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 for how traffic in the data center network is routed. 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 Traffic Engineering (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)).

The design of a new TE scheme is a time and work-intensive process. Obtaining a deployable distributed protocol from an initial 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 (Alizadeh et al. (2014); Katta et al. (2016); Benson et al. (2011)). The protocol must meet line-rate, exchange information at (sub)millisecond granularity, and cope with limited computational requirements (Alizadeh et al. (2014); Katta et al. (2016); Xiong & Zilberman (2019)). 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 three 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. 3) `Mistill` provides one framework that can realize multiple TE designs in an automated fashion.

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. We find that `Mistill` can learn distributed

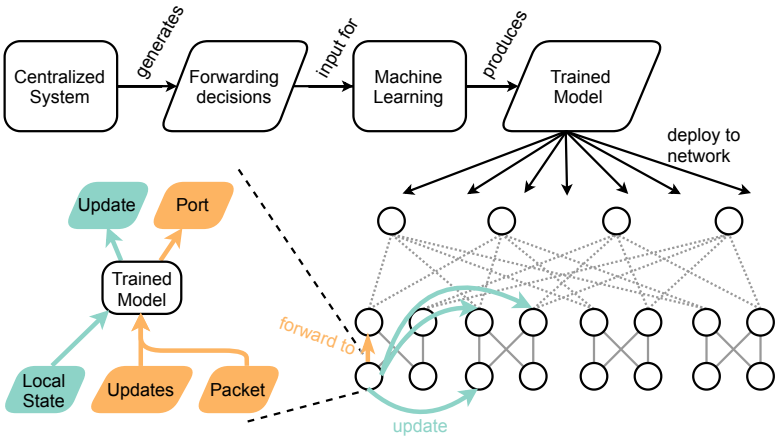


Figure 1: A high-level overview of `Mistill`. First, `Mistill` trains a NN with forwarding decisions. `Mistill` uses forwarding decisions from an in initial prototype of the TE scheme. Then, `Mistill` deploys the trained NN to the switches, which query the NN for forwarding decisions.

protocols that closely implement the desired policy. Further, we find that the content of the messages and their exchange is reasonable.

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 behind `Mistill` and the expected outcome of `Mistill`. As Fig. 1 shows, `Mistill` starts with a centralized system that implements a TE policy. For example, this system can be a packet or flow level simulation. Here, 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.

Next, `Mistill` 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 two aspects: first, how to process and exchange the local state of switches. Second, how to map the exchanged state into forwarding decisions.

After training, `Mistill` deploys the model onto the switches in the network, i.e., all switches in the network have *the same* model. In order to make correct forwarding decisions, `Mistill` conditions 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 neural network can be implemented on the switches with e.g., P4 or SmartNICs (Sanvito et al. (2018); Siracusano & Bifulco (2018)).

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. During training, the model must learn how to react to changes in the network, such as node and link failures, and changes in monitored measures e.g., the link utilization.

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). The

network protocols must meet three requirements. They must react within milliseconds; they must be data efficient and computational tractable (Alizadeh et al. (2014); Hsu et al. (2020); Katta et al. (2016)).

The learned network protocols must react to changes in the network within milliseconds, e.g., changes in link utilization (Alizadeh et al. (2014); Katta et al. (2016)). 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. (Hsu et al. (2020); 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); Katta et al. (2016)). 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)), and thus able to learn complex functions. This is important because the forwarding decisions of a TE policy might 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 link utilization can lead to a sudden change in the forwarding policy.

2) NNs have the potential to be executed with the limited computational resources in the data-plane. Here, the choice of activation function is important. Activations like the $\mathrm{sigmoid}$ or $\tanh$ require expensive floating point evaluations, that are not suitable for data-plane deployment (Glorot et al. (2011)). In contrast, 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 only computation (Gholami et al. (2021)), or even to binary networks.

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)), albeit with significant effort. 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, also driven by the increasing trend to offload NN inference in image and speech recognition to the network.

3 NEURAL ARCHITECTURE

`Mistill` uses a NN to learn a distributed protocol for a TE policy. The NN maps the local state of switches into update messages. Further, the NN learns how switches rely on other switches states and maps the exchanged state into forwarding decisions. 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.

Output. The NN produces 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 in-

[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.

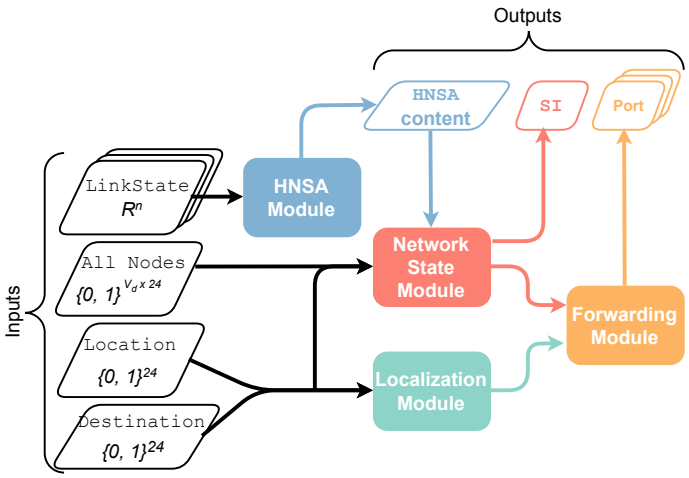


Figure 2: The NN architecture consists of four modules: The LocMod, the HNSAMod, the NSMod and the Fwd Mod.

cident 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, i.e., the forwarding behavior will change based on the value of `Location`. `Destination` represents the destination node of a packet. Thus, the output of the NN does also depend on the destination node. 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.

Training. The NN is trained end-to-end. Training end-to-end tailors `HNSA`s and SI towards the behavior of a specific forwarding policy. The NN propagates its error in predicting the forwarding decisions back through all modules. Thus, the NN adjusts the `HNSA`s and SI to reduce its error.

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

The `Link State` $\in \mathbb{R}^6_{\geq 0}$ is a real-valued vector and represents the incident edges. The state represents each link with a vector from $\mathbb{R}^6_{\geq 0}$. The first two attributes are the one-hot encoded availability of the link in the outgoing direction. The next two attributes are the one-hot encoded availability in the incoming 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. The neurons in the hidden layer are either one or zero. 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:

$$\texttt{HNSA} \sim \mathrm{Cat}(n, \theta_1) \mid\mid \ldots \mid\mid \mathrm{Cat}(n, \theta_m). \tag{1}$$

The NN calculates the parameters θ_i from the `Link State` input. The NN passes the `Link State` through a feed forward NN with one hidden layer and ReLU activation. The output layer corresponds to the θ_i. 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.

Since the categorical samples are one-hot encoded, the size of the `HNSA` content can be further reduced. The switch can represent each block with $m\lceil\log_2(n)\rceil$ bits, i.e., the switch sends the position of the one in the block.

3.3 LEARNING TO COMMUNICATE

The NSMOD learns two things: 1) whose `HNSA`s are needed to make a forwarding decision on a specific switch for a given destination, and 2) how to process those `HNSA`s to make the necessary forwarding decision. The learning objective of the first task corresponds to the SI output. The SI determines how `HNSA`s are exchanged between switches, i.e., which switches rely on the `HNSA`s of another switch.

To learn the SI, we use scaled dot-product attention (Vaswani et al. (2017)). In `Mistill`, attention 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. The Keys and Queries are used to compute attention-weights A, with which a convex combination of the Values is computed. K, Q and V are matrices, where K and V must have the same number of rows (Vaswani et al. (2017)). The attention weights A are then calculated as:

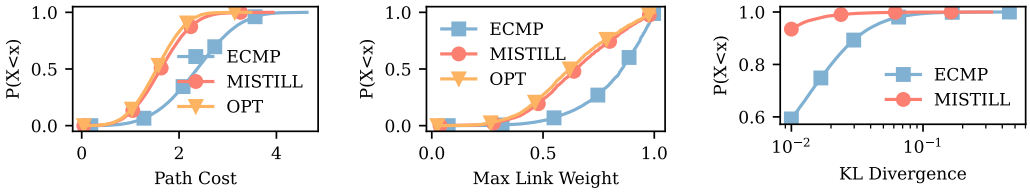
$$A = \mathrm{gumbelSoftmax}\left(\frac{1}{\sqrt{r}} W_q Q \cdot (W_k K)^T\right), \tag{2}$$

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 then:

$$A \cdot (W_v V), \tag{3}$$

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. Thus, each head can concentrate on separate parts of the input Vaswani et al. (2017). In our case, each head focuses on individual nodes. The output of the attention heads is concatenated and passed through a hidden layer with ReLU activation. This output is passed as input into the FWD MOD.

The SI is obtained from the attention weights of the individual attention heads. If an attention weight is zero, the corresponding `HNSA` is zeroed out and plays no further role in making forwarding decisions. Non-zero entries on the other hand signal, that the corresponding `HNSA` contributes



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

Figure 3: CDFs of performance metrics for `LCP` and `MinMax`.

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 in the network. 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 `HNSAs` , 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

In our 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 (Katta et al. (2016); Dutt). To forward traffic in the case a route exists, we consider three policies: 1) `WCMP`: Splits traffic over neighbors in proportion to the available down-stream bandwidth, 2) `LCP`: forwards traffic 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

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 consists of $50\,000$ problem instances. We use the validation set for parameter tuning. The final results are obtained from a packet level simulation.

(a) Training Progress.

(b) Performance.

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

4.2 Routing Performance

We perform 10 simulations with random node and link failures and random edge weights for each policy, which are not part of the training or validation set. In each simulation, one host communicates to all other hosts in the network.

LCP: Fig. 3a shows a CDF of the cost of paths that Mistill and ECMP choose, and the actual shortest paths (shown as OPT) accumulated for all 10 runs. The cost of a path is the sum of the weights of the edges in the path. Fig. 3a shows that OPT always has the lowest cost — the CDF is the most left one. The costs of paths for Mistill closely match the distributions of the shortest paths. The paths that ECMP selects have a higher cost — the CDF is the most right one. The behavior of the routing protocol learned with Mistill closely matches the LCP policy.

MinMax: Fig. 3b compares the maximum edge weight of edges from the paths chosen with Mistill and ECMP against the actual minimum maximum link weight (OPT). Fig. 3b shows the routing protocol obtained with Mistill closely matches the behavior of the actual routing policy. The CDFs for Mistill and OPT almost overlap. In contrast, ECMP selects paths with higher maximum edge weights.

WCMP: We use the Kullback-Leibler Divergence (KLD) to evaluate WCMP. We interpret the splitting ratios on each node as a discrete probability distribution. We compare the KLD between the splitting ratios of Mistill and ECMP with the true distribution. Fig. 3c shows the CDFs of the KLD for 10 runs. Fig. 3c shows that the learned protocol has smaller KLDs than ECMP. Almost all KLD values of the learned protocol are smaller than 0.02. For ECMP, about $76\,\%$ of the values are smaller than 0.02. The average KLD of Mistill is 0.0453 and of ECMP 0.2608.

In summary, our experiments show that Mistill can learn the forwarding behavior of three popular TE policies from exemplary samples.

4.3 HNSA Size

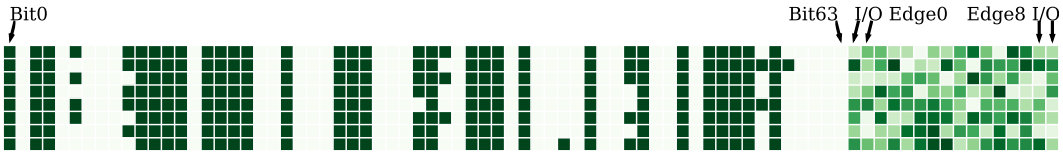
We investigate the impact of the number of blocks on the MinMax policy. Fig. 4a shows that an increasing number of blocks results in a smaller loss. Fig. 4a shows that the decrease in loss is strong during the initial stages of training and converges towards the end.

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

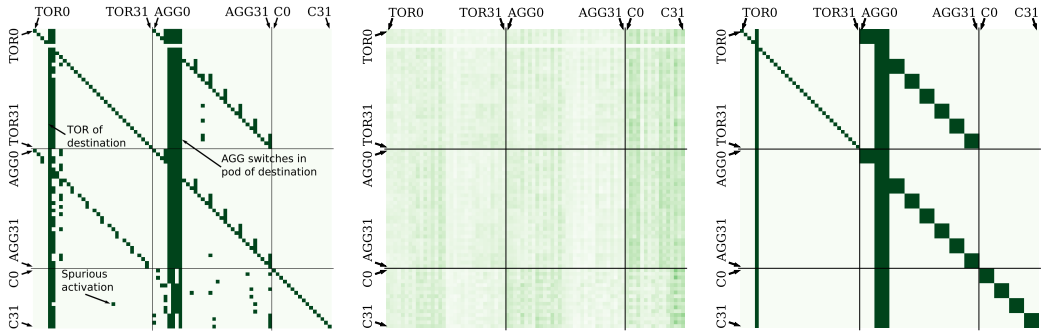
This section visualizes the learned HNSA messages and the attention weights. We compare the attention weights of the gumbelSoftmax activation with the weights of a model trained with the sparsemax activation for the MinMax policy.

(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.

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.

Zhaoqi Xiong and Noa Zilberman. Do switches dream of machine learning? toward in-network classification. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, pp. 25–33, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370202. doi: 10.1145/3365609.3365864. URL https://doi.org/10.1145/3365609.3365864.

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 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.

A.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:

$$p(e_{ij} \mid E_i, E_j) = \frac{\exp\left(-\alpha d_H(E_i, E_j)\right)}{\sum_{k \in \mathcal{V}} \exp\left(-\alpha d_H(E_i, E_k)\right)} \tag{4}$$

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.:

$$\mathcal{L}(G, \Theta) = -\sum_{(i,j) \in \mathcal{E}} \mathbb{E}\left[\log p(e_{ij} \mid E_i, E_j)\right]_\Theta, \tag{5}$$

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. (5) 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. (5). 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. (4). 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. (5) 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:

$$\mathcal{L}(G, \Theta) = -\alpha \sqrt{\left(d_H(E_i, E_j) - \beta\right)^2}, \tag{6}$$

where β is a scalar parameter. The intuition behind Eq. (6) 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.

A.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. 1 lists the results obtained for this learning task. We use addresses of up to 24

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 1: 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.

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.