Queries Q and Values V. K, Q and V are matrices, where K and V must have the same number of rows (Vaswani et al. (2017)). K and Q are used to compute attention-weights A, with which a convex combination of V is computed. The attention weights A are $A = \mathrm{gumbelSoftmax}\left(\frac{1}{\sqrt{r}} W_q Q \cdot (W_k K)^T\right)$, where W_q and W_k are a trainable linear transformation of Q and K, r is the number of rows of K and $\mathrm{gumbelSoftmax}$ is the gumbel softmax reparameterization trick (Jang et al. (2017); Maddison et al. (2017)). The output of the attention layer is $A \cdot (W_v V)$, where W_v is a trainable linear transformation of V. In `Mistill`, Q corresponds to the concatenation of `Destination` and `Location`, K to `All Nodes` and V to `HNSA`s. We use multiple attention heads, i.e., parallel attention mechanisms and concatenate their output. The concatenated output is passed through a hidden layer with ReLU activation. This output is passed as input into the Fwd Mod.

The SI are the attention weights of the individual attention heads. Non-zero entries signal that the corresponding `HNSA` contributes information for the forwarding decision. To enforce sparse patterns, we use the $\mathrm{gumbelSoftmax}$ trick to calculate attention weights (Jang et al. (2017); Maddison et al. (2017)). Thus, each attention head selects one switch. The number of attention heads is an upper bound on the messages that are exchanged, and allows control over the signaling overhead.

3.4 Learning to Forward

The last learning task is to predict the forwarding decision of the central algorithm. The forwarding decision of the NN is based on the network state encoded in `HNSA`s , i.e., the output of the NSMod, the destination and the switch that makes the forwarding decision. The NN learns to combine this information to react to changes in the state of the network. In addition, the NN learns how forwarding decisions differ for switches and destinations.

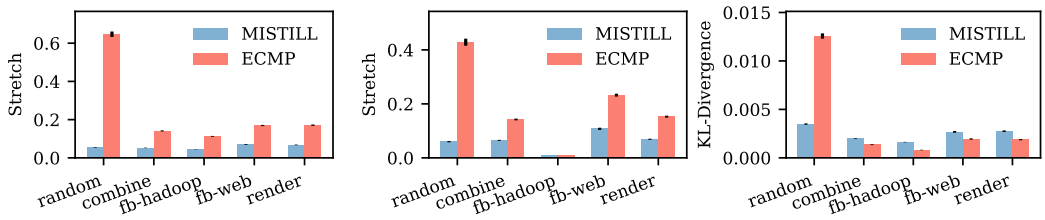
The calculation of the out-port requires the execution of the NSMod, the LocMod and the Fwd Mod. The LocMod and the Fwd Mod each consist of one feed forward NN with one hidden layer and ReLU activation.

4 Experiments

`Mistill` learns a TE policy protecting the data center network from traffic whose destination is not reachable. If no route to a host exists, then traffic is dropped. This feature can be important for applications that send larger amounts of data with the User Datagram Protocol (UDP). Existing TE schemes would forward this traffic to the last hop of the network and thus waste bandwidth (Hsu et al. (2020); Dutt). If a route exists, we consider three policies to forward traffic: 1) `WCMP`: Splits traffic over neighbors in proportion to the available down-stream bandwidth, 2) `LCP`: forwards traffic

	HNSAMOD		LOCMOD	NSMOD					FWD MOD
	FCN	NumBlocks	FCN	Heads	W_q	W_k	W_v	FCN	FCNs
MinMax	108	64	90	14	48	41	17	109	$104, 84$
LCP	63	64	90	14	48	48	50	125	$102, 105$
WCMP	48	64	90	14	62	62	27	110	$76, 71$

Table 1: Hyper parameters for the best models.



(a) LCP (b) MinMax (c) WCMP

Figure 3: Bar plots for TE policies and traffic patterns.

along the path with the least weight, and 3) `MinMax`: Forwards traffic along the path with the smallest maximum link utilization. All policies should work in the presence of arbitrary node and link failures. We compare the quality of the learned protocols to Equal Cost Multi-Pathing (ECMP) and the actual forwarding decision.

4.1 DATA GENERATION AND NN TRAINING

We generate data for a $k = 8$ Fat Tree having 128 hosts and 80 switches (Al-Fares et al. (2008)). To make the protocols robust to node and link failures, we remove between $\{0, \ldots, 10\}$ edges and $\{0, \ldots, 5\}$ nodes at random from the topology. For the remaining edges, we sample edge weights uniformly at random. We then sample a source and destination pair and compute the forwarding decisions for each policy on the generated graph for the sampled pair.

The training set consist of 200 000 problem instances, the validation set of 50 000 problem instances. The final results are obtained from a packet level simulation. We use the Asynchronous Successive Halving Algorithm (ASHA) Li et al. (2018) implemented in the Ray Tune Liaw et al. (2018) library for hyperparameter optimization, and select the best performing configuration for the evaluation. Tbl. 1 lists the hyperparameter of the best performing models. For all policies, the HNSAMOD consists of one fully connected layer with ReLu activation, and a linear transformation that generates the logits for the gumbelSoftmax function. The LOCMOD consists of one fully connected layer with ReLu activation. The NSMOD consists of one MHA module, and the FWD MOD of two fully connected layers. We describe the training process and the hyper parameter search space in detail in Appendix A. Training a single model takes in the order of 5 hours on a NVIDIA Tesla V100 GPU. To evaluate the inference time, we implemented the NN in C and evaluate the `LCP` NN, the largest network, on an Intel(R) Core(TM) i7-7700HQ CPU with 2.80 GHz. Inference on a single CPU core without parallelization resulted in 4 ms, which is sufficient to make per-flow decisions in some data centers Benson et al. (2010). Moreover, parallelization can drastically reduced the inference time and expand the application scenarios of `Mistill`.

4.2 ROUTING PERFORMANCE

We evaluate five traffic patterns and perform 100 simulations with random node and edge failures for each pattern. Each simulation samples between zero and ten edge failures, and between zero and five node failures. The traffic patterns are: `random`, `combine`, `fb-hadoop`, `fb-web` and `render`. In `random`, the utilization value of each edge is sampled uniformly at random, and corresponds to the distribution of edge weights during training. The other patterns are created from the traffic matrices of datacenter applications. `fb-hadoop` is based on a Hadoop cluster in a facebook datacenter, `fb-web` is based on the production traffic of a facebook datacenter, `combine`

(a) Training Progress.

(b) Performance.

Figure 4: Training loss for varying number of blocks in the GS for the calculation of HNSAs.

is based on the Combine application, and `render` on the render application. Note that `combine`, `render`, `fb-hadoop` and `fb-web` result in utilization patterns in the network that strongly differ from the random utilization values the NNs were trained with.

LCP: Fig. 3a shows bar plots for the `LCP` policy, and compares `Mistill` to `ECMP`. The Y-Axis shows the average stretch of the objective, i.e., the ratio that `Mistill` and `ECMP` are worse than the optimal solution. Error bars correspond to the 99 % confidence interval of the mean. Fig. 3a shows that `Mistill` is better for all traffic patterns, with paths whose length is close to the optimal lengths. Further, Fig. 3a shows that the error of `Mistill` is similar across all traffic patterns. The NN thus generalizes to distributions of the inputs that it has never seen during training.

MinMax: Fig. 3b shows a similar plot as in Fig. 3a for the `MinMax` policy. Fig. 3b shows that `Mistill` is always better than `ECMP`, except for `fb-hadoop`. For `fb-hadoop` `Mistill` and `ECMP` are both almost zero. This is due to the specific pattern in `fb-hadoop`, in which the up-links of hosts are the most loaded links, and the TE policy has thus almost no impact on the objective. As for `LCP`, the NN generalizes to unseen input distributions.

WCMP: We use the Kullback-Leibler Divergence (KLD) to evaluate `WCMP`. Fig. 3c shows the average KLD for `Mistill` and `ECMP` for the traffic patterns. Error bars correspond to the 99 % confidence interval of the mean. Fig. 3c shows that `Mistill` is better than `ECMP` for the `random` traffic pattern. `ECMP` has slightly smaller average KLDs for the other patterns. This is because available bandwidth is relatively equally distributed over paths. A uniform distribution is thus a good guess and difficult to improve. Note how the average KLDs of `Mistill` decreaese for the other traffic patterns. This indicates that the NN also generalizes in this case to unseen input distributions.

In summary, our experiments show that `Mistill` can learn the forwarding behavior of three popular TE policies from exemplary samples. Further, we show that the NN generalizes to previously unseen input distributions.

4.3 HNSA Size

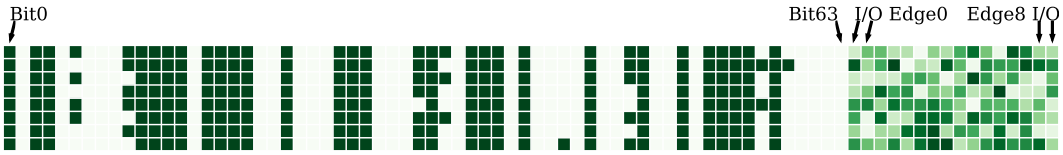
We investigate the impact of the number of blocks on the `MinMax` policy. Fig. 4a shows that an increasing number of blocks results in a smaller loss. Fig. 4a shows that the decrease in loss is strong during the initial stages of training and converges towards the end.

Fig. 4b compares the ratio between `ECMP` the true solution to the ratio of `Mistill`. The solid line corresponds to the mean, the shaded area to the mean's 99 % confidence interval. Fig. 4b shows that 8 blocks result in a protocol that performs better than `ECMP`. The protocol with 8 blocks, i.e., 8 bits, mostly accounts for link failures and little for edge weights. The performance of `Mistill` improves with more blocks and approaches the ground truth. The confidence interval shrinks with more blocks. The improvement between 8 and 16 blocks is larger than between 16 and 64 blocks. With 64 blocks, `Mistill` selects paths with similar metrics to the ground truth.

4.4 Visualizations

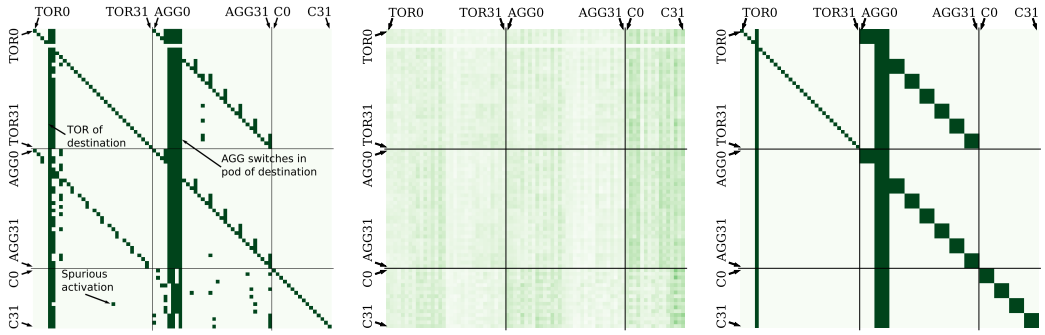
This section visualizes the learned `HNSA` messages and the attention weights. We compare the attention weights of the gumbelSoftmax activation with the weights of a model trained with the sparsemax activation for the `MinMax` policy.

(a) Link failures.



(b) Edge Weights.

Figure 5: Bit pattern in `HNSA` messages for edge availability and edge weights.



(a) gumbelSoftmax (b) sparsemax (c) Edge Cover

Figure 6: Attention weights. Each row contains the weights for one switch over all other switches.

`HNSA` content Fig.5 shows exemplary `HNSA`s of `Mistill` for the `MinMax` policy. Fig. 5a illustrates how link failures reflect in the `HNSA`s. Fig. 5b illustrates how edge weights reflect in the `HNSA`s. The left part of Fig. 5a and Fig. 5b show the `HNSA`s. The right part shows the availability of edges in Fig. 5a, and the edge weights in Fig. 5b. White corresponds to a value of zero, green to a value of one.

Fig. 5a shows that a large portion of the bits encode the link availability. In fact, 56 out of 64 bits can be associated with link availability. Fig. 5a shows that some bits directly correspond to the availability of edges. `Bit59` and `Bit26` corresponds to `Edge8`, and `Bit4` to `Edge7`.

Consequently, Fig. 5b shows that most bits are constant for different edge weights. The NN uses eight out of the 64 bits to encode edge weights. The relation between bits and edge weights is more complex than between link availability and bits, and no clear correspondence is observable.

Attention Scores Fig. 6 illustrates the attention scores of all switches in the network for a specific destination node. To evaluate the effect of the gumbelSoftmax activation, we additionally trained a NN with the sparsemax activation function Martins & Astudillo (2016). Further, we handcrafted attention scores corresponding to an edge cover of all edges belonging to the shortest paths between a switch and the destination. Fig. 6a shows the attention scores of the gumbelSoftmax activation, Fig. 6b of the sparsemax activation, and Fig. 6c of the edge cover. The attention scores of individual attention heads are summed up and clipped to a maximum value of one.

Fig. 6 shows that the attention scores of the gumbelSoftmax resemble the edge cover in Fig. 6c. In contrast, the attention scores of the sparsemax cover all nodes. The attention scores in Fig. 6 show that the NN relies on the state of the ToR switch connected to the host and the aggregation switches in the pod of the destination. Further, all switches rely on their state. Some activations in Fig. 6a are

not intuitive. For example, it is unclear why one core switch relies on the state of a ToR switch in a different pod than the destination.

Fig. 6a and Fig. 6b also show that the $\mathrm{sparsemax}$ activation results in a dense pattern that results in a large signaling overhead of the learned protocol. The resulting protocol would exchange one message for each non-zero entry. Here, the $\mathrm{gumbelSoftmax}$ activation has a clear advantage and requires exchange of at most 10 messages, which is in the order of the edge cover, which requires at most 9 messages.

In summary, the majority of the bits in the `HNSA`s encode the edge availability. The attention scores resemble an edge cover of edges that belong to the shortest paths to a destination. This behavior does also hold for the `WCMP` and `LCP` policy.

5 Related Work

Routing and Traffic Engineering. TE solution for DC networks addressing different goals and challenges exist Noormohammadpour & Raghavendra (2017). Solutions range from using well established protocols such as OSPF Chiesa et al. (2016); Michael & Tang (2014) over new centralized designs Al-Fares et al. (2010); Benson et al. (2011); Zhang et al. (2016) to distributed protocols that operate directly in the data plane Hsu et al. (2020); Kandula et al. (2007); Alizadeh et al. (2014); Katta et al. (2016). The closest to `Mistill` is CONTRA Hsu et al. (2020), which proposes a high-level policy language expressing TE goals. In contrast to previous work, `Mistill` is *not* yet another TE scheme, but a method to automate the process to obtain such a scheme by distilling distributed protocols from data. In contrast to CONTRA, `Mistill` does not require a policy language or relies on an already established Inband-Network Telemetry approach. Instead, `Mistill` learns the exchange and the format of information from data.

ML and Networking. Combinations of ML and networking have been explored from different angles in the past Boutaba et al. (2018). Closest to `Mistill` are work from Geyer and Carle Geyer & Carle (2018) and Xiao et. al. Xiao et al. (2020), which rely on Recurrent Neural Networks to learn forwarding behavior. In contrast to previous work, `Mistill` focuses on data-center applications. Proposals in Geyer & Carle (2018); Xiao et al. (2020) focus on wide area networks and do not apply to data centers. Both proposals require the exchange of several messages between neighboring nodes, similar to a distance-vector algorithm, to accommodate a change in the network. This procedure is too slow for the fast-changing traffic conditions in data-center networks. `Mistill` does not suffer from this problem and maps state updates directly into forwarding decisions. Further, we are the first to design such a system with practical deployment in mind, meaning that key parts of the design of `Mistill` are such that `Mistill` can, in principle, be deployed in the network.

6 Conclusion

This paper investigates if it is possible to learn a distributed protocol that implements a custom TE policy in a data-driven manner. Accordingly, this paper proposes `Mistill`, an ML-based approach that learns distributed protocols from exemplary forwarding decisions. `Mistill` learns the processing and exchange of local information with other network elements and the computation of forwarding decisions from the exchanged information. Further, we design `Mistill` with deployment on hardware in mind. `Mistill` does not rely on calculations that are not available on network devices.

To show the applicability of `Mistill`, we learn the forwarding behavior of three TE policies and evaluate the learned protocols in simulations. Our results show that it is possible to learn distributed protocols from data that closely match the original forwarding decisions. We believe that our work opens many research opportunities for the future: learning routing protocols in arbitrary network topologies; or designing switch architectures that implement learned routing protocols.

REPRODUCIBILITY STATEMENT

We make all source code as well as the trained models available for the reviewers. Upon acceptance, we will make the code and the models publicly available. To improve reproducibility, we include scripts and manuals on how to recreate figures from the paper.

We will post the link to the anonymous git repository using the discussion forum. Once the discussion forum opens, we will post a link to the repository there with restricted visibility to the reviewers and area chairs.

REFERENCES

Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, August 2008. ISSN 0146-4833. doi: 10.1145/1402946.1402967. URL http://doi.acm.org/10.1145/1402946.1402967.

Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, Amin Vahdat, et al. Hedera: dynamic flow scheduling for data center networks. In *Nsdi*, pp. 89–92, 2010.

Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. *SIGCOMM Comput. Commun. Rev.*, 44(4):503–514, August 2014. ISSN 0146-4833. doi: 10.1145/2740070.2626316. URL https://doi.org/10.1145/2740070.2626316.

Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pp. 267–280, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0483-2. doi: 10.1145/1879141.1879175. URL http://doi.acm.org/10.1145/1879141.1879175.

Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pp. 8:1–8:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1041-3. doi: 10.1145/2079296.2079304. URL http://doi.acm.org/10.1145/2079296.2079304.

Raouf Boutaba, Mohammad A Salahuddin, Noura Limam, Sara Ayoubi, Nashid Shahriar, Felipe Estrada-Solano, and Oscar M Caicedo. A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):1–99, 2018.

Li Chen, Justinas Lingys, Kai Chen, and 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, pp. 191–205, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5567-4. doi: 10.1145/3230543.3230551. URL http://doi.acm.org/10.1145/3230543.3230551.

Marco Chiesa, Gábor Rétvári, and Michael Schapira. Lying your way to better traffic engineering. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pp. 391–398, 2016.

Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient Coflow Scheduling with Varys. In *SIGCOMM '14*, pp. 443–454, Chicago, IL, USA, August 2014. ACM. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2626315.

Nandita Dukkipati and Nick McKeown. Why Flow-completion Time is the Right Metric for Congestion Control. *SIGCOMM Comput. Commun. Rev.*, 36(1):59–62, January 2006. ISSN 0146-4833. doi: 10.1145/1111322.1111336.

Dinesh G. Dutt. *BGP in the Data Center*. O'Reilly Media, Inc. ISBN 9781491983409.

Fabien Geyer and Georg Carle. Learning and Generating Distributed Routing Protocols Using Graph-Based Deep Learning. In *Proceedings of the 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, Big-DAMA '18, pp. 40–45, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5904-7. doi: 10.1145/3229607.3229610. URL http://doi.acm.org/10.1145/3229607.3229610.

Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630, 2021. URL https://arxiv.org/abs/2103.13630.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pp. 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. JMLR Workshop and Conference Proceedings.

Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *KDD '16*, pp. 855–864, San Francisco, CA, USA, 2016. ACM. ISBN 9781450342322. doi: 10.1145/2939672.2939754.

William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *CoRR*, abs/1709.05584, 2017. URL http://arxiv.org/abs/1709.05584.

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

Eric Jang, Shixiang Gu, and Ben Poole. Categorical Reparameterization with Gumbel-Softmax. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=rkE3y85ee.

Srikanth Kandula, Dina Katabi, Shantanu Sinha, and Arthur Berger. Dynamic load balancing without packet reordering. *ACM SIGCOMM Computer Communication Review*, 37(2):51–62, 2007.

Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. HULA: Scalable Load Balancing Using Programmable Data Planes. In *Proceedings of the Symposium on SDN Research*, SOSR '16, pp. 10:1–10:12, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4211-7. doi: 10.1145/2890955.2890968. URL http://doi.acm.org/10.1145/2890955.2890968.

Liam Li, Kevin G. Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. Massively parallel hyperparameter tuning. *CoRR*, abs/1810.05934, 2018. URL http://arxiv.org/abs/1810.05934.

Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.

Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=S1jE5L5gl.

Andre Martins and Ramon Astudillo. From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification. In Maria Florina Balcan and Kilian Q. Weinberger (eds.), *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pp. 1614–1623, New York, New York, USA, June 2016. PMLR. URL http://proceedings.mlr.press/v48/martins16.html.

Nithin Michael and Ao Tang. Halo: Hop-by-hop adaptive link-state optimal routing. *IEEE/ACM Transactions on Networking*, 23(6):1862–1875, 2014.

Vinith Misra and Sumit Bhatia. Bernoulli Embeddings for Graphs. *CoRR*, abs/1803.09211, 2018.

Mohammad Noormohammadpour and Cauligi S Raghavendra. Datacenter traffic control: Understanding techniques and tradeoffs. *IEEE Communications Surveys & Tutorials*, 20(2):1492–1525, 2017.

Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. *SIGCOMM Comput. Commun. Rev.*, 45(4):123–137, August 2015. ISSN 0146-4833. doi: 10.1145/2829988.2787472. URL http://doi.acm.org/10.1145/2829988.2787472.

Davide Sanvito, Giuseppe Siracusano, and Roberto Bifulco. Can the Network Be the AI Accelerator?, 2018. URL https://doi.org/10.1145/3229591.3229594.

Giuseppe Siracusano and Roberto Bifulco. In-network Neural Networks. *CoRR*, abs/1801.05731, 2018. URL http://arxiv.org/abs/1801.05731.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 30*, pp. 6000–6010. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high speed ip routing lookups. In *SIGCOMM '97*, SIGCOMM '97, pp. 25–36, Cannes, France, 1997. ACM. ISBN 089791905X. doi: 10.1145/263105.263136.

Shihan Xiao, Haiyan Mao, Bo Wu, Wenjie Liu, and Fenglin Li. Neural Packet Routing. In *Proceedings of the Workshop on Network Meets AI & ML*, NetAI '20, pp. 28–34, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-8043-0. doi: 10.1145/3405671.3405813. URL https://doi.org/10.1145/3405671.3405813. event-place: Virtual Event, USA.

Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pp. 160–173, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4193-6. doi: 10.1145/2934872.2934880. URL http://doi.acm.org/10.1145/2934872.2934880.

Mingyang Zhang, Radhika Niranjan Mysore, Sucha Supittayapornpong, and Ramesh Govindan. Understanding Lifecycle Management Complexity of Datacenter Topologies. In *NSDI 19*, pp. 235–254, Boston, MA, USA, February 2019. USENIX Association. ISBN 978-1-931971-49-2.

Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. High-resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference*, IMC '17, pp. 78–85, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5118-8. doi: 10.1145/3131365.3131375. URL http://doi.acm.org/10.1145/3131365.3131375. event-place: London, United Kingdom.

A HYPE PARAMETER OPTIMIZATION

A.1 SHORTEST PATH

Finding the parameter was a two stage process. Step one was finding parameters that are able to reproduce the shortest path without any network state info. For this, we use only the LOCMOD. We experimented here with different node representations (see later appendix), as well as different network architectures. We tried multi-head attention with the IP of the destination as query and the IPs of the neighbors as keys and values. We also tried simple feed forward neural networks. It turned out, that a simple feed forward neural net with one hidden layer and the concatenation of destination IP and the IP of the current location was enough to reliably learn the forwarding behavior. We performed a parameter analysis for this with ASHA in Ray Tune. Hyper parameters were:

- batch size: $\{64, \ldots, 256\}$
- hidden layer sizes: $\{8, \ldots, 100\}^3$, i.e., up to three layers.
- learning rate: $[0.001, 0.0001]$.
- samples for ASHA: 100.

We used the adam optimizer.

We took the best performing model and did not change it when moving to the learning of network state.

A.2 NETWORK STATE

Search space:

- batch size: $\{64, \ldots, 128\}$
- final fcns in out mod: $\{50, \ldots, 128\}^{2,3}$ two or three layers.
- FCN for encoding links: $\{32, \ldots, 150\}$
- number of heads for MHA in NSMOD: $\{9, \ldots, 14\}$
- dim fcn for MHA module in NSMOD: $\{70, \ldots, 129\}$
- hidden dimension of MHA heads: $\{20, \ldots, 64\}$
- output dim of MHA heads: $\{16, \ldots, 4\}$
- learning rate: $[10^{-4.3}, 10^{-3.5}]$

Fixed parameters and the dimensions are the result of initial experimentations that gave us some impression on the parameter regions that are sensible and can result in good performance. Further tuning for smaller network models can be done in future work, i.e., we did not try to make the models as small as possible.

B CLOS-TOPOLOGIES

Data center networks naturally seem as a good fit to start with learning network protocols. They have regular and repetitive structures and their addressing schemes can encode the location of network nodes. Clos networks are one such common data center topology. One special instance of Clos networks is the Fat Tree Al-Fares et al. (2008); a well-known example that employs a three-tier structure and can interconnect are large number of servers using simple commodity hardware. Fig. 7 shows a $k = 4$ Fat Tree topology. A Fat Tree topology has three layers of switches: Top-of-the-Rack, aggregation and core switches. Hosts are connected to Top-of-Rack (ToR) switches, which in turn are connected to aggregation switches. ToR and aggregation switches are organized in pods. The Fat Tree in Fig. 7 has four pods which are interconnected by four core switches. A Fat Tree can support $\frac{k^3}{4}$ server nodes with $\frac{5k^2}{4}$ k-port switches. For instance, a Fat Tree constructed from 8-port switches can support 128 hosts with 80 switches. Clos networks provide a rich, high-capacity

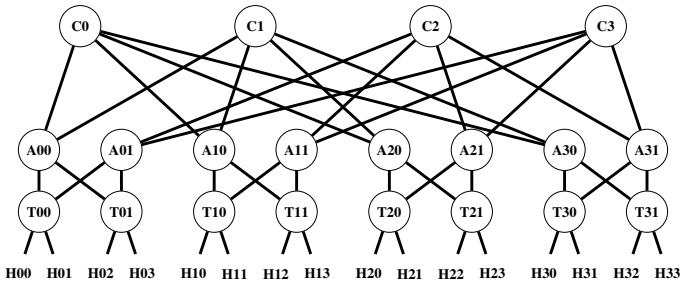


Figure 7: Fat Tree topology - A scalable three-tier Clos network. Data center networks are the main focus of this paper to start with the study of the question: can we completely learn a routing protocol?

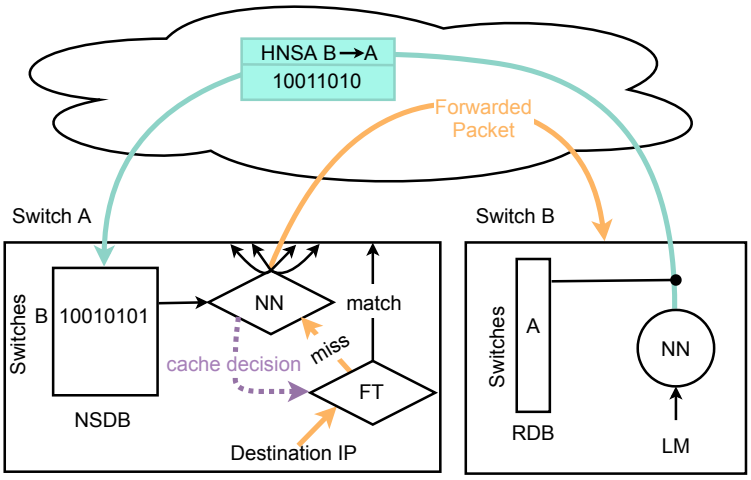


Figure 8: Realization of `Mistill` on a switch.

connectivity matrix, i.e., $\frac{k^2}{4}$ paths exist between any pair of servers. The Fat Tree in Fig. 7 already provides 4 paths between every server pair.

Clos networks have two advantages: Simplified inventory Management and resilience to failures Al-Fares et al. (2008); Dutt. Inventory management is simplified because the network can be build from fixed-form switches. This makes it easy to stock spare devices, and replace failed ones, since each switch is the same. Resilience of the topology is increased through high path diversity. If a link or device fails, enough alternative routes exist.

C Deploying Distilled Protocols

C.1 In-Network deployment

To be operable, `Mistill` needs routines and data structures on switches that interface with the NN. Each switch has four entities: A Flow(let) Table (FT), the Network State Database (NSDB), the NN, and a Receiver Database (RDB). Fig. 8 shows the entities and their interaction.

The NSDB stores `HNSA`s received from other switches in the network. The switch populates the NSDB with the content of `HNSA`s . Before storing the `HNSA`s in the NSDB, the switch decompresses the `HNSA` content. That is, the switch restores the previous one-hot encoding.

The NN is used in two functions on the switch: 1) to compute forwarding decisions, and 2) to compute the `HNSA` content. To compute forwarding decisions, the switch needs information from the packet. In our examples, this amounts to the source and destination IP address. All other inputs

reside in the memory of the switch. To compute the HNSA content, the switch monitors the required statistics, e.g., the link utilization or queue sizes. The switch executes the HNSAMOD with the current link state and compresses the one-hot encoding. The resulting HNSA is propagated through the network to other parts of the network as indicated by the RDB.

The FT caches forwarding decisions to avoid packet-reordering at the destination. We believe that it is likely that the content of the NSDB is updated frequently during the life-time of longer flows. As a result, it is likely that packets of the same flow are forwarded along different paths in the network. Thus, the switch caches the forward decision the NN produces for the first packet of a flow(let) and re-uses this decision for subsequent packets. This approach additionally decreases the computational effort. Caching of forwarding decisions to avoid packet re-ordering is also performed by other in-dataplane TE solutions **??**Hsu et al. (2020).

The RDB controls to which nodes in the network HNSAs are send. The RDB is lazily populated based on the source IP addresses of packets the switch observes during operation. The next section explains the sending of HNSAs in detail.

C.2 EDGE-BASED DEPLOYMENT

Put the NSMOD on the switches and the forwarding decisions are made on the hosts. Needs the sequential execution of the NN per flow. Could be realized in the network then with MPLS for example.

Problem: NN is too large for existing P4 enabled switches. Would need accelerating hardware that does not exist.

Opportunity: The computation of the HNSAs turned out to be quite cheap. Only a two layered NN is needed here, which could potentially be further optimized for size or realized as binary neural network. This one could be a realistic one to execute in the data plane with P4. Thus, split the tasks. The network devices execute only the HNSA Module. The path is calculated on the end hosts, for which accelerating hardware is more readily available in the form of Smart NICs, and where with eBPF a ready to use solution exists that can easily be utilized.

New Problem: We have $\frac{k^2}{4}$ paths in the network, not counting potential detours and so on in case of failures. If we want to predict the path, then the output layer quickly becomes very large, i.e., grows quadratic with k.

Solution: Chain multiple calls of the NN, which mimicks a walk of the packet through the network. The output at each stage can be used to select the next hop. The end-hosts need topology information for this. The end hosts have to queyr the net up to 4 times. IT is not possible to make detours and react to failures on a switch for packets in flight. The end-host can route later packets around this.

Opportunity: This is a form of source routing that could be mapped to MPLS and thus even work with legacy equipment. Here, the HNSAs would have to be calculated in the control plane of the switches, though.

C.3 HNSA PROPAGATION

We design an efficient propagation mechanism based on multi-casting that is inspired from the pattern in the SI and takes advantage of the structure of Clos-topologies. The propagation mechanism keeps the number of HNSAs that circulate in the network in the order of $\mathcal{O}(N)$, where N is the number of switches in the network.

The propagation mechanism exploits two observations: 1) up-stream switches need update messages from down-stream switches to make forwarding decisions. 2) Switches in one pod need update messages from a similar set of down-stream switches, since each switch is likely to see a similar set of destination IPs. Thus, we propose to multi-cast a HNSA to all switches in the source pod. HNSAs have a header with two fields 1) the IP of the switch from which the HNSA originates and 2) a bit array of length k, where k is the number of pods. Each switch tracks the pods from which traffic passing them originates, i.e., the RDB. The source pod of a packet can be deduced from the source IP-address Al-Fares et al. (2008). The switch indicates those pods in a bit array stored in the HNSA header.

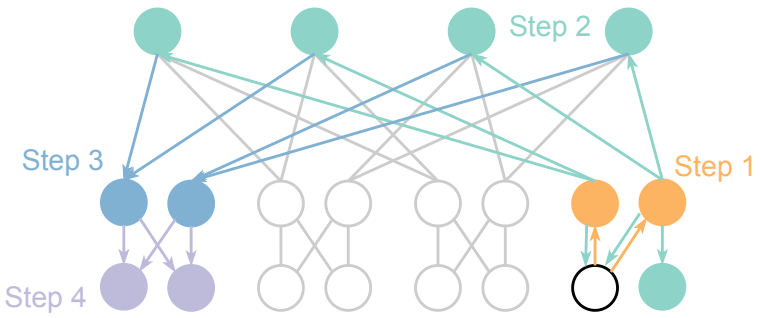


Figure 9: Probing scheme for all node types.

Fig. 9 shows an exemplary HNSA propagation from a ToR. First, the ToR sends its HNSAs to all neighboring aggregation switches (Step 1). The aggregation switches forward the probes to all ToR switches in the pod as well as all adjacent cores switches (Step 2). The ToR switches consume the probes and do not circulate them further. The core switches forward the HNSA to all adjacent aggregation switches in pods indicated by the bit array in the HNSA header (Step 3). The aggregation switches then forward the HNSA to *only* the incident ToR switches, i.e., do not send it back up into the core layer (Step 4). For this, the aggregation switches check for each HNSA that they receive whether the HNSA originated in their own pod. If it originated in a different pod, then the aggregation switch does not forward the probe to the core layer. The ToR switches do not further circulate the HNSA .

C.4 Edge-based deployment

Put the NSMod on the switches and the forwarding decisions are made on the hosts. Needs the sequential execution of the NN per flow. Could be realized in the network then with MPLS for example.

Problem: NN is too large for existing P4 enabled switches. Would need accelerating hardware that does not exist.

Opportunity: The computation of the HNSAs turned out to be quite cheap. Only a two layered NN is needed here, which could potentially be further optimized for size or realized as binary neural network. This one could be a realistic one to execute in the data plane with P4. Thus, split the tasks. The network devices execute only the HNSA Module. The path is calculated on the end hosts, for which accelerating hardware is more readily available in the form of Smart NICs, and where with eBPF a ready to use solution exists that can easily be utilized.

New Problem: We have $\frac{k^2}{4}$ paths in the network, not counting potential detours and so on in case of failures. If we want to predict the path, then the output layer quickly becomes very large, i.e., grows quadratic with k.

Solution: Chain multiple calls of the NN, which mimicks a walk of the packet through the network. The output at each stage can be used to select the next hop. The end-hosts need topology information for this. The end hosts have to queyr the net up to 4 times. IT is not possible to make detours and react to failures on a switch for packets in flight. The end-host can route later packets around this.

Opportunity: This is a form of source routing that could be mapped to MPLS and thus even work with legacy equipment. Here, the HNSAs would have to be calculated in the control plane of the switches, though.

D Node addressing schemes

Addresses in communication networks should follow a pattern to, e.g., keep routing tables small Waldvogel et al. (1997). If IP addresses did not contain a useful pattern, routing tables would

have to store forwarding decisions for each and every IP address. Since the addresses are written to the header of every packet, they should also be concise to reduce overhead.

IP addresses today are geared towards the available technology, e.g., towards longest prefix matching. It is unclear if the way nodes are addressed today is a good fit for routing protocols learned with ML.

Thus in this section, we investigate whether ML based routing protocols can be learned with IP addresses, whether latent space models, as used in natural language processing or graph learning, yield an addressing scheme that results in better performance, and whether the addressing scheme plays a role at all for the ability to learn a routing scheme.

D.1 Learning binary embeddings of nodes in graphs

Learning useful representations for nodes in graphs is subject to ongoing research Hamilton et al. (2017). We use Bernoulli Embeddings Misra & Bhatia (2018) to learn node addresses. Bernoulli Embeddings map nodes in a graph into a binary space that encodes the neighborhood relationship. We do not consider approaches such as `node2vec` Grover & Leskovec (2016) that embed nodes into a continuous latent space as the size the necessary floating point numbers is prohibitive.

A binary embedding is a d-dimensional binary vector, i.e., the embedding of node i is $E_i \in \{0,1\}^d$. The binary vector is sampled from d independent Bernoulli distributions, parameterized by a matrix $\Theta \in \mathbb{R}^{|\mathcal{V}| \times d}$. The kth element in the embedding of node i is thus sampled according to: $E_i^k \sim \text{Bernoulli}(\Theta_{\text{i,k}})$. The embedding can be optimized by changing the parameterization of the independent Bernoulli distributions such that an appropriate loss function is minimized.

The intuition behind the method in Misra & Bhatia (2018) is to encode neighborhood relation into the embeddings by casting the problem as a link prediction problem: The probability of an edge e_{ij} from node i to node j is proportional to the hamming distance d_H between the embeddings E_i and E_j of those nodes:

$$p(e_{ij} \mid E_i, E_j) = \frac{\exp\left(-\alpha d_H(E_i, E_j)\right)}{\sum_{k \in \mathcal{V}} \exp\left(-\alpha d_H(E_i, E_k)\right)} \tag{1}$$

The parameter α is a scaling factor. The hamming distance d_H between two vectors is defined as: $d_H : \{0,1\}^d \times \{0,1\}^d \to N; x, y \to x^T(1-y) + (1-x)^T y$.

The embeddings can then be trained by optimizing the log probability, i.e.:

$$\mathcal{L}(G, \Theta) = -\sum_{(i,j) \in \mathcal{E}} \mathbb{E}\left[\log p(e_{ij} \mid E_i, E_j)\right]_{\Theta}, \tag{2}$$

assuming that edges are independent of each other given the embeddings of incident nodes.

Misra et al. Misra & Bhatia (2018) use a continuous approximation to the objective in Eq. (2) to circumvent the expectation over a discrete valued variable. In contrast, we make use of the Gumbel-Softmax trick Jang et al. (2017); Maddison et al. (2017), a reparameterization trick for discrete random variables, to optimize the objective in Eq. (2). In addition, Misra et al. Misra & Bhatia (2018) use Noise Contrastive Estimation (NCE) to avoid the summation over all nodes in the graph in the denominator of Eq. (1). We do not use NCE since optimization is feasible for the size of communication networks. In addition, we experienced that NCE makes learning of unique embeddings more difficult due to aliasing effects between nodes.

Addresses in communication networks must be unique which translates to the embeddings of the nodes. The optimization objective in Eq. (2) is not designed for this constraint and might assign the same embedding to different nodes. In fact, for a fully connected graph a single embedding for all nodes would in fact be the optimal solution. We thus adapt the objective in order to avoid duplicate embeddings. Specifically, we use the following energy function for the model of edge probabilities:

$$\mathcal{L}(G, \Theta) = -\alpha \sqrt{\left(d_H(E_i, E_j) - \beta\right)^2}, \tag{3}$$

where β is a scalar parameter. The intuition behind Eq. (3) is that a probability of one for an edge is obtained only, if the embeddings are different. The parameter β controls the scaling of the difference. For example, $\beta = 1$ pushes the embeddings of adjacent nodes towards having a hamming distance of one.

Optimal	Random	Learned12	Learned16	Learned20	Learned24	Composed	IP
0.3707	0.6210	0.5397	0.4835	0.5002	0.6541	0.4185	0.3921

Table 2: Training results on the shortest path training tasks for graphs and embeddings. For the learned embeddings, different vector sizes were evaluated. Optimal is the average loss value for the ground truth.

D.2 RESULTS

The suitability of addressing schemes is evaluated by learning static shortest paths. An addressing scheme is suitable if the forwarding rules for static ECMP can be learned reliably, i.e., if we obtain a low loss value. Tbl. 2 lists the results obtained for this learning task. We use addresses of up to 24 bits. IP addresses follow the scheme in Al-Fares et al. (2008). We remove the first octet since it is constant and thus does not contribute any information.

The random embedding results in the highest, i.e., worst, loss value. The addresses do not contain any form of pattern, after all. The learned embeddings result in lower loss values. The dimensionality of the embedding does not have an effect on performance. No consistent increasing or decreasing trend is observable. The IP address based embedding results in the lowest loss values. Good performance here is explainable, since the forwarding behavior can be perfectly computed from the IP address alone in case of the Fat Tree. The composed embedding for the Fat Tree achieves a loss value of 0.4185 and is competitive with the IP-based result of 0.3921. This hints at filtering out similar structures in the graph and assigning embeddings to those structures in an equal manner.

In summary, IP addresses are well suited to learn forwarding decisions on well structured topologies such as Fat Trees. This has the advantage that learned routing protocols are backwards compatible, can be incrementally deployed, or run side-by-side with traditional routing protocols. For `Mistill`, we thus use the last 24 bits of the IP addresses to represent nodes.