Learning to Configure Computer Networks with Neural Algorithmic Reasoning

Luca Beurer-Kellner[1,*] **Martin Vechev**[1] **Laurent Vanbever**[1] **Petar Veličković**[2]

[1]ETH Zurich, Switzerland [2]DeepMind

https://github.com/eth-sri/learning-to-configure-networks

Abstract

We present a new method for scaling automatic configuration of computer networks. The key idea is to relax the computationally hard search problem of finding a configuration that satisfies a given specification into an approximate objective amenable to learning-based techniques. Based on this idea, we train a neural algorithmic model which learns to generate configurations likely to (fully or partially) satisfy a given specification under existing routing protocols. By relaxing the rigid satisfaction guarantees, our approach (i) enables greater flexibility: it is protocol-agnostic, enables cross-protocol reasoning, and does not depend on hardcoded rules; and (ii) finds configurations for much larger computer networks than previously possible. Our learned synthesizer is up to $490\times$ faster than state-of-the-art SMT-based methods, while producing configurations which on average satisfy more than 92% of the provided requirements.

1 Introduction

Configuring large-scale networks is a challenging and important task as network configuration mistakes regularly lead to massive internet-wide outages affecting millions (resp. billions[2]) of Internet users [35, 25]. Typically, network operators provide a router-level configuration W which, after applying protocols such as shortest-path routing, induces a certain forwarding behaviour FWD as illustrated in Figure 1. As this remains a challenging task, much recent research has focused on automating configuration by leveraging synthesis techniques [15, 5, 31]: A synthesizer is used to automatically generate a router-level configuration W that, after applying routing protocols results in forwarding behavior that satisfies a given specification S on how traffic should be routed.

SMT-based Synthesis Due to the hardness of the configuration synthesis problem [7], many effective tools in this domain [15, 14] resort to satisfiability modulo theory (SMT) solvers, which employ search-based procedures to find a solution to a set of first-order logic constraints. This enables comprehensive and exact synthesis by modelling network behavior in first-order logic. However, these tools are typically protocol-specific, hand-coded, and can exhibit discrepancies in behavior when compared to actual router hardware [6]. Most importantly, however, they can be very slow or fail to complete for large networks. For example, the state-of-the-art SMT-based tool NetComplete [15] requires more than 6 hours to synthesize a configuration for a network with 64 nodes, for which other SMT-based tools like SyNET [14] take even longer (> 24 hours) [15]. Non-SMT-based tools such as Propane [4] or Zeppelin [39] have achieved better performance, but at the cost of generality.

*Correspondence to luca.beurer-kellner@inf.ethz.ch.

[2]As of 2021, Facebook reportedly has 2.9 billion monthly active users [1].

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

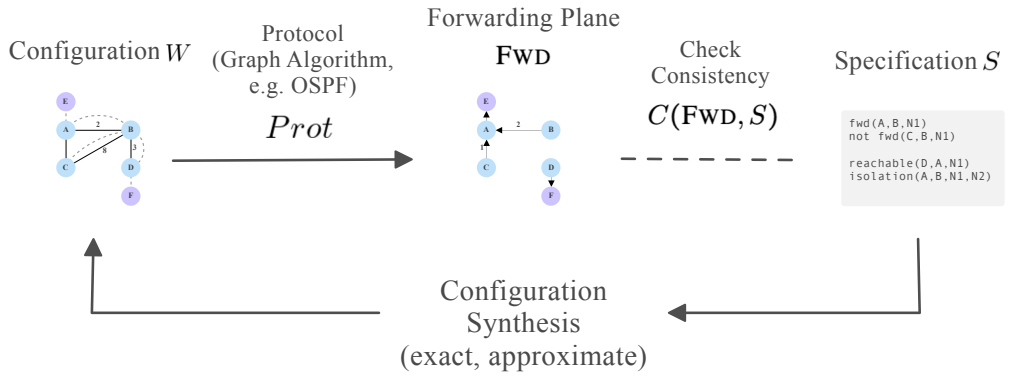

Figure 1: **Network configuration synthesis.** The goal of configuration synthesis is to find a configuration W that maximizes consistency $C(\text{FWD}, S)$ for a given specification S.

Addressing the scalability barrier The reason for the wide-spread use of SMT in configuration synthesis is the inherent computational complexity of the underlying synthesis problem, parts of which have been shown to be NP-hard [7, 16, 44]. This means that any exact synthesis method is bound to run into scalability issues (as do all SMT-based methods). However, to be practically useful, a synthesizer must scale to the size of real-world networks and the frequency at which configurations are updated. For example, in a Tier-1 ISP, network operators modify their configurations up to 20 times per day, on average [36]. We argue that one way to address this scalability barrier, is relaxing the configuration synthesis problem to admit approximate solutions with high utility – configurations which may not always satisfy all requirements of a given specification but may satisfy almost all of them. Such a configuration would be a much better starting point for a network operator than having no automated support whatsoever. The core technical challenge then is coming up with a strategy likely to find solutions with high utility.

This Work: enabling fast and scalable configuration via neural algorithmic reasoning We address this challenge and present the first learning-based framework for approximate configuration synthesis. Our relaxed formulation allows us, for the first time, to apply end-to-end learning to the problem of network configuration synthesis, thereby enabling fast and scalable configuration synthesis with almost interactive response times (<90s even for large networks). Technically, we leverage the observation that routing protocols can often be formulated as Bellman-Ford style graph algorithms, a class of problems that has recently been studied in the area of neural algorithmic reasoning (NAR) [43]. Connecting the two fields and building on ideas from NAR, we are able to train a graph-based neural model with a strong inductive bias to learn an inverse mapping from specifications back to network configurations: Our model learns how to perform synthesis from a dataset of (specification, configuration) pairs obtained by simulating the involved protocols and observing the computed forwarding state. With this method, we can support cross-protocol reasoning and do not have to manually provide any hardcoded synthesis rules. Concretely, we introduce a generic embedding scheme for topologies and configurations, making our method protocol-agnostic. During synthesis, given a specification, our model predicts distributions of network configurations from which we can sample possible results.

Main Contributions Our core contributions are:

- We formulate a relaxation of the exact configuration synthesis problem, which enables fast and scalable network configuration, amenable to learning-based techniques (Section 2).
- We propose a NAR-based neural network architecture for learning synthesizer models that rely on a graph-based encoding of topologies and configurations, and a strong inductive bias towards an iterative synthesis procedure (Section 4).
- We conduct an extensive evaluation of our learning-based synthesizer with respect to both precision and scalability. We demonstrate that our learned synthesizer is up to $490\times$ faster than a state-of-the-art SMT-based tool while producing high utility configurations which on average satisfy $> 93\%$ of provided constraints (Section 5).

2 Configuration Synthesis: Exact and Learned

We first state the general configuration synthesis problem and explain why it is hard to solve. We then present a rather different approach based on learning that addresses the scalability barrier of traditional synthesis.

Forwarding Behavior and Specifications We focus on the level of the *forwarding plane* of a network. This means we consider how a network forwards traffic, given a packet with a certain destination. The forwarding plane is determined by a distributed computation that depends on the different routing protocols in use. More formally, we define the forwarding plane FWD as follows:

$$\text{FWD} := Prot(W;T)$$

$Prot(W;T)$ corresponds to the result of applying routing protocols to the network topology T and the configuration W (e.g. link weights). In the following, we omit T as it remains fixed in synthesis. The resulting forwarding plane FWD can be understood as a directed graph superimposed on topology T. It specifies a subset of links that are used to forward packets. To illustrate consider Figure 1: applying the routing protocols yields forwarding plane FWD which corresponds to the subset of links $(C, A), (B, A), (A, E), (D, F)$. The other links of the network are not part of the forwarding plane and will thus not be used to forward traffic.

Given FWD , we consider a forwarding specification $S := \{R_i\}_i$ as an input to the synthesis problem. Each requirement in S is modelled as a function R_i, where $R_i(\text{FWD}) = 1$ if FWD satisfies the requirement and 0 otherwise. Practical example requirements include reachability, traffic isolation or specifying concrete forwarding paths.

Exact Configuration Synthesis We formulate the general configuration synthesis problem as the following optimization objective:

$$W^\star := \underset{W \in P(W)}{\arg\max}\, C(Prot(W), S) \quad \text{where} \quad C(\text{FWD}, S) := \frac{\sum_{R_i \in S} R_i(\text{FWD})}{|S|} \tag{1}$$

$P(W)$ denotes the set of all possible configurations and $C(\text{FWD}, S)$ is the *specification consistency* of a forwarding plane FWD w.r.t specification S. In traditional, exact configuration synthesis, this objective is solved by limiting the search to globally-optimal configurations such that $C(\text{FWD}, S) = 1.0$, i.e. *all* requirements must be satisfied. Such exact methods typically resort to SMT solvers because of the hardness of the underlying problem: configurations comprise a large number of tunable parameters, where the execution of several interacting protocols yields the overall forwarding state. Even worse, parts of the configuration synthesis problem have been shown to be NP-hard [7, 16, 44]: For example, already the subproblem of finding link weights that yield a given set of forwarding paths under shortest-path routing is NP-hard [7]. This makes scaling exact synthesis to real-world networks extremely challenging.

Learning-Based Synthesis To enable fast and scalable configuration synthesis, we propose to relax both the optimality as well as the rigid satisfaction requirements w.r.t the specification S. Concretely, we relax the set of admissible solutions to include configurations that are not optimal, but still satisfy a large number of provided requirements. Note that this is not the same as merely allowing solutions with $C(\text{FWD}, S) < 1.0$, because maximum satisfiability does not relax the hardness of the problem. Instead, we propose to search for near-optimal, good solutions and rely on the value of specification consistency C as a measure of quality.

An approximate synthesis formulation relaxes the hardness of the problem, however, it also leads to the difficult technical challenge of finding solutions with high utility (e.g., where many requirements are satisfied). To address this challenge, we propose a rather different approach where we learn synthesis from data. Concretely, we learn an inverse mapping that attempts to predict approximate solutions $\hat{W}^\star$ with high utility, as guided by the following objective:

$$\hat{W}^\star = Prot^{-1}(\text{FWD}) \;\text{ s.t. }\; C(\text{FWD}, S) \text{ is high}$$

This can be implemented as a synthesizer model M_{Syn} which produces a solution given just the topology T and the specification S:

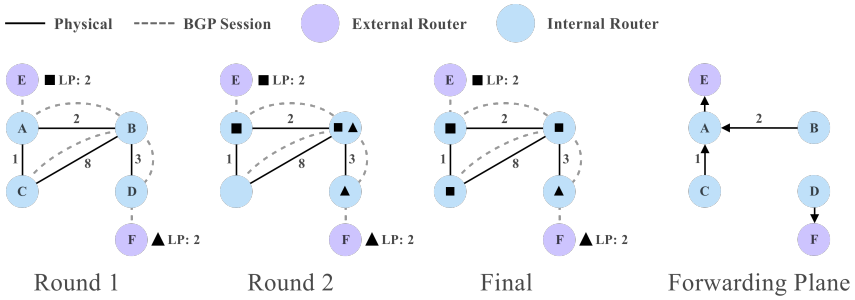

Figure 2: An illustration of propagating BGP route announcements in a small network, including internal peers (blue), external peers (purple), BGP sessions as well as physical links. The last graph illustrates the resulting forwarding plane after applying BGP/OSPF.

$$\hat{W}^{\star} = M_{Syn}(S;T)$$

We propose a learning-based approach for training such synthesizer models based on neural algorithmic reasoning (cf. Section 4). First, however, we discuss the routing protocols that make up function $Prot$ and how they relate to graph algorithms and by extension to NAR.

3 Routing Protocols as Graph Algorithms

In practice, the routing protocols defining $Prot$ are implemented as distributed systems in which multiple peers communicate to determine the network's forwarding state. However, theoretical work on routing algebras [21] has shown that the underlying computation can be understood as a traditional message-passing graph algorithm. As a consequence, many routing protocols can be formulated as Bellman-Ford (BF) style propagation processes. This class of problems has also recently been subject to work on NAR [43, 41] and algorithmic alignment [45]. The authors of these works demonstrate that neural networks are capable of closely imitating BF-style algorithms when provided with a suitable inductive bias. Based on this insight, NAR proposes to replace traditional algorithms with neural networks to learn improved algorithmic procedures or extend existing algorithms to be applicable to raw data [41]. Following the idea of NAR, we implement a synthesizer model for network configurations as an iterative Graph Neural Network (GNN) to learn $Prot^{-1}$ by relying on a Bellman-Ford style inductive bias.

Synthesis Setting Our approach is general for the domain of networks, but we focus on two widely-used routing protocols: (1) Open Shortest Path First (OSPF) [30] – it uses link weights to route traffic along the shortest path towards the destination, and the (2) Border Gateway Protocol (BGP) [32], used to exchange reachability information, mostly on the level of larger backbone networks. When using BGP, a routing destination announces its existence to other networks and routers by sending out BGP announcements. Receivers of announcements then choose to pass them on to other peers, redistribute them internally and/or modify them according to a set of decision rules. BGP and OSPF interact, for instance, BGP will consider the OSPF cost of internal destinations in its routing decisions. This means, that effective BGP/OSPF synthesis tools must implement cross-protocol reasoning, configuring BGP and OSPF, such that together they yield the desired forwarding behaviour.

Example To provide a basic intuition about BGP/OSPF routing, consider the simple example network in Figure 2. We apply the two protocols to obtain the forwarding plane. Physical OSPF edges are labelled with a corresponding link weight determining the OSPF cost of paths through the network. Dotted lines represent designated BGP edges, used to propagate BGP announcements. Our network imports multiple BGP announcements ■ and ▲ from E and F respectively. Both represent a route to the same destination. As shown, the announcements have a so-called BGP local preference of 2 (we ignore other BGP properties in this example). The announcements are propagated through the network and the best route is selected according to a designated BGP decision procedure. Figure 2 shows the intermediate states of this propagation process. As both announcements have the same local preference value, the decision cannot eliminate based on that. Instead, in round 2, node B

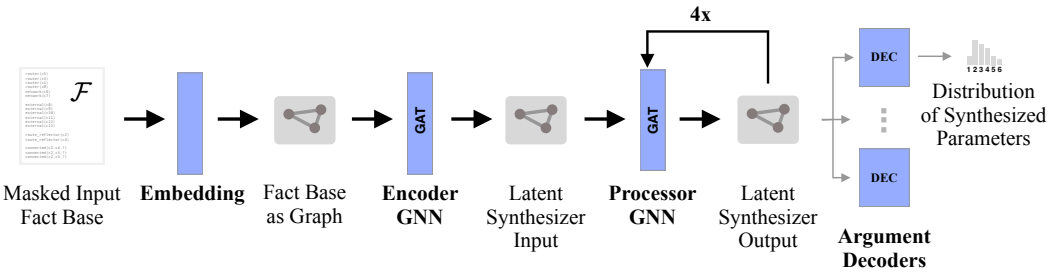

Figure 3: **Neural synthesizer architecture**. A provided input fact base is first embedded as a graph including the masked to-be-synthesized parameters. Then encoder GNN, processor GNN and decoder networks are applied to obtain a distribution over synthesized parameters.

selects ■ over ▲ due to a lower OSPF cost (shorter path) of 2 via node A as compared to 3 via node D. Node D selects ▲ over ■, since it learns this route directly from an external peer which is preferred in BGP. After Round 3, the BGP propagation process converges to a stable state and we can derive the forwarding plane as shown on the left in Figure 2. For completeness, we include the full BGP decision process in Appendix A.3.

Configuration Parameters For our purposes, we define the set of synthesized configuration parameters as follows: For OSPF, we synthesize link weights as explored in existing work [15, 16]. For BGP, we focus on a setting, where we synthesize BGP import policies only. This means we synthesize the modifications required for BGP announcements when entering the network, to satisfy the routing specification. Previous work has confirmed that this is a realistic configuration setting that applies to a majority of real-world networks [9, 12, 38].

Based on the observation that routing protocols such as OSPF and BGP can be expressed as message-passing algorithms, we heavily rely on GNNs/NAR for the design of our synthesizer model as discussed next. In our evaluation, we then train such a model for the concrete case of BGP/OSPF and compare synthesis performance with a traditional SMT-based tool.

4 Neural Configuration Synthesis Model

We train a graph neural network (GNN) as the neural synthesis configuration model. The model's input consists of a topology, a specification, and a configuration sketch where to-be-synthesized parameters are omitted. Following the NAR paradigm, we first encode this synthesizer input in latent space. Then, we apply an iterative processor network based on the graph attention mechanism [42]. Last, we apply a decoder network to predict the values of omitted configuration parameters, thereby synthesizing a configuration. To remain protocol-agnostic, our model is generic with respect to the input format, using an intermediate representation based on Datalog-like facts. Figure 3 provides an overview of our graph-based neural synthesizer architecture.

4.1 Training Dataset of Inverse Pairs

A learning-based synthesizer model is formulated as a supervised learning problem. Thus, we can directly train a neural network to learn the inverse mapping, given a dataset of corresponding input-output pairs. To obtain such a dataset, we sample a random network configuration for some topology using a uniform generative process. Then, we simulate the involved protocols using $Prot$, to obtain the corresponding forwarding plane. Next, we extract a specification by randomly selecting properties that hold for the computed forwarding plane. This leaves us with a pair of specification and topology as input, and a corresponding configuration as output.

The key to constructing the dataset is the implementation of $Prot$. Even though $Prot$ is protocol-specific, it turns out the overall implementation effort is comparatively low, especially when compared to SMT-based synthesis methods. Protocols are well-defined algorithms that can be easily simulated, whereas the alternative of implementing hardcoded synthesis rules directly often requires expert

```
router(A)                    network(N1)
router(B)                    bgp_route(E,N1,2,3,1,0,1)
(...)
                             fwd(A,B,N1)
conn(A,B,2)                  not fwd(B,A,N1)
conn(A,C,?)❶                 (...)
(...)
```

Figure 4: A fact base encoding a network's topology, parameters such as link weights (`conn` facts) and a specification (`fwd` facts).

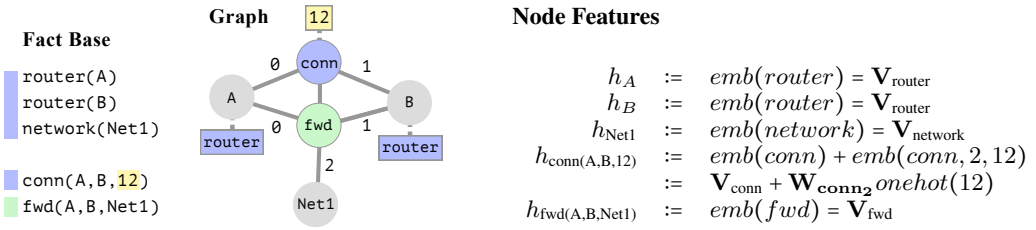

Figure 5: An example of embedding a simple fact base using our generic graph embedding scheme. Different neighborhoods are indicated as edge labels `0,1,2`. The full set of structural embedding rules is provided in Figure 6, Appendix A.1. We mark nodes and facts relating to topology, specification and configurations in color.

knowledge of SMT solvers. This process may also be adapted to rely on actual router hardware to compute the result of $Prot$, thereby capturing real-world behavior precisely.

4.2 Embedding Topologies, Specifications and Configurations

To remain agnostic with respect to routing protocols, our model architecture implements a generic graph-based encoding of Datalog-like facts, similar to knowledge graphs [33]: we first encode topologies, configurations and specifications as a set of Datalog-like facts and then employ a generic embedding scheme to embed these facts into latent space.

Fact Base A set of Datalog-like facts, as depicted in Figure 4, serves as the input *fact base* $\mathcal{F}$ to our synthesizer model. $\mathcal{F}$ is a set of facts $f(a_0, \ldots, a_n)$ where arguments may be constants (e.g. A or B) or integer literals. Each fact has a corresponding boolean truth value denoted as $[f(a)]_{\mathcal{B}}$. For instance, from the fact base in Figure 4 we can derive $[fwd(A, B, N1)]_{\mathcal{B}}$ = `true` and $[fwd(B, A, N1)]_{\mathcal{B}}$ = `false`.

Synthesis as Completion Task A network's topology, protocol configuration, parameters, and the specification are all represented in a single fact base. To predict the value of unknown, to-be-synthesized parameters, we support the notion of unknown parameters as illustrated at ❶ in Figure 4, where the link weight between router nodes `A` and `C` is omitted. Based on this input format, synthesis corresponds to using our model to predict the value of unknown parameters in a provided fact base.

Embedding To transform a fact base into a graph with node features, we apply a structurally-defined embedding scheme. An example of this embedding scheme is given in Figure 5. As shown, we embed topology, specification and configuration all into a single graph. This places network routers, the related specification predicates as well as configuration parameters in close adjacency to each other, simplifying the synthesis procedure for the processor network. In our scheme, both facts and constants are represented as distinct nodes. The relationships between facts and constants are encoded as node adjacency as defined by neighborhood functions N_i. We use multiple neighborhoods, i.e. multiple types of edges, to encode the argument position of a constant occurring in a fact. To handle unknown parameters, we replace the corresponding embedding with a learned $\mathbf{V}_{\mathbf{hole}}$ embedding. Overall, the embedding function EMB relies on a set of learned parameters, including $\mathbf{V}_{\mathbf{f}} \in \mathbb{R}^D$ per fact type in $\mathcal{F}$, $\mathbf{W}_{\text{bool}} \in \mathbb{R}^{D \times 2}$ for boolean values, $\mathbf{W}_{\mathbf{f_i}} \in \mathbb{R}^{D \times N}$ per integer argument of fact type f and $\mathbf{V}_{\mathbf{hole}} \in \mathbb{R}^D$ to represent unknown parameters. D is the dimensionality of the latent space and N

Table 1: Average consistency with 3×16 BGP/OSPF requirements, sampling configurations randomly from a uniform distribution and multi-shot sampling using our synthesizer model.

	Random	1-Shot	4-shot	8-shot
Small	0.87±0.11	0.94±0.04	**0.95**±0.03	**0.95**±0.04
Medium	0.81±0.10	**0.96**±0.04	**0.96**±0.04	**0.96**±0.04
Large	0.80±0.05	0.93±0.05	0.93±0.05	**0.94**±0.05

specifies the number of supported integer values. For the complete embedding scheme, please see Appendix A.1.

4.3 A NAR-based Synthesizer Model

Our synthesizer model employs an encode-process-decode architecture, inspired by neural algorithmic reasoning [41]. It employs four main components to produce the output distribution for an unknown parameter in a fact base $\mathcal{F}$: The fact base embedding $\textsc{Emb}$, the encoder $\textsc{Enc}_{GAT}$, the processor $\textsc{Proc}_{GAT}$ and the fact-type specific decoder network $\textsc{Dec}_{f_i}$.

Overall, the model can be expressed as follows, where X_j and H_j refer to the intermediate node representations per node j in the graph constructed via our fact base embedding.

$$\begin{aligned} X_j &\coloneqq \textsc{Enc}_{GAT}(\textsc{Emb}(\mathcal{F}))_j + \mathbf{z} \quad \text{where} \quad \mathbf{z} \sim \mathcal{N}(0,1) \\ H_j &\coloneqq \textsc{Proc}_{GAT}(\{X_i\})_j \\ O_{f_j(a_0,\ldots,a_{i-1},?,\ldots,a_n)} &\coloneqq softmax(\textsc{Dec}_i^f(H_j)) \end{aligned} \tag{2}$$

$\textsc{Enc}_{GAT}$ is a GNN relying on Graph Attention Layers (GAT) [42] for propagation. We additionally apply noise to the latent node representation X_j via $\mathbf{z}$ as a source of non-determinism, anticipating the fact that the synthesis problem often has more than one solution. The processor $\textsc{Proc}_{GAT}$ is modelled as an iterative process. It consists of a 6-layer graph attention module which we apply for a total of 4 iterations. According to the NAR paradigm, this computational structure encodes an inductive bias towards an iterative solution to the synthesis problem.

In the encoder and processor GNNs, we rely on a composed variant of the graph attention layer as introduced by [42]. We employ multiple graph attention layers in lockstep, one per neighborhood N_i defined by our graph embedding (cf. Section 4.2), and combine the intermediate results after each step by summation. For details on the graph attention module, please refer to Appendix A.2.

Finally, an argument decoder $\textsc{Dec}_{f_i}$ is used, to produce the output distribution $O_{f_j(a_0,\ldots,a_{i-1},?,\ldots,a_n)}$ for an unknown parameter at position i of fact $f_j \in \mathcal{F}$, corresponding to node j. For this, the synthesizer model provides one decoder per integer argument, per supported fact type. For example, a model for the fact base in Figure 4 would provide a decoder network $D_2^{\text{conn}} : \mathbb{R}^D \to \mathbb{R}^N$ to decode values for the third argument of a `conn` fact. Decoder networks are implemented as simple multi-layer perceptron models.

Supervision Signal During training, we mask all fact arguments that represent to-be-synthesized configuration parameters in our synthesis setting as unknown parameters. As a supervision signal we use the masked values as the ground-truth and apply a negative log-likelihood loss to the output distribution of the corresponding argument decoder networks.

Multi-Shot Sampling To sample values from the output distributions of unknown parameters, we apply a multi-shot sampling strategy. We sample values only for a subset of unknown parameters, insert them in the input fact base and run the synthesizer again. We repeat until no more unknown parameters remain. This multi-shot strategy allows the model to incorporate concrete values of previously synthesized parameters in its computation. In our evaluation, we compare this approach to sampling all parameters at once.

5 Evaluation

In this section, we assess the performance of our model trained for BGP/OSPF synthesis. For this, we trained a synthesizer model on a dataset of $10,240$ samples, constructed by randomly sampling

Table 2: Average specification consistency of our synthesizer model, where standard deviation is reported with respect to the different topologies in a dataset. We apply the model to synthesis tasks with $3 \times N$ requirements, i.e. N requirements per supported specification fact.

Dataset		`fwd`	`reachable`	`trafficIsolation`	Overall	Full Matches	> 90% Matches
3×2	S	0.97	0.94	1.00	**0.96**±0.07	6/8	6/8
	M	0.95	0.94	1.00	**0.94**±0.08	5/8	5/8
	L	0.92	1.00	1.00	**0.94**±0.06	4/8	4/8
3×8	S	0.98	0.98	0.91	**0.96**±0.05	4/8	7/8
	M	0.97	0.98	1.00	**0.98**±0.03	4/8	8/8
	L	0.96	0.92	0.97	**0.95**±0.03	1/8	8/8
3×16	S	0.98	0.92	0.95	**0.95**±0.03	2/8	8/8
	M	0.95	0.95	0.98	**0.96**±0.04	3/8	7/8
	L	0.94	0.91	0.95	**0.93**±0.05	1/8	6/8

topologies, corresponding specifications and BGP/OSPF configurations as described in Section 4.1. For details on BGP/OSPF dataset generation and training, please see Appendix A.3. Lastly, we also evaluate our architectural design decisions in an ablation and parameter study in Appendix D.

Dataset, Metrics and Experimental Setup We compare using datasets Small (S), Medium (M), and Large (L). Each dataset comprises 8 real-world topologies taken from the Topology Zoo [29], where the number of nodes lies between 0-18, 18-39, and 39-153, respectively. To obtain random forwarding specifications we use the same generative pipeline as discussed in Section 4.1. Regarding forwarding requirements, we implement support for three specification facts: `fwd` requirements to set/block forwarding paths, `reachable` to specify reachability and `trafficIsolation` to induce traffic isolation among traffic classes (no shared links). In each topology, we do synthesis for 4 different traffic classes (routing destinations) at a time. Overall, this results in 3 datasets x 8 topologies per dataset x 3 differently-size specifications = 72 synthesis tasks. To assess synthesis quality, we determine specification consistency as the relative number of specification facts in a fact base that are satisfied by the synthesized configuration. We run all experiments on an Intel(R) i9-9900X@3.5GHz machine with 64GB of system memory and an NVIDIA RTX 3080 GPU with 10GB of video memory.

5.1 Synthesis Quality

To assess the quality of synthesized network configurations, we examine specification consistency with increasingly large topologies and specifications. For each synthesis task, we run our synthesizer 5 times using 4-shot sampling and report the network configurations with the highest overall consistency. Table 2 documents the results. On average, our synthesis model achieves > 93% specification consistency for all datasets and specifications. With few requirements, the synthesizer model even succeeds in producing fully-consistent configurations (cf. Full Matches in Table 2). We observe a slight decrease in consistency with increasingly large topologies.

Multi-Shot Sampling To determine the effect of multi-shot-sampling, we report consistency when using 1-shot, 4-shot and 8-shot sampling in Table 1. As a baseline, we also show consistency when sampling configurations from a uniform distribution (per parameter). This simple method can achieve suprisingly good results, as parts of a specification may be satisfied naturally by the mechanics of shortest-path routing. Still, competing with this baseline our synthesizer model shows clear improvements and multi-shot sampling further increases consistency.

Number of Samples We also consider the number of times we sample from our synthesizer model. We observe that sampling more than one alternative configuration can lead to improved best-of specification consistency across all datasets. Based on this observation, we boost synthesis peformance by sampling multiple times per synthesis task. For each configuration that we obtain in this way, we simulate the routing protocols, obtain the forwarding plane and check specification consistency. This fully automated process allows us to determine the best result without consulting the user. We select the best result as the overall output for synthesis, dismissing the other samples. We experiment how the number of times we sample affects the resulting consistency in Appendix B.

Table 3: Comparing consistency and synthesis time of our method (Neural) with the SMT-based NetComplete. The notation n/8 TO indicates the number of timed out runs out of 8 (25+ minutes).

# Requirements		NetComplete (s)	Neural CPU (s)	Speedup	∅ Consistency	Full Matches
2 reqs.	S	18.07s±14.55	0.72s±0.54	**25.2x**	0.97±0.09	7/8
	M	60.86s±33.39	3.18s±4.32	**19.1x**	0.94±0.13	6/8
	L	1389.48s±312.58 7/8 TO	24.25s±28.35	**57.3x**	0.99±0.03	7/8
8 reqs.	S	247.69s±436.90	1.25s±1.02	**198.7x**	0.96±0.08	6/8
	M	>25m 8/8 TO	4.55s±4.30	**329.8x**	0.97±0.04	4/8
	L	>25m 8/8 TO	31.28s±28.53	**48.0x**	0.97±0.05	5/8
16 reqs.	S	1416.83s±235.25 7/8 TO	2.88s±1.66	**492.0x**	0.92±0.06	1/8
	M	>25m 8/8 TO	6.53s±5.10	**229.8x**	0.95±0.05	2/8
	L	>25m 8/8 TO	87.99s±141.97	**17.0x**	0.95±0.03	2/8

Overall, sampling more than once is beneficial for all datasets. Average best consistency values appear to be reached after 4-5 samples for 3×16 BGP/OSPF requirements. Hence, we rely on 5 samples in all other experiments as a trade-off of fast synthesis time and good specification consistency.

Unsatifiable Specifications In practice, network operators may sometimes provide unsatisfiable specifications. While exact methods will typically return an error for such inputs, our relaxed setting enables us to consider partial solutions, i.e. configurations that still achieve high specification consistency while ignoring unsat requirements. This may be preferable in some scenarios, especially when perfect specification consistency is not critical anyway. To simulate this scenario, we evaluate specification consistency for OSPF-only synthesis tasks, which were all verified to be unsatisfiable using the SMT-based synthesizer NetComplete [15]. We still observe a comparatively high average consistency of 0.90 for our learned synthesizer. This suggests that our model is indeed capable of handling unsatisfiable specifications, while still producing good, partial solutions. For more detailed unsat results and methodology, see Appendix B.2.

5.2 Comparison to SMT-based Synthesis

We compare the synthesis time of our learned synthesizer with the SMT-based, state-of-art configuration synthesis tool NetComplete [15]. We compare BGP/OSPF and OSPF-only synthesis time. For each topology/specification, we report the total time of running our synthesizer 4 times using 4-shot sampling, whereas for consistency we report the best out of the 4 runs. If a fully consistent configuration is found, we do not continue to sample and only report time until then. For NetComplete, we time out at 25 minutes and report that as a lower bound if exceeded. We restrict our comparison to specifications that only include forwarding paths (i.e. `fwd` facts), as the type of supported requirements in NetComplete does not fully align with our model.

Table 3 shows that our synthesizer model outperforms NetComplete by multiple orders of magnitude. We observe a speedup of $20 - 490\times$ that increases with the size of topology/specification. The loss of precision due to approximation remains moderate, with the average consistency of the synthesized configurations being greater than 92% even for large topologies. In comparison, NetComplete times out on more than half of the synthesis tasks, which means that we only observe a lower bound for speedup. Running our model on a GPU can result in even greater speedups (up to $900\times$ given enough GPU memory, cf. Appendix B.3). Comparing OSPF-only synthesis time, our synthesizer model achieves a speedup of 2-10x over NetComplete. With 16 requirements on dataset L this can also increase up to $500\times$. Further, our model produces fully-consistent OSPF configurations even more often than for BGP/OSPF. See Appendix B.4 for the full OSPF-only synthesis comparison.

5.3 Discussion

We have shown that our learning-based synthesizer reaches a very high degree of consistency, often producing fully-consistent configurations. With respect to synthesis time, we outperform SMT-based methods by a large margin, especially for larger topologies. However, this comes at a price: our model sometimes fails to produce fully consistent configurations, especially for large topologies

with many requirements. In comparison, SMT-based synthesis will always produce fully consistent configurations if and once it completes. We, therefore, observe a trade-off between consistency and synthesis time. We also note that our evaluation considers real topologies (Topology Zoo [29]) but not real specifications. This is due to the lack of a large, practical dataset thereof. Still, we experiment with robustness (Appendix D) by introducing distribution shift regarding the size of specifications during training and achieve comparable performance.

Scaling to Even Larger Topologies The Topology Zoo [29] as used for our evaluation, provides a good range of realistically-sized networks. However, if we consider synthesis at very large scale (e.g. thousands of routers), we note the following scalability limitations: (1) Our models consume a lot of video memory, reaching beyond the amounts available on current consumer GPUs (> 12GB). This limit is reached with networks of 150 or more routers, and we have to do inference on the CPU (as indicated in Table 3). If even faster synthesis is important at this scale, more than one such GPU is needed for inference. Further, (2) the distance that information is propagated in the synthesizer GNN is finite due to the model's fixed number of iterations. This means that in very large networks, our synthesizer model will only be capable of deriving solutions by local reasoning which is very likely to impact synthesis quality. While out of the scope of this paper, longer training with more synthesizer iterations and larger topologies may be necessary to obtain comparable results at very large scale.

Nonetheless, we envision a wide range of practically-relevant applications for fast, approximate synthesis, including ML-guided synthesis, unsatisfiable specifications, and hybrid synthesizers, leveraging both learning and SMT solvers. We list a number of future directions in Appendix C.

6 Related Work

Traditional Configuration Synthesis Next to methods based on compilation such as Propane/PropaneAT [5], there is a number of exact configuration synthesis methods based on constraint solving, e.g. ConfigAssure [31], SyNet [14] and NetComplete [15]. These tools are typically hand-coded, very protocol-specific, and can be slow due to the solvers they employ. In contrast, our approach is approximate but scales to much larger networks. Further, as a side-product of our learning-based approach, we can easily adapt to new protocols and allow for transparent cross-protocol reasoning, merely by training on different protocol data.

GNNs and Networking DeepBGP [2] relies on GNNs and reinforcement learning to do configuration synthesis. However, it is limited to BGP configuration and is slower than SMT-based synthesis. In contrast, our learning-based framework is cross-protocol, does not rely on reinforcement learning and provides better synthesis times. Apart from configuration synthesis, GNNs have also been applied to other problems in the networking domain. For example, the authors of RouteNet [34] use GNNs to predict networking performance metrics. Other work focuses on learning improved protocols like Q-Routing [8] and Graph-Query Neural Networks [18].

Neural Algorithmic Reasoning NAR [41] refers to the idea of replacing algorithms with neural networks to learn improved algorithmic procedures. Successful applications include graph algorithms [43], combinatorial optimization problems [10, 26, 37] and multi-task settings [23]. Our synthesis framework is the first application of NAR to the networking domain and relies on the NAR-native encode-process-decode architecture. In the hierarchy of NAR approaches in [10], our method is an algorithm-level approach, as we do not supervise on intermediate steps. Although step-level methods promise better generalization, it is not clear what an intermediate result of a general synthesis procedure would be. Future work on a step-level approach may further improve our model.

7 Conclusion

We presented a learning-based method to enable approximate but scalable network configuration synthesis. For BGP/OSPF routing, our neural synthesizer is up to $490\times$ faster than SMT-based methods, while producing configurations with very high specification consistency. We believe there are future research that can be explored in the direction of learning-based synthesis and ML-guided network configuration. **Ethical Issues** This work does not raise any ethical issues.

Acknowledgments This work was partially supported by an ETH Research Grant ETH-03 19-2.

References

[1] 2022. Number of monthly active Facebook users worldwide as of 3rd quarter 2021. (Feb 2022). https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/, Last-Access: 02.02.2022.

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

[3] T. Bates, R. Chandra, and E. Chen. 2000. BGP Route Reflection - An Alternative to Full Mesh iBGP, RFC2796. (2000). https://datatracker.ietf.org/doc/html/rfc2796.html

[4] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 328–341.

[5] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2017. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 437–451.

[6] Rüdiger Birkner, Tobias Brodmann, Petar Tsankov, Laurent Vanbever, and Martin T Vechev. 2021. Metha: Network Verifiers Need To Be Correct Too!. In *NSDI*. 99–113.

[7] Andreas Bley. 2007. Inapproximability results for the inverse shortest paths problem with integer lengths and unique shortest paths. *Networks: An International Journal* 50, 1 (2007), 29–36.

[8] Justin A Boyan and Michael L Littman. 1994. Packet routing in dynamically changing networks: A reinforcement learning approach. In *Advances in neural information processing systems*. 671–678.

[9] Matthew Caesar and Jennifer Rexford. 2005. BGP routing policies in ISP networks. *IEEE network* 19, 6 (2005), 5–11.

[10] Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. 2021. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint arXiv:2102.09544* (2021).

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[12] Luca Cittadini, Stefano Vissicchio, and Giuseppe Di Battista. 2010. Doing don'ts: Modifying BGP attributes within an autonomous system. In *2010 IEEE Network Operations and Management Symposium-NOMS 2010*. IEEE, 293–300.

[13] Iddo Drori and Nakul Verma. 2021. Solving Linear Algebra by Program Synthesis. *arXiv preprint arXiv:2111.08171* (2021).

[14] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2017. Network-wide configuration synthesis. In *International Conference on Computer Aided Verification*. Springer, 261–281.

[15] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 579–594.

[16] Bernard Fortz and Mikkel Thorup. 2000. Internet traffic engineering by optimizing OSPF weights. In *Proceedings IEEE INFOCOM 2000. conference on computer communications. Nineteenth annual joint conference of the IEEE computer and communications societies (Cat. No. 00CH37064)*, Vol. 2. IEEE, 519–528.

[17] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Harry Liu. 2017. Automatically repairing network control planes using an abstract representation. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 359–373.

[18] Fabien Geyer and Georg Carle. 2018. 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*. 40–45.

[19] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 1263–1272.

[20] GitHub.com. 2021. GitHub Copilot. (2021). `https://copilot.github.com`

[21] Timothy G Griffin and Joäo Luís Sobrinho. 2005. Metarouting. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*. 1–12.

[22] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580* (2012).

[23] Borja Ibarz, Vitaly Kurin, George Papamakarios, Kyriacos Nikiforou, Mehdi Bennani, Róbert Csordás, Andrew Dudzik, Matko Bošnjak, Alex Vitvitskyi, Yulia Rubanova, Andreea Deac, Beatrice Bevilacqua, Yaroslav Ganin, Charles Blundell, and Petar Veličković. 2022. A Generalist Neural Algorithmic Learner. (2022). `https://doi.org/10.48550/ARXIV.2209.11142`

[24] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*. PMLR, 448–456.

[25] Santosh Janardhan. 2021. More details about the October 4 outage. (Oct 2021). `https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/`, Last-Access: 06.10.2021.

[26] Chaitanya K Joshi, Quentin Cappart, Louis-Martin Rousseau, Thomas Laurent, and Xavier Bresson. 2020. Learning TSP requires rethinking generalization. *arXiv preprint arXiv:2006.07054* (2020).

[27] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[28] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[29] Simon Knight, Hung X Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. 2011. The internet topology zoo. *IEEE Journal on Selected Areas in Communications* 29, 9 (2011), 1765–1775.

[30] J. Moy. 1998. OSPF Version 2, RFC2328. (1998). `https://datatracker.ietf.org/doc/html/rfc2328`

[31] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. 2008. Declarative infrastructure configuration synthesis and debugging. *Journal of Network and Systems Management* 16, 3 (2008), 235–258.

[32] Yakov Rekhter, Tony Li, Susan Hares, et al. 1991. A border gateway protocol 4 (BGP-4), RFC4271. (1991). `https://datatracker.ietf.org/doc/html/rfc4271`

[33] Paolo Rosso, Dingqi Yang, and Philippe Cudré-Mauroux. 2020. Beyond triplets: hyper-relational knowledge graph embedding for link prediction. In *Proceedings of The Web Conference 2020*. 1885–1896.

[34] Krzysztof Rusek, José Suárez-Varela, Paul Almasan, Pere Barlet-Ros, and Albert Cabellos-Aparicio. 2020. RouteNet: Leveraging Graph Neural Networks for network modeling and optimization in SDN. *IEEE Journal on Selected Areas in Communications* 38, 10 (2020), 2260–2270.

[35] Adam Satariano. 2021. What is Fastly, the company behind the worldwide internet outage? (June 2021). `https://www.nytimes.com/2021/06/08/business/fastly-internet-outage.html`, Last-Access: 02.02.2022.

[36] Tibor Schneider, Rüdiger Birkner, and Laurent Vanbever. 2021. Snowcap: synthesizing network-wide configuration updates. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 33–49.

[37] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding high-performance SAT solvers with unsat-core predictions. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 336–353.

[38] Samuel Steffen, Timon Gehr, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2020. Probabilistic verification of network configurations. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 750–764.

[39] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2018. Synthesis of fault-tolerant distributed router configurations. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 1 (2018), 1–26.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[41] Petar Veličković and Charles Blundell. 2021. Neural Algorithmic Reasoning. *arXiv preprint arXiv:2105.02761* (2021).

[42] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[43] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. 2019. Neural execution of graph algorithms. *arXiv preprint arXiv:1910.10593* (2019).

[44] Stefano Vissicchio, Luca Cittadini, Laurent Vanbever, and Olivier Bonaventure. 2012. iBGP deceptions: More sessions, fewer routes. In *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2122–2130.

[45] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2019. What can neural networks reason about? *arXiv preprint arXiv:1905.13211* (2019).

Checklist

The checklist follows the references. Please read the checklist guidelines carefully for information on how to answer these questions. For each question, change the default **[TODO]** to [Yes] , [No] , or [N/A] . You are strongly encouraged to include a **justification to your answer**, either by referencing the appropriate section of your paper or providing a brief inline description. For example:

- Did you include the license to the code and datasets? [Yes] We plan to publish our code and dataset under an open source license and make it available to others.

Please do not modify the questions and only use the provided macros for your answers. Note that the Checklist section does not count towards the page limit. In your paper, please delete this instructions block and only keep the Checklist section heading above along with the questions/answers below.

1. For all authors...

(a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
(b) Did you describe the limitations of your work? [Yes]
(c) Did you discuss any potential negative societal impacts of your work? [N/A]
(d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Appendix

A Implementation and Model Details

A.1 Structural Fact Base Embedding

To transform a fact base into a graph with node features, we employ a structurally-defined embedding scheme. Given a fact base $\mathcal{F}$, we apply the rules in Figure 6 to obtain node features and adjacency information. In the resulting graph, both a fact f and a constant c are represented as distinct nodes with features h_c and h_f, respectively. The relationships between facts and constants is encoded as node adjacency as defined by the neighborhood functions N_i. We use multiple neighborhood functions, i.e. multiple types of edges, to encode the argument position of a constant occurring in a fact. Hence, the number of neighborhood functions corresponds to the maximum number of arguments in a single fact, e.g. 3 in Figure 4. Integer arguments and unary facts are directly accounted for in the respective node or fact embedding. To handle unknown parameters, we replace the corresponding $emb(...)$ term with the learned $\mathbf{V_{hole}}$ embedding.

Considering the embedding function emb, the learned parameters of our embedding are $\mathbf{V_f} \in \mathbb{R}^D$ per fact type in $\mathcal{F}$, $\mathbf{W}_{\text{bool}} \in \mathbb{R}^{D\times 2}$ for boolean values, $\mathbf{W_{f_i}} \in \mathbb{R}^{D\times N}$ per integer argument of fact type f and $\mathbf{V_{hole}} \in \mathbb{R}^D$ to represent unknown parameters. Hyperparameter D represents the dimension of the latent space the synthesizer model operates in and N specifies the number of supported integer values.

A.2 Graph Attention Layer

The recurrence relation of a node j's representation h_j is defined as:

$$
\begin{aligned}
z_j &:= h_j + Drop(\sum_{N_i} h'_{GAT_i}) \\
h'_j &:= FFN(Norm(z_j)) \\
\\
FFN(x_j) &:= Norm(Lin2(Drop(Lin1(x_j)))))
\end{aligned}
$$

h'_{GAT_i} represents the output of a graph attention layer operating with neighborhood N_i according to [42], h_j the representation of node j at the previous step. $Norm$ refers to batch normalization [24], $Drop$ to dropout regularization [22] and $Lin1$ and $Lin2$ to linear layers with an inner dimension of $4N$. The mechanic of combining the results of multiple graph attention layers h'_{GAT_i} directly corresponds to how multiple attention heads are combined in the original Transformer architecture [40].

$$
\begin{aligned}
&emb(f) := \mathbf{V_f} && \textit{(learned fact type embedding)} \\
&emb(b) := \mathbf{W}_{\text{bool}}\ onehot(b) && \textit{(learned boolean embedding)} \\
&emb(f,i,v) := \mathbf{W_{f_i}}\ onehot(v) && \textit{(learned integer embedding)} \\
&emb(f,i,?) := \mathbf{V_{hole}} && \textit{(learned hole embedding)} \\
&h_c := \sum_{f(c)\in\mathcal{F}} emb(f) && \textit{(constant embedding)} \\
&h_{f(a_0,\ldots,a_n)} := emb(f) + emb([f(a_0,\ldots,a_n)]_{\mathcal{B}}) + \sum_{(i,v)\in\mathcal{I}(f(a_0,\ldots,a_n))} emb(f,i,v) && \textit{(fact embedding)} \\
&\mathcal{I}(f(a_0,\ldots,a_n)) := \{(i,v) \mid f(a_0,\ldots,a_{i-1},v,\ldots,a_n)\in\mathcal{F} \wedge v : Int \wedge i\in\mathbb{N}\} && \textit{(integer arguments)} \\
&N_i(h_c) := \{h_f \mid \exists i\in\mathbb{N}. f(a_0,\ldots,a_{i-1},c,a_{i+1},\ldots,a_n)\in\mathcal{F}\} && \textit{(constant node neighbors)} \\
&N_i(h_f) := \{h_c \mid \exists i\in\mathbb{N}. f(a_0,\ldots,a_{i-1},c,a_{i+1},\ldots,a_n)\in\mathcal{F}\} && \textit{(fact node neighbors)}
\end{aligned}
$$

Figure 6: Translating a fact base $\mathcal{F}$ to node features. f denotes a fact type, v integer values, b boolean values as 0 or 1, i argument indices and $[f(a)]_{\mathcal{B}}$ the truth value of a fact.

A.3 Dataset Generation and Training

To train a synthesizer model, we need a large number of fact bases encoding a variety of network topologies, configurations and forwarding specifications. Further, we must provide the model with target values for unknown parameters as a supervision signal. This section discusses how we construct such a dataset for BGP/OSPF synthesis and train a corresponding synthesis model.

Dataset Generation To obtain network configurations and corresponding forwarding specifications we employ a generative process: We first randomly configure BGP and OSPF parameters for some topology. Then we simulate the OSPF and BGP protocol and compute the resulting forwarding plane. Based on this, we extract a forwarding specification which fixes a random subset of forwarding paths, reachability and isolation properties that are satisfied by the forwarding plane.

Topologies For training, our dataset is based on random topologies with 16-24 routers. We generate the physical layout of these graphs by triangulation of uniformly sampled points in two-dimensional space.

Simulation To compute the forwarding plane given an OSPF and BGP configuration, we simulate the protocols. For OSPF, we implement shortest path computation to obtain the forwarding plane. For BGP, we implement the full BGP decision process as shown in **??**, except for the MED attribute. In addition to the basic networking model discussed in Section 3, we implement support for eBGP and iBGP, fully-meshed BGP session layouts as well as layouts relying on route reflection. For more details of these BGP concepts see [3].

Fact Base Encoding We encode synthesis input using a small set of facts. For an example of a corresponding fact base see Figure 7. We use the `router`, `external` and `network` facts to mark routers, external peers and routing destinations respectively. Physical links between routers are encoded as `connected(R1,R2,W)` facts where `W` denotes the OSPF link weight. Further, we rely on `ibgp`, `ebgp`, `route_reflector` and `bgp_route` facts to represent different types of BGP sessions and imported routes. Using the notion of unknown parameters as discussed in Section 4.2, we encode configuration parameters, e.g., properties of imported routes or OSPF link weights.

Specification Language We support a small specification language with three types of forwarding predicates:

- `fwd(R1, Net, R2)` specifies that router `R1` forwards traffic destined for `Net` to its neighbor `R2`.
- `reachable(R1, Net, R2)` specifies that traffic destined for `Net` that passes through `R1` also passes through `R2` before reaching its destination.
- `trafficIsolation(R1, R2, N1, N2)` specifies that only one of $[\texttt{fwd(R1,N1,R2)}]_{\mathcal{B}}$ and $[\texttt{fwd(R1,N2,R2)}]_{\mathcal{B}}$ can be true at a time.

In the negative case of each predicate, the opposite must hold true for the fact to be satisfied. This specification language can easily be extended by including new specification predicates in the dataset generation process. One must simply provide a method of extracting positive and negative cases of a specification fact from a given forwarding plane. Once included in the training dataset of a synthesizer model, the resulting synthesis system will be able to consider specifications relying on the new type of predicates.

Training We generate a dataset of $10,240$ input/output samples and instantiate our synthesizer model with a hidden dimension $D = 64$ and supported parameter values $N = 64$. For optimization, we use the Adam optimizer [27] with a learning rate of 10^{-4}. We stop training after the specification consistency on a validation dataset no longer increases at ~ 2800 epochs.

```
# topology and configuration
router(c5)
router(c3)
router(c1)
router(c0)
network(c6)
network(c7)

external(c8)
external(c9)
external(c10)
external(c11)
external(c12)
external(c13)

route_reflector(c2)
route_reflector(c4)

connected(c2,c4,?)
connected(c2,c5,?)
connected(c2,c3,?)
connected(c4,c5,?)
connected(c4,c3,?)
connected(c4,c0,?)
connected(c5,c1,?)
connected(c5,c0,?)
connected(c3,c0,?)
connected(c3,c1,?)
connected(c1,c0,?)

ibgp(c2,c4)
ibgp(c2,c5)
ibgp(c2,c0)
ibgp(c2,c1)
ibgp(c4,c3)
ibgp(c4,c4)
```

```
ebgp(c2,c12)
ebgp(c5,c9)
ebgp(c5,c10)
ebgp(c5,c13)
ebgp(c0,c8)
ebgp(c0,c11)

bgp_route(c8,c6,?,?,1,?,1,8)
bgp_route(c9,c6,?,?,0,?,1,9)
bgp_route(c10,c6,?,?,0,?,1,10)
bgp_route(c11,c7,?,?,0,?,1,11)
bgp_route(c12,c7,?,?,0,?,1,12)
bgp_route(c13,c7,?,?,2,?,1,13)

# forwarding specification
fwd(c0,c6,c3)
fwd(c3,c6,c2)
fwd(c2,c6,c4)
not fwd(c3,c6,c0)
not fwd(c0,c6,c11)
not fwd(c11,c6,c0)

not trafficIsolation(c1,c0,c7,c6)
not trafficIsolation(c1,c0,c7,c6)
trafficIsolation(c0,c3,c6,c7)
trafficIsolation(c0,c3,c6,c7)

reachable(c5,c6,c10)
reachable(c1,c6,c4)
not reachable(c0,c6,c5)
reachable(c5,c6,c10)
```

Figure 7: An example of a fact base encoding a topology, a sketch of a BGP and OSPF configuration and a forwarding specification.

BGP Decision Process For completeness, we include the BGP decision process as assumed for our synthesis setting. The shown rules are applied in order until a single best BGP announcements remains.

1. Highest Local preference
2. Lowest AS path length
3. Prefer Origin (IGP > EBGP > INCOMPLETE)
4. Lowest Multi-exit discriminator (MED)
5. External over internal announcements
6. Lowest IGP cost to egress
7. Lowest BGP peer id (tie breaker)

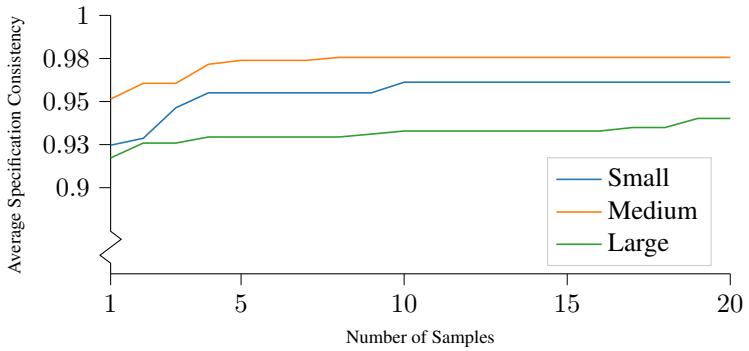

Figure 8: Average Best Specification Consistency with increasingly many samples from our synthesizer model with 3×16 BGP/OSPF requirements.

B More Evaluation Results

B.1 Number Of Samples

Figure 8 shows the average best consistency across our three datasets with an increasing number of samples for 3×16 BGP/OSPF requirements. As the number of samples increases, the graph indicates the best consistency value reached so far. We observe that sampling more than once improves the average specification consistency across all datasets. With increasing size of the topology, more samples appear to be necessary for the resulting best specification consistency to converge. Therefore, we note that increasing the number of samples can improve consistency but it will also affect overall synthesis time as the model needs to be executed again for each sample. In our other experiments we rely on 4/5 samples per specification, which we consider a good trade-off of fast synthesis time and good specification consistency.

B.2 Unsatisfiable Specifications

To asses how our model performs in the presence of unsatisfiable specifications, we evaluate its synthesis performance on multiple datasets of unsatisfiable OSPF synthesis tasks.

Specifications We construct the datasets UNSAT-N, where $N \in \{2, 4, 6, 8, 10, 12, 14, 16\}$, by first generating 16 solvable OSPF forwarding path requirements per topology in Small, Medium and Large. Then we replace N of the requirements per topology with paths obtained under other, random link weights. We repeat this process up to five times, until we can verify that all variants of a topology as contained in the different UNSAT-N datasets are indeed unsatisfiable using the SMT-based synthesizer NetComplete [15]. Overall, this leaves us with 8 UNSAT-N datasets of OSPF synthesis tasks, where we expect the maximum achievable specification consistency to decrease with increasing N. For some topologies, we were not successful in generating unsatisfiable variants for all N and therefore we removed them from the considered set of topologies for this part of our evaluation. We count 15 topologies per UNSAT-N dataset, where the number of nodes ranges between 16 and 153.

Methodology For synthesis, we use the same sampling configuration as in Section 5.1 where we again report the best consistency value across 5 different runs of our synthesis model per synthesis task. We further configure our trained synthesizer model to do OSPF synthesis by only predicting link weights.

Results We report the results of applying our synthesizer model to unsatisfiable OSPF specifications in Figure 9. For comparison, we also include the results of applying our synthesizer model to an UNSAT-0 dataset, i.e. a dataset of satisfiable OSPF synthesis tasks. The average specification consistency drops with unsatisfiable specifications which is expected since the theoretical upper bound for specification consistency is lower with conflicting requirements. At the same time, specification consistency remains high, which suggests that our model still attempts to maximize specification consistency, even when provided with an unsatisfiable specification with many conflicting requirements.

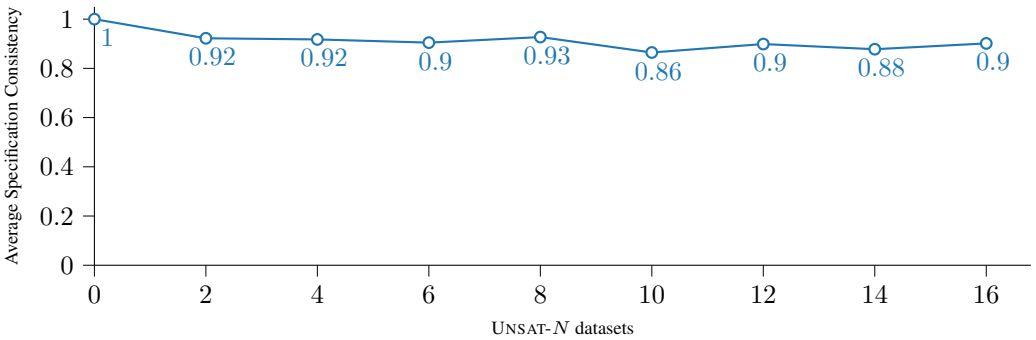

Figure 9: Average OSPF specification consistency of our synthesizer model when applied to unsatisfiable specifications.

In contrast, SMT-based synthesizers like NetComplete will not be able to produce any configurations for the synthesis tasks in our UNSAT-N datasets.

B.3 BGP/OSPF Synthesis Time (GPU)

In the following table we report BGP/OSPF consistency and synthesis time of our method (Neural) running on a GPU, compared with the SMT-based state-of-art synthesis tool NetComplete. The notation $_{n/8\ \text{TO}}$ indicates the number of timed out synthesis runs out of 8 (25+ minutes). Dataset Large is omitted due to GPU memory restrictions.

BGP Synthesis Time (GPU, 5 samples)

		Synthesis Time (s)			Accuracy (Neural)	
# Reqs.		NetComplete	Neural (GPU)	Speedup	∅ Consistency	∅ No. Full Matches
2 reqs.	S	18.07s±14.55	0.44s±0.26	**41.1x**	1.00±0.00	8/8
	M	60.86s±33.39	0.74s±0.69	**81.8x**	0.94±0.13	6/8
8 reqs.	S	247.69s±436.90	0.91s±0.70	**270.7x**	0.95±0.08	5/8
	M	>25m $_{8/8\ \text{TO}}$	1.23s±0.72	**1220.6x**	0.97±0.04	4/8
16 reqs.	S	1416.83s±235.25 $_{7/8\ \text{TO}}$	1.55s±0.51	**913.3x**	0.92±0.06	1/8
	M	>25m $_{8/8\ \text{TO}}$	1.65s±0.51	**906.5x**	0.93±0.05	1/8

B.4 OSPF Synthesis Time

We also compare consistency and synthesis time of our method (Neural) with the SMT-based synthesis tool NetComplete doing OSPF-only synthesis. The notation $_{n/8\ \text{TO}}$ indicates the number of timed out synthesis runs out of 8 (25+ minutes). For this experiment, our neural synthesizer can run on the GPU, as OSPF-only synthesis consumes less GPU memory.

Below, we report results of OSPF-only synthesis sampling from the synthesizer model 4 times and 5 times respectively. We observe that sampling 4 times can be enough with smaller specifications to obtain full matches for all synthesis tasks. For larger specifications and networks, sampling more than 4 times can improve average consistency. Note however, that sampling 5 times for small specifications and topologies can lead to synthesis times that are longer than with NetComplete (0.7x with 2 reqs., 5 samples, dataset S).

OSPF Synthesis Time (4 samples)

		Synthesis Time (s)			Accuracy (Neural)	
		NetComplete	Neural (GPU)	Speedup	∅ Consistency	∅ No. Full Matches
2 reqs.	S	0.27s±0.09	0.09s±0.00	**2.9x**	1.00±0.00	8/8
	M	0.40s±0.09	0.10s±0.00	**4.1x**	1.00±0.00	8/8
	L	0.94s±0.33	0.14s±0.08	**6.7x**	1.00±0.00	8/8
8 reqs.	S	0.88s±0.18	0.16s±0.12	**5.3x**	0.98±0.04	6/8
	M	1.52s±0.40	0.10s±0.01	**14.7x**	1.00±0.00	8/8
	L	3.50s±1.11	0.32s±0.15	**11.1x**	0.98±0.03	4/8
16 reqs.	S	1.65s±0.37	0.21s±0.14	**7.9x**	0.99±0.01	5/8
	M	2.65s±0.70	0.25s±0.16	**10.8x**	0.99±0.01	4/8
	L	566.64s±772.90 $_{3/8\ \text{TO}}$	0.23s±0.11	**2430.4x**	0.99±0.04	7/8

OSPF Synthesis Time (5 samples)

		Synthesis Time (s)			Accuracy (Neural)	
		NetComplete	Neural (GPU)	Speedup	∅ Consistency	∅ No. Full Matches
2 reqs.	S	0.27s±0.09	0.36s±0.01	**0.7x**	1.00±0.00	8/8
	M	0.40s±0.09	0.37s±0.00	**1.1x**	1.00±0.00	8/8
	L	0.94s±0.33	0.52s±0.21	**1.8x**	1.00±0.00	8/8
8 reqs.	S	0.88s±0.18	0.70s±0.63	**1.3x**	0.98±0.04	6/8
	M	1.52s±0.40	0.37s±0.00	**4.1x**	1.00±0.00	8/8
	L	3.50s±1.11	1.69s±1.08	**2.1x**	0.98±0.02	4/8
16 reqs.	S	1.65s±0.37	0.81s±0.69	**2.0x**	0.99±0.02	6/8
	M	2.65s±0.70	1.01s±0.73	**2.6x**	0.99±0.01	5/8
	L	566.64s±772.90 $_{3/8\ \text{TO}}$	1.09s±0.89	**518.1x**	0.99±0.04	6/8

C Future Research Directions

The key challenge with our learning-based system remains its imprecision, i.e. that it produces only partially-consistent configurations in some cases. On one hand, parts of the configuration synthesis problem are NP-hard [7, 16, 44], which means that practical and scalable synthesis tools have to compromise on precision unless P=NP. At the same time, satisfying the complete specification may be crucial, depending on the scenario. Therefore, our learning-based synthesis will not be applicable in the same way as precise, SMT-based synthesizers. Based on this insight, we envision a whole range of future directions for which imprecise, learning-based synthesis will be very useful:

ML-assisted Configuration Imprecise synthesis may be used to enable ML-assisted configuration in the form of a semi-automated process: users provide a specification, apply the synthesizer and perform additional tweaking after they obtain a sufficiently consistent configuration. Optionally, our synthesizer model can also be applied again to predict alternative values for some of the configuration parameters, by masking only the desired parameters in a configuration (cf. Section 4.2). Recently, similar systems for code completion, like GitHub CoPilot [11, 20], have demonstrated that this query-predict-modify workflow can be highly effective at combining an imprecise ML-guided recommendation system with user interaction [13].

Hybrid Systems Given that further tweaking of a configuration may be necessary, future work may also explore seeding SMT-based, partial synthesis methods [15] or configuration repair methods [17] with generated candidate solutions. This can further automate the process of obtaining fully consistent configurations while maintaining some of the performance benefits of a learning-based system. Existing work on partial synthesis has already shown that SMT-based synthesis can be much faster when a rough sketch of a configuration is already provided as an input [15].

Imprecision in SMT-based synthesis Tweaking the result of a synthesizer is undesirable, but it is also a common practice with existing SMT-based synthesis, as it can be imprecise too. This can be caused by hand-coded SMT synthesis rules which do not always hold up in practice, given the complex behavior of real-world routers [6]. Similarly, our learning-based method is imprecise, but could actually be trained on real-world data, possibly bridging the gap between an assumed formal model and real-world behavior. Future work is necessary to clarify how the configuration tweaking process differs in practice between SMT-based and learning-based synthesis.

Optimality with Unsatisfiable Specifications As demonstrated in Appendix B.2, our learning-based system produces highly consistent configurations even in the presence of an unsatisfiable specification. In practice, this can be helpful as operators do not know whether their specification is unsatisfiable ahead of time. In these cases, a partially consistent configuration is the best one can hope for. In contrast, using SMT solvers in this scenario will not produce a configuration at all, while unsatisfiable specifications can be quite difficult to debug.

Service Level Agreements Lastly, the notion of relaxed specification consistency is not entirely unbeknownst to the world of networking. Networks often operate in accordance with so-called Service Level Agreements (SLA), contracts that specify the guaranteed properties of service in terms of relative availability over time. Given this concept of SLAs, partially violated specifications in some scenarios can be tolerable, as explored by previous work like [38].

D Additional Experiments

In response to the reviews we carried out a number of additional experiments. We will integrate these results into our draft where applicable.

D.1 Ablation and Parameter Study

We conducted an ablation study to explore both the effectiveness of our architecture as well as the use of different components (e.g. different GNN layer modules). We compare across the following configurations. Configurations marked with (*) correspond to the configuration presented in the main body of the paper.

Noise and Edge Types NoNoise corresponds to our model without adding gaussian noise before applying the processor network. NoEdgeTypes corresponds to our model with only one edge type. NoiseEdgeTy (*) correspond to our model as presented in the paper.

GNN Modules We also experiment with different GNN modules used internally by the encoder and processor network (cf. Section 4.3). In addition to a graph attention module (GAT, [42]), we also evaluate models that employ a simple message-passing layer (MPNN-max, [19]) and a graph convolutional layer (GCN-max, [28]), both aggregating by maximization.

Hidden Dimensionality We also experiment with different values (16, 32, 64 and 128) for dimensionality D of the synthesizer model.

Separate Latent Spaces According to our graph-based encoding, we embed information on topology, configurations and specification all in a common latent space. However, we also experiment with embedding the topology and configuration information into three separate latent spaces. For this, we train models with three different pairs of encoder+processor networks, one per type of facts/nodes relating to topology, configuration and specification respectively. At each layer and synthesizer iteration, the different encoder/processor GNNs can attend to the intermediate representations of the adjacent nodes according to the underlying fact base graph. As this induces a threefold increase in parameters, we compare the results of such synthesizer models SepSpace-16, SepSpace-64 and our model ComSpace-64 (*). Due to the long training time of these models, we compare after training these configuration for 2000 epochs only.

Sampling, Dataset and Metrics For the examined configurations, we train a synthesizer model using the same training setup as described in A.3. Using the resulting models, we perform synthesis for all datasets S, M, and L, relying on 5 samples per task. For NoNoise and NoiseEdgeTy (*) we additionally perform synthesis for the same datasets, but with 20 samples. Here, the test datasets S, M and L include all generated synthesis tasks of the specification size classes 3x2, 3x8 and 3x16 as used in other parts of our evaluation. Overall, this leads to 24 synthesis tasks on 8 different topologies per dataset S/M/L. We measure average best specification consistency, as well as number of full (Full) and partial, good ($> 90\%$) matches in terms of specification consistency.

Results The results of our experiments are provided in Table 4 and Table 5. For supporting multiple edge types in NoiseEdgeTy, we observe a clear benefit over NoEdgeTypes, both with respect to average best consistency as well as the total number of full and good matches. With just 5 samples, the effect of NoNoise is not very pronounced. However when increasing the number times we sample from the synthesizer model per task, we observe a clear performance improvement. When sampling up to 20 times from the model, NoiseEdgeTy (*) outperforms NoNoise in almost all metrics (cf. Table 5), leading to higher average consistency and a higher number of full and good matches. We hypothesize that adding noise helps the model in producing a wider variety of solutions, eventually leading to better results when selecting the best one.

Regarding the choice of GNN module, we observe that GAT is the most effective, but MPNN-max can also be a good choice.

Regarding the choice of hidden dimension, we observe that 64 is a good balance of performance and memory usage, while 128 only brings slight improvements. For our purposes we selected $D = 64$, since this can improve synthesis time significantly when running on a GPU (cf. Appendix B.3). For

Ablation Results (5 samples)

Configuration	S	Full	> 90%	M	Full	> 90%	L	Full	> 90%
NOISEEDGETY (*)	**0.97±0.05**	**15/24**	**20/24**	**0.96±0.05**	**12/24**	**19/24**	0.95±0.03	7/24	18/24
NONOISE	0.96±0.07	**15/24**	19/24	0.95±0.05	10/24	**19/24**	**0.96±0.04**	**8/24**	**21/24**
NOEDGETYPES	0.94±0.07	11/24	18/24	0.93±0.08	9/24	17/24	¸0.92±0.07	6/24	16/24
GAT [42] (*)	**0.96±0.05**	12/24	**21/24**	**0.96±0.06**	**12/24**	**20/24**	**0.94±0.05**	**6/24**	**18/24**
MPNN-Max [19]	**0.96±0.05**	**14/24**	20/24	0.95±0.05	9/24	18/24	0.93±0.05	5/24	14/24
GCN-Max [28]	0.93±0.09	11/24	16/24	0.94±0.08	9/24	17/24	0.92±0.06	4/24	14/24
Hidden Dim 16	0.96±0.05	14/24	19/24	0.93±0.07	7/24	18/24	0.90±0.08	4/24	11/24
Hidden Dim 32	0.96±0.05	12/24	18/24	0.94±0.06	8/24	18/24	0.94±0.07	6/24	18/24
Hidden Dim 64 (*)	**0.97±0.05**	**15/24**	20/24	**0.96±0.05**	**12/24**	19/24	**0.95±0.03**	7/24	18/24
Hidden Dim 128	**0.97±0.05**	**15/24**	19/24	**0.96±0.05**	**12/24**	**20/24**	**0.95±0.06**	**9/24**	**20/24**
COMSPACE-64 (*)	**0.96±0.05**	14/24	**20/24**	**0.95±0.06**	**11/24**	19/24	0.95±0.05	8/24	19/24
SEPSPACE-64	0.95±0.09	**15/24**	**20/24**	**0.95±0.06**	10/24	18/24	**0.96±0.03**	**9/24**	**21/24**
SEPSPACE-16	0.94±0.07	10/24	16/24	**0.95±0.05**	**11/24**	**21/24**	0.93±0.07	6/24	17/24

Table 4: The main results of our ablation and parameter study.

Ablation Results (20 samples)

Configuration	S	Full	> 90%	M	Full	> 90%	L	Full	> 90%
NOISEEDGETY (*)	**0.98±0.03**	16/24	**22/24**	**0.99±0.03**	**16/24**	**22/24**	**0.97±0.03**	**12/24**	**23/24**
NONOISE	0.97±0.06	**18/24**	**22/24**	0.97±0.04	12/24	21/24	0.96±0.05	9/24	19/24

Table 5: Synthesis performance when sampling 20 times from a synthesizer model with and without adding noise (NOISEEDGETY (*) vs. NONOISE).

$D = 128$ this may not always be possible, especially when limited to a single GPU and working with large topologies.

Lastly, we cannot observe a significant benefit of separating the latent spaces of topology, configuration and specification as described above using multiple encoder+processer networks. SEPSPACE-64 appears to perform mostly on par with our COMSPACE-64 configuration, while having significantly more parameters. SEPSPACE-16 is more comparable in terms of the number of parameters, but it performs worse than COMSPACE-64 in most metrics.

D.2 Dataset Statistics and Distribution Shift

To examine the synthesis performance of our model with real topologies, we source our evaluation datasets from the Topology Zoo [29], a collection of real-world topologies. For training on the other hand, we only rely on synthetic data based on random topologies, configurations and specifications that are easy to generate by simulation (cf. A.3). As a consequence, we observe distribution shift between the synthetic data we train on and the closer-to-real-world data we evaluate on.

Training and Evaluation Datasets Figure 10 illustrates parts of the distribution shift with respect to the number of nodes and specifications size. Our synthetic training dataset (Training Dataset) only contains samples of very limited size (15-25 nodes) with ~20-40 different specification predicates per traffic class. In contrast, our Topology Zoo datasets used for evaluation contain topologies of much larger size (5-153) and larger specifications (~1-60 predicates per traffic class). Considering this distribution shift and the good performance of learned synthesis on the evaluation datasets, our model appears to generalize well to larger topologies/specifications, even without having seen similarly-sized problem instances during training. Comparably strong generalization properties have also been previously observed with other NAR-based models [43].

Specification Distribution Shift (SmallSpec) Our evaluation datasets rely on real topologies but have to resort to synthetic specifications due to the lack of a large, practical dataset in this space. Nonetheless, we want to provide some preliminary insight into the possible effect of a specification

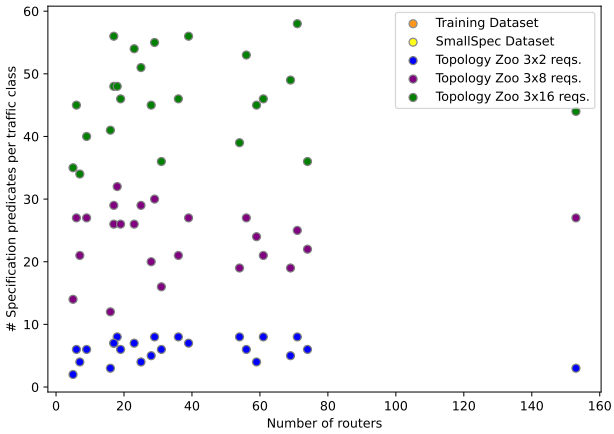

Figure 10: Dataset distribution of network and specification size in terms of routers and the number of specification predicates per traffic class respectively.

SmallSpec Distribution Shift in Training (5 samples)

Configuration	S	Full	> 90%	M	Full	> 90%	L	Full	> 90%
Training Dataset (*)	**0.97±0.05**	**15/24**	20/24	**0.96±0.05**	**12/24**	**19/24**	**0.95±0.03**	**7/24**	18/24
SmallSpec	**0.97±0.04**	14/24	**22/24**	0.95±0.06	10/24	18/24	0.94±0.05	6/24	**19/24**

Table 6: Examining the effect of specification distribution shift by comparing synthesis performance of a model trained on the SmallSpec dataset (smaller specifications) with a model trained on our regular training dataset.

distribution shift, as would be the case when applying our model to real-world synthesis tasks. For this, we construct a SmallSpec dataset (cf. Figure 10) which is similar to our training dataset but with smaller specifications. In Table 6 we report the results of training a synthesizer model on the SmallSpec dataset and comparing its performance with a model trained on our regular Training Dataset. Both models are evaluated on the same Topology Zoo evaluation datasets as in our evaluation. The resulting SmallSpec model achieves slightly lower but comparable synthesis performance. This provides some evidence regarding the robustness of our models when it comes to a distribution shift of the specifications used for synthesis.

D.3 Varying Number of Samples

In a previous revision of the paper, our evaluation relied on 5 samples to obtain consistency results and only 4 to determine synthesis times. In practice, this can be a sensible choice, trading off accuracy for speed. However, to rectify this discrepancy in our results, we carried out the respective dual experiments to also determine average synthesis times and consistency results with 5 and 4 samples respectively. The experiments in the main body of this revision all rely on 5 samples.

Consistency We compare synthesis performance when sampling 4/5 times from our synthesis model. Below we list the results for average best consistency, number of full matches as well as the number of > 90% matches for each of the datasets S, M, L. The results correspond directly to the results in column "Overall" in Table 2, where we report the results for sampling 5 times.

BGP/OSPF Consistency (4 samples vs. 5 samples)

Req.	Samples	S	Full	> 90%	M	Full	> 90%	L	Full	> 90%
3x2	4 samples	**0.96±0.07**	**6/8**	**6/8**	0.91±0.11	4/8	4/8	0.91±0.11	**5/8**	**5/8**
	5 samples	**0.96±0.07**	**6/8**	**6/8**	**0.94±0.08**	**5/8**	**5/8**	**0.94±0.006**	4/8	4/8
3x8	4 samples	**0.97±0.04**	3/8	**7/8**	**0.98±0.02**	3/8	**8/8**	**0.95±0.04**	**1/8**	7/8
	5 samples	0.96±0.04	**4/8**	**7/8**	**0.98±0.03**	**4/8**	**8/8**	**0.95±0.03**	**1/8**	**8/8**
3x16	4 samples	**0.95±0.03**	**2/8**	**8/8**	0.95±0.04	2/8	6/8	**0.95±0.04**	**1/8**	**6/8**
	5 samples	**0.95±0.03**	**2/8**	**8/8**	**0.96±0.04**	**3/8**	**7/8**	0.93±0.05	**1/8**	**6/8**

These results suggest that a higher number of samples can increase average consistency as well as the number of full and good, partial matches. Especially small specifications (cf. 3x2), seem to benefit from more samples. Overall, however there is not a very large difference between sampling 4/5 times.

Synthesis Time We also compare synthesis time of sampling 4/5 times below. We report the synthesis time of running our synthesizer model with 4 samples as reported in the paper. Next, we report synthesis time of running our synthesizer model with 5 samples. Last, we report the speedup over NetComplete when running our synthesizer model with 4 and 5 samples, respectively.

BGP/OSPF Synthesis Time (4 samples vs. 5 samples)

# Requirements		4 samples (s)	5 samples (s)	Speedup (4 samples)	Speedup (5 samples)
2 reqs.	S	0.64s±0.38	0.72s±0.54	**28.2x**	25.2x
	M	2.75s±3.29	3.18s±4.32	**22.2x**	19.1x
	L	22.30s±26.86	24.25s±28.35	**62.3x**	57.3x
8 reqs.	S	1.07s±0.84	1.25s±1.02	**232.5x**	198.7x
	M	3.47s±3.34	4.55s±4.30	**432.2x**	329.8x
	L	30.96s±28.18	31.28s±28.53	**48.4x**	48.0x
16 reqs.	S	2.53s±1.85	2.88s±1.66	**560.5x**	492.0x
	M	5.48s±3.90	6.53s±5.10	**273.7x**	229.8x
	L	69.09s±108.17	87.99s±141.97	**21.7x**	17.0x

Overall, running our synthesizer model with 5 samples means that the model is invoked one more time per synthesis task. This is clearly reflected by the resulting synthesis times. However, as the number of samples is only a linear factor for overall synthesis time, the speedup over NetComplete remains very significant.