Towards Integrating Formal Methods into ML-Based Systems for Networking

Fengchen Gong  
Princeton University  
Divya Raghunathan  
Princeton University  
Aarti Gupta  
Princeton University  
Maria Apostolaki  
Princeton University

Abstract

Owing to its adaptability and scalability, Machine Learning (ML) has gained significant momentum in the networking community. Yet, ML models can still produce outputs that contradict knowledge, i.e., established networking rules and principles. On the other hand, Formal Methods (FM) use rigorous mathematical reasoning based on knowledge, but suffer from the lack of scalability. To capitalize on the complementary strengths of both approaches, we advocate for the integration of knowledge-based FM into ML-based systems for networking problems. Through a case study, we demonstrate the benefits and limitations of using ML models or FM alone. We find that incorporating FM in the training and inference of an ML model yields not only more reliable results but also better performance in various downstream tasks. We hope that our paper inspires a tighter integration of FM-based and ML-based approaches in networking, facilitating the development of more robust and dependable systems.

CCS Concepts

• Networks → Network monitoring.

Keywords

Telemetry, Imputation, Formal Methods, Transformer

1 Introduction

Advances in Machine Learning (ML) have been disruptive in multiple domains, spanning natural language processing, computer vision, and recommendation systems. Networking has also become a playground for ML, thanks to its multiple intricate and largely unresolved problems, complex patterns, and abundance of data. Consequently, ML-based algorithms have been proposed for handling a wide range of long-standing networking challenges, including congestion control [19, 31], anomaly detection [4, 49], synthetic data generation [41], and performance forecasting [43].

While scalable, adaptable, and often more efficient than traditional solutions, ML-based algorithms lack correctness guarantees and are susceptible to overfitting patterns or settings. Thus, ML models could produce results that contradict domain knowledge, meaning that they are inferior to primitive algorithms, defy physical laws, or contradict common sense [1, 21]. For instance, in predicting delay, certain outcomes are implausible, such as those exceeding the speed of light. Similarly, in synthetic traffic-trace generation, the traffic rate originating from a port could not surpass its capacity.

It is evident that knowledge represented as mathematical equations, physical laws, probabilistic relationships, or logic rules should be somehow incorporated into solutions. Formal Methods (FM) is, of course, the time-honored way of leveraging knowledge in generating correct and sound results. As a result, FM has also seen significant success in networking problems such as congestion control [9], configuration synthesis [20, 50], traffic engineering [27], and finding bugs [32, 34, 37]. Unfortunately, FM-based networking solutions have difficulty in scaling and in learning patterns.

To get the best of both worlds, this paper advocates for the integration of FM into ML-based networking to enhance its dependability. Doing so could foster more reliable models with soundness guarantees, which could be trained with less data, and generalize better. While FM has been proposed as a mechanism to enhance confidence in trained models based on Deep Reinforcement Learning (DRL) [21, 57] and domain knowledge drives the selection of the model and/or the features in ML-based networking [57, 58], we take one step further and argue for knowledge-augmented training and inference. In fact, knowledge-augmented models are already proposed in physics and biology with Physics- [35] (or Biology- [38]) Informed Neural Networks.

To investigate the potential of the integration of FM into ML-based networking, we focus on a problem of network telemetry, but our methodology can translate to other networking problems. In particular, we impute fine-grained queue-length time series in switches by jointly analyzing their coarse-grained counterparts, together with other time series such as packet and drop counts. This is a suitable use case. First, fine-grained queue monitoring is crucial for multiple downstream tasks such as anomaly detection, provisioning, and root-cause analysis. Second, fine-grained queue monitoring is challenging due...
to hardware limitations and scale [13, 24, 60]. Finally, queue lengths are affected by traffic patterns and follow well-studied principles.

At first glance, this problem could be solved by either ML or FM. On the one hand, imputing time series can be seen as an analogy to image superresolution (i.e., turning a low-resolution image to a high-resolution one), which is often solved using generative models in ML [18, 40, 53, 54, 59, 62]. However, we find that using ML alone yields results that are often evidently inaccurate, i.e., inconsistent with measurements or against known principles, especially for uncommon incidents. On the other hand, given a set of measurements (coarse-grained time series), and a set of constraints connecting them to their fine-grained counterparts and to each other, an FM model could, in principle, compute a fine-grained time series. Unfortunately, we find that such a solution is hard to scale, due to the large search space.

While seemingly straightforward, combining FM and ML for queue length imputation poses multiple challenges which also generalize to other networking problems. First, there is no standard way to incorporate domain knowledge into ML models. Second, incorporating knowledge can significantly increase the complexity of the learning process, which might cause scalability issues for complex models. Third, designed to learn from data, ML models cannot easily ingest traditional knowledge such as rules and relationships.

To address these challenges, we start from a pure ML-based approach and strategically incorporate some of the constraints of our FM model in its training and inference. The ML-based approach (a transformer) ingests multiple sampled (coarse-grained) switch-level time series and outputs fine-grained queue lengths. Specifically, we first select constraints that can be directly evaluated on the transformer output. Doing so maintains system scalability because the system does not need to reason about detailed (per-packet) scenarios. Next, we transform those constraints to a differentiable form such that they can be incorporated into the loss function of the transformer. Finally, we enforce the constraints that the transformer failed to satisfy by correcting its output post-imputation. We show that combining ML with FM effectively increases queue-length monitoring granularity by 50x (from 50ms to 1ms) and yields 11-96% better results compared to ML alone.

We posit that knowledge-augmented ML-based models for networking instigate a vibrant research trajectory. This trajectory includes questions such as: Which networking problems require both ML and FM to be solved? How do we represent decades of accumulative knowledge on network calculus, network tomography, and optimizations in an ML-friendly way? How to use knowledge to fight the scarcity or bias of datasets? How can we verify that an ML system has indeed learned networking principles?

To demonstrate the potential of the integration of ML-based networking with FM, we consider the task of fine-grained monitoring of queue lengths in network switches. Queue monitoring is a particularly useful but challenging task. Queue lengths affect latency guarantees [63] and expensive on-chip buffer provisioning [56]. However, queue lengths can change at the microsecond granularity making queueing the most unpredictable part of a packet’s journey [44] and monitoring them particularly hard [13]. Instead of investigating hardware upgrades and/or telemetry operators, we work on a purely software approach. Concretely, we seek to use existing coarse-grained time series of various signals to impute fine-grained queue lengths.

2 Case study

Consider an operator of a large datacenter who has to run a set of downstream tasks, e.g., deciding how much on-chip buffer to provision to network switches, or detecting adversarial traffic patterns. To inform these tasks, ideally, the operator needs to have fine-grained (say microsecond level) measurements of queue length of each switch over time. Indeed, longitudinal analyses of fine-grained queue length measurements will give the operator an idea of the common burst sizes and frequencies to inform the trade-off between accommodating bursts and reducing switch cost [55].

Let us assume, without loss of generality, that the datacenter has output-queued switches with N ports, each with two queues and a shared buffer across all queues, as shown in Fig. 2. In an ideal case, the operator would collect fine-grained time series of the length for each queue, \( Q = \{Q_1, \ldots, Q_{2N}\} \) and use that for the downstream tasks. In practice, the operator has access to monitoring only at a much coarser granularity, say, every 50ms. Consider, for instance, that the operator can use (i) periodic sampling of each queue, which provides instantaneous queue lengths; (ii) LANZ [8], which provides the per-queue maximum length within each interval\(^1\), but does not specify exactly when the maximum occurred; and (iii) SNMP [22],

\(^1\)In practice, LANZ will only monitor a queue after its length exceeds a configurable threshold, but for simplicity, we assume that the operator configured this low enough s.t. LANZ reports some value for each interval unless the queue is constantly zero.
which provides per-port counts of packets sent, and dropped every interval. As an illustration, Fig. 1 shows a queue length time series at fine granularity (continuous line) and the measurements available to the operator (dots). Arguably, sampling hides critical details.

**Insights:** While each routinely collected coarse-grained time series (i.e., queue lengths, packet counts, and drops) alone hides important incidents, we posit that we can discover more about the network by analyzing them together. In fact, all these series follow the operating principles of a switch and are affected by common traffic, thus are correlated. First, as the buffer is shared across multiple queues, their lengths are correlated: a longer queue prevents other queues from growing by taking up space in the buffer [2, 3, 6, 14]. Second, loss rate and queue lengths are correlated: loss only occurs when queues are longer than a threshold. Finally, queue lengths are correlated with the incoming packet rate: a queue only forms in a many-to-one (fan-in) scenario, i.e., when the service rate is lower than the incoming rate. As shown in Fig. 1, an increase in the queue length is accompanied by an increase in the coarse-grained packets sent and dropped in the same interval.

These insights beg the question: Can we use the available coarse-grained monitoring to impute fine-grained queue lengths? One strategy is to train an ML model (e.g., a transformer) to predict fine-grained queue lengths given the coarse-grained time series (§2.2). Another strategy is to model a switch using Satisfiability Modulo Theories (SMT) [10] constraints and use an SMT solver [15] to find a solution to the system of constraints, e.g., when the input data is predominately skewed towards smaller values. To make matters worse, we found that our model also violated switch-specific constraints. For example, the total number of packets that would need to have been dequeued for the imputed queue to be formed exceeded the SNMP count.

2.2 Telemetry Imputation with ML

Next, we explain why using ML is a natural choice and what challenges we encountered in the process.

**Why ML is a good idea:** ML techniques are good at handling complex and non-linear relationships between variables by learning directly from data. This allows us to leverage correlations that are too complex to model or for which we do not have perfect knowledge. For example, for predicting queue length, one would need to model congestion control, dynamic buffer sharing, scheduling, and even traffic patterns and demands. Moreover, ML is scalable: models can be parallelized in the training and inference phases.

We find transformers to be particularly suitable models for telemetry imputation. A transformer is a sequence-to-sequence model that is able to learn correlations over a long sequence in parallel based on the attention mechanism [52]. Its flexibility and efficiency have already made it popular in the networking context [16, 29, 39, 58].

**Why ML is not a good idea:** The most significant downside of using ML for telemetry imputation is that the output evidently lacks correctness. As an illustration, Fig. 4b shows the output of a transformer we trained to impute queue lengths from coarse-grained time series (§4 provides details for the model and data generation). We observe that the imputed time series (blue line) is not consistent with the measurements. For example, the transformer did not impute a queue length that is as high as the (known) max queue length (red dot) of the interval between 50-100ms, although it was part of the transformer’s input. It may seem surprising that the model does not “realize” the connection between the provided max queue lengths and the ground-truth (fine-grained) queue lengths. However, this issue arises due to the inherent challenge of predicting large values when the input data is predominately skewed towards smaller values. To make matters worse, we found that our model also violated switch-specific constraints. For example, the total number of packets that would need to have been dequeued for the imputed queue to be formed exceeded the SNMP count.

2.3 Telemetry Imputation with FM

The inability of our ML model to produce a fine-grained time series consistent with our knowledge, i.e., the coarse-grained measurements and domain-specific rules, motivates the use of FM for telemetry imputation.

**Why formal methods is a good idea:** FM allows us to express our knowledge about how the switch operates and use automated reasoning to find a scenario that fits the coarse-grained observations. Importantly, FM provides a guarantee that a result is plausible, i.e., could have occurred in a switch given the observed measurements and domain-specific constraints.

Inspired by prior work [7] on using FM to analyze performance in switches, we use Satisfiability Modulo Theories (SMT) constraints to model switch behavior at the level of a single packet. We divide time into discrete time steps, where a time step is the time taken to transmit or receive a packet. We then use Z3 [15] to find a solution to the system of constraints, which corresponds to a time series of queue lengths.

At a high level, we model two types of constraints: (i) operational, and (ii) measurement. The former describes the way a switch works. The latter incorporates the measurements, effectively demanding that the result of utilizing monitoring tools (e.g., max queue length, packet counts) on the fine-grained time series would result in the coarse-grained measurements. Due to space limitations, we have omitted the formal equations.

**Operational constraints.** Every packet that arrives at the switch is mapped to some output queue, and a packet is dequeued if the scheduler selects some queue. If the queue was unbounded, the number of packets in the queue at time step _t_,
pkts_q,t is the sum of the queue length at t – 1 and the number of packets received at t. In practice, the queue is bounded, thus if pkts_q,t exceeds a dynamically calculated threshold (thr_q,t), the excess packets are dropped; the remaining packets are enqueued (pkts_q,t). The queue length at t is pkts_q,t minus 1 if a packet is dequeued. Additional constraints over the selected queue and dynamic threshold model the scheduling and buffer management algorithms, respectively.

**Measurement constraints.** The number of packets received, sent, and dropped at each port must equal the counts reported by SNMP. The maximum queue length during the monitoring interval must equal the maximum as reported by LANZ (m_max). If the queue length is sampled at time steps T_samples, the imputed queue length must match the sampled queue length (m_len_q,t).

**Why formal methods alone is not a good idea.** Although our FM approach guarantees the imputed result is plausible, its scalability is limited. Z3 successfully generated imputed queue lengths for simple scenarios in a few minutes, but could not handle more realistic scenarios in even 24 hours. Due to the high port bandwidth, there are ≈ 90 time steps in 1 ms, leading to a large search space of traffic scenarios which makes the problem intractable for the solver. For instance, if two packets are enqueued sequentially on an empty queue, different inter-arrival gaps are considered, though they have the same effect on the queue length.²

³ Combine ML and FM

Driven by our previous observations demonstrating the complementary nature of FM and ML, we design a synergistic approach shown end-to-end in Fig. 3. The raw (fine-grained) time series T_r is sampled to T_c (e.g., by generic monitoring tools) and fed to a transformer which is trained using a Knowledge-Augmented Loss (KAL) function. During inference, a Constraint Enforcement Module (CEM) corrects the transformer’s output (Qc) by minimally changing it until it satisfies certain constraints producing Q̂_c. Note that an operator would only have T_c available to infer Q̂_c. For training, she can use a simulation or a short real trace to generate T_r.

Exact modeling as defined by the operational and measurement constraints is too expensive, as we discussed in §2.3. We navigate the trade-off between accuracy and scalability by reducing our system of constraints to three that we can directly test against the transformer-imputed queue lengths.

The first two constraints are related to measurements, particularly max and periodic sampling of queues:

\[
\max_{0 \leq t < T} \hat{q}_t[q][t] = m_{\text{max}}_q \\
\forall t \in T_{\text{samples}}. \ Q_c[q][t] = m_{\text{len}}_{q,t}
\]

where \(Q_c[q][t]\) denotes the length of queue \(q\) at the \(t^{th}\) ms.

Finally, we observe that if some queue in port \(i\) is nonempty for \(NE_i\) time steps, then \(NE_i\) packets will be dequeued, as schedulers are typically work-conserving. An empty queue can send a packet if one arrives; hence \(NE_i\) is a lower bound on the number of packets sent, \(m_{\text{out}}_i\):

\[
NE_i \leq m_{\text{out}}_i
\]

where \(NE_i = \sum_{t=0}^{T-1} \text{ite}(\bigvee_{q \in \text{Queues}} \hat{q}_t[q][t] > 0, 1, 0)\)

\(\text{ite}(c, v_1, v_2)\) means if \(c\) is true, then return \(v_1\), else return \(v_2\).

The resulting set defines an over-approximation of the switch behavior: every plausible imputed result satisfies these constraints, but not every imputed result consistent with them is plausible. We integrate these constraints on the transformer’s training and inference. For the training, we modify the loss function, effectively teaching the transformer to obey domain knowledge (§3.1). For the inference, we correct the transformer’s output to be consistent with the constraints (§3.2).

### 3.1 Knowledge Augmented Loss (KAL)

Our transformer (§2.2) is trained with EMD (Earth Mover’s Distance) [47] which allows it to learn queue-length distributions but not to satisfy known constraints. Thus, we augment the loss function to incorporate certain constraints. Besides

²This example scenario is on-purpose simplified, there are other cases that cannot be easily distinguished or pruned.

³This example scenario is on-purpose simplified, there are other cases that cannot be easily distinguished or pruned.
EMD($T_r, \hat{Q}_r$), the transformer aims at satisfying both the equality constraints (C1, C2), and the inequality constraint (C3), presented by differentiable functions $\Phi(T_r, \hat{Q}_r)$ and $\Psi(T_r, \hat{Q}_r)$, respectively. To turn (C3) (which is non-differentiable as it involves an $ite$) into the differential $\Psi$, we (i) apply a Tanh function to each scaled queue length, resulting in 1 when the length is greater than 0, and 0 otherwise; and (ii) sum the results across all queues in the port to model the disjunction. The loss function is thus turned into a constrained optimization problem:

$$\text{min} \ EMD(T_r, \hat{Q}_r) \text{ s.t. } \Phi(T_r, \hat{Q}_r) = 0, \; \Psi(T_r, \hat{Q}_r) \leq 0$$

To have the model learn to satisfy the constraints, we adopt the augmented Lagrangian method, inspired by [17]. This involves introducing penalty terms into the objective function to represent constraint violations. These penalty terms are weighted by Lagrange multipliers $\lambda^q_i$ for equality constraint $\Phi$ evaluated at each training example $(T_{r_i}, \hat{Q}_{r_i})$, and similarly $\lambda^{ineq}_i$ for inequality constraint $\Psi$. At each training step, each Lagrange multiplier is updated by multiplying the violations of the corresponding output data by a parameter $\mu$. The importance of a violation in the loss function increases as its magnitude becomes higher, requiring more effective minimization. Finally, the loss function is given by

$$L = EMD(T_r, \hat{Q}_r) + \sum_{i=1}^{N} \mu \Phi(T_{r_i}, \hat{Q}_{r_i})^2 + \sum_{i=1}^{N} \lambda^q_i \Phi(T_{r_i}, \hat{Q}_{r_i}) + \sum_{i=1}^{N} \lambda^{ineq}_i \Psi(T_{r_i}, \hat{Q}_{r_i}) + \sum_{i=1}^{N} \mu \lambda^{ineq}_i \left[ \lambda^{ineq}_i > 0 \vee \Psi > 0 \right] \Psi(T_{r_i}, \hat{Q}_{r_i})^2$$

where $N$ is the training dataset size. Training with these penalty terms enables the model to learn the consistency between the input and output, enforced by the constraints. As shown in Fig. 4c, the imputed queue length of a transformer trained with KAL approaches the maximum value much more (compared to the pure transformer in Fig. 4b).

### 3.2 Constraint Enforcement Module (CEM)

While the incorporation of constraints in the loss function improves the imputation accuracy, it still provides no guarantee that the constraints will be satisfied. Thus, we introduce the Constraint Enforcement Module (CEM) which aims at correcting the output of the transformer using the SMT solver Z3 (i.e., forces it to satisfy the constraints we outlined before C1, C2, C3) while changing it as little as possible. We use variables $\hat{Q}_r^e[q][t]$ to denote the corrected queue lengths at each time step. To ensure that the corrected results remain close to the ML model’s output, we use the following objective that minimizes the total difference between the corrected and original values, ignoring the time steps in which the queue length is sampled.

$$\text{min} \sum_{t=0}^{T-1} \sum_{i \in T_{\text{samples}}} |\hat{Q}_r^e[q][t] - \hat{Q}_r[q][t]|$$

Fig. 4d showcases the output generated using both KAL and CEM. In this case, we observe that the imputed values precisely follow the maximum values and periodic samples.

Observe that CEM does not significantly deteriorate the scalability of the system. First, selected constraints do not require the solver to calculate the state for every time step as in §2.3. Second, the transformer output has already satisfied some of the constraints, thanks to KAL.

### 4 Preliminary evaluation

We compare various imputation methods in terms of consistency and performance when their output is used for various downstream tasks. We find that a transformer augmented with CEM and KAL yields the best results while being scalable.

**Imputation methods:** We use four methods: a non-ML one and three transformer-based. First, we use the IterativeImputer [48], a statistical method that retains the periodic samples, models the feature with missing values as a linear function of other features iteratively. To feed IterativeImputer with the maximum queue length, we place the max at the midpoint of each interval. Second, we use a transformer encoder that encodes the set of coarse-grained time series and a linear layer as decoder to generate the fine-grained time series. We use EMD as our loss function as opposed to MSE because it improves the accuracy of the model in locating bursts. While MSE is more commonly used, we found that it encourages the model to find averages of plausible solutions that are overly smooth and is disadvantageous for bursts. Finally, we augment the transformer with KAL and with CEM as we describe in §3.
Table 1: Downstream tasks’ performance is significantly better when the transformer is augmented with KAL and CEM.

<table>
<thead>
<tr>
<th>Error Metric</th>
<th>IterImputer</th>
<th>Transformer</th>
<th>Transformer + KAL</th>
<th>Transformer + KAL+CEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. Max Constraint</td>
<td>0.49</td>
<td>0.88</td>
<td>0.93</td>
<td>0</td>
</tr>
<tr>
<td>b. Periodic Constraint</td>
<td>0.078</td>
<td>0.16</td>
<td>0.15</td>
<td>0</td>
</tr>
<tr>
<td>c. Burst Frequency</td>
<td>0.43</td>
<td>0.52</td>
<td>0.028</td>
<td>0</td>
</tr>
<tr>
<td>d. Burst Detection</td>
<td>0.32</td>
<td>0.18</td>
<td>0.17</td>
<td>0.16</td>
</tr>
<tr>
<td>e. Burst Height</td>
<td>0.98</td>
<td>0.35</td>
<td>0.76</td>
<td>0.69</td>
</tr>
<tr>
<td>f. Burst Frequency</td>
<td>0.94</td>
<td>0.28</td>
<td>0.24</td>
<td>0.29</td>
</tr>
<tr>
<td>g. Burst Interarrival</td>
<td>6.33</td>
<td>2.83</td>
<td>0.13</td>
<td>0.1</td>
</tr>
<tr>
<td>h. Empty Queue Frequency</td>
<td>0.95</td>
<td>0.26</td>
<td>0.19</td>
<td>0.18</td>
</tr>
<tr>
<td>i. Avg count of concurrent bursts</td>
<td>0.53</td>
<td>0.12</td>
<td>0.09</td>
<td>0.08</td>
</tr>
</tbody>
</table>

Data Generation: To generate realistic queue lengths we use the ns-3 simulator [45] for the scenario described in [2]. The generated traffic follows the websearch and incast traffic patterns. Each port is mapped to two queues with different classes. We collect fine-grained time series (ground-truth) corresponding to queue lengths, per-port packet and drop counts every 1ms. We choose 1ms as our fine granularity to reduce noise as in [24]. We generate coarse-grained time series by sampling (as described in §2.1) at 50ms granularity. Thus, the goal of the imputation methods is to zoom into the coarse-grained queue length by a factor of 50, i.e., turn from 50ms granularity to 1ms.

Downstream tasks: Our tasks relate to bursts as they are hard to capture and critical for network operations. We identify bursts in both ground truth and imputed queue lengths using a specific method [56]. Subsequently, we calculate the normalized errors of burst occurrence, burst height, burst frequency, average inter-arrival time between consecutive bursts, and the number of queues experiencing a burst at the same 1ms interval during 10s. We also consider the frequency of empty queues which is crucial for queue health [23].

Results: Fig. 4 illustrates a representative example of a queue length on a short time period. When the IterativeImputer imputes it (Fig. 4a), it learns nothing from the auxiliary time series and simply connects periodic and maximum queue values. The transformer alone (Fig. 4b) detects the location of the burst but not its max and is thus inconsistent. The Transformer+KAL (Fig. 4c) learns to impute more consistent queue lengths, while Transformer+KAL+CEM (Fig. 4d) is forced to be consistent.

Table 1 summarizes our results on evaluating different versions for: (a-c) the queue imputation consistency constraints; (d-g) downstream tasks related to bursts; (h) queue health and (i) concurrent queue bursts. Lower values are better in all rows. The consistency errors (a-c) are mostly improved by incorporating KAL, and nullified by CEM. As KAL encourages higher values when bursts occur, the transformer can end up overshooting, leading to an increase in max-constraint error when only KAL is incorporated. This highlights the need for both CEM and KAL. The remaining tasks are improved by 11-96% when incorporating both KAL and CEM compared to transformer alone. We also observe that CEM does not always improve the performance of burst-related tasks. That is a trade-off between enforcing consistency and learning useful patterns. The IterativeImputer does 16-98% worse than Transformer+CEM+KAL across all tasks.

The average time for CEM to correct a 50ms transformer output is 1.47s, a significant improvement compared to FM alone which did not terminate for such imputations (§2.3).

5 Future Directions

Other means of integrating network knowledge: Our current system only uses constraints that are (or can be easily made) differentiable and thus can be incorporated into the transformer’s loss. However, network constraints are often not differentiable with respect to learnable parameters. An interesting research direction for those constraints is to train an ML model to learn the functions that imply satisfaction of the property [12]. Taking one step further, the model could output intermediate variables representing physical quantities [51] that are easier to constrain and can then be used to calculate the final result. For example, instead of imputing queue lengths directly, we can impute port incoming rates using ML and get queue lengths by associating them with routing information.

Research Question: How to optimally incorporate network-domain knowledge into ML-based systems?

Combining FM and ML in other networking problems: Many networking problems tackled separately by ML and FM could be prime candidates for combining the two. For instance, performance estimators like DeepQueueNet [58] or Mimicnet [61] can benefit from FM by bounding the delay predictions according to the shared buffer size. Moreover, generating adversarial examples for network protocols could leverage ML-based solutions [25, 33, 46] for discovering adversarial inputs and FM-based systems to ensure these inputs mirror real-world scenarios. Finally in generating synthetic traces [41, 53], one can use knowledge in the form of rules to transform existing or (synthetic) traces into new ones. Research Question: How do we strive a balance between the accuracy of FM and the creativity of ML for networking problems?

Towards practical network telemetry imputation: Our initial exploration shows the potential of software imputation of network telemetry as an alternative to hardware upgrades. Yet, our current system is quite limited in terms of: (i) target signal (queue length); (ii) imputation granularity (1ms from 50ms); and incorporated knowledge. Further investigation is needed to allow such a system to generalize to other settings and time series. Among other improvements, we believe making the system work under strict timing requirements would be particularly useful, as some tasks (such as performance-driven routing [5, 11, 26, 30], rate adaptation [28], and attack detection/prevention [36, 42]) drive real-time network activation and are hence subject to strict timing constraints. Research Question: Can we make telemetry imputation generalize and/or work in real-time?

6 Acknowledgement

This work was supported by the National Science Foundation (NSF) through Grant CNS-2319442, a Google Research Scholar Award, and a Princeton Innovation Fund Award.