Self-Learning Multi-Objective Service Coordination Using Deep Reinforcement Learning

Stefan Schneider, Ramin Khalili, Adnan Manzoor, Haydar Qarawlus, Rafael Schellenberg, Holger Karl, and Artur Hecker

***Abstract*—Modern services consist of interconnected components, e.g., microservices in a service mesh or machine learning functions in a pipeline. These services can scale and run across multiple network nodes on demand. To process incoming traffic, service components have to be instantiated and traffic assigned to these instances, taking capacities, changing demands, and Quality of Service (QoS) requirements into account. This challenge is usually solved with custom approaches designed by experts. While this typically works well for the considered scenario, the models often rely on unrealistic assumptions or on knowledge that is not available in practice (e.g., a priori knowledge).**

We propose DeepCoord, a novel deep reinforcement learning approach that learns how to best coordinate services and is geared towards realistic assumptions. It interacts with the network and relies on available, possibly delayed monitoring information. Rather than defining a complex model or an algorithm on how to achieve an objective, our model-free approach adapts to various objectives and traffic patterns. An agent is trained offline without expert knowledge and then applied online with minimal overhead. Compared to a state-of-the-art heuristic, DeepCoord significantly improves flow throughput (up to 76 %) and overall network utility (more than 2x) on real-world network topologies and traffic traces. It also supports optimizing multiple, possibly competing objectives, learns to respect QoS requirements, generalizes to scenarios with unseen, stochastic traffic, and scales to large real-world networks. For reproducibility and reuse, our code is publicly available.

***Index Terms*—Network and Service Management, Reinforcement Learning, Self-Learning, Self-Adaptation, Multi-Objective**

I. Introduction

Service provisioning and coordination in networks with geographically and topologically distributed compute nodes is an ongoing challenge [1], [2]. In edge and fog computing, this challenge is exacerbated by limited compute capacities as well as link delay between the nodes [2]. Furthermore, service demand in terms of incoming flows is also distributed across the network and varies over time. Services can consist of multiple interconnected components, which process incoming flows. Examples are microservices in a service mesh [1], chained virtual network functions (VNFs) in network function virtualization (NFV) [3], or machine learning functions in a pipeline [4]. Each component can run on any node in the network and scale flexibly, by starting/stopping additional instances, according to the current demand. Service coordination requires online decisions on how to scale and where to instantiate each instance as well as how to schedule incoming flows to these instances.

While service coordination has been the focus of intensive study [2], [5], most existing work (detailed in Sec. II) has three major limitations. First, existing work mostly focuses on long-/medium-term planning how to scale and place service components based on given service deployment requests (e.g., [6]–[9]). In doing so, hard-wired chains of component instances are placed in the network to process all incoming flows. This is problematic as operational reality often diverges from any initial plan. For example, actual service demand by users likely differs from the expected load in a given service deployment request. Hence, scaling, placement, and flow scheduling should be adjusted dynamically and online according to the actual service demand.

Second, existing approaches typically use heuristics or numerical solvers (e.g., [6], [10], [11]) and rely on carefully designed models, tailored to specific scenarios, and build on corresponding assumptions. Applying them to scenarios with slightly different assumptions or new optimization objectives (e.g., for QoS) may again require time-consuming manual adjustments by experts.

Third, these models often rely on information that is not available in practice, such as a priori traffic knowledge. In reality, complete knowledge about incoming traffic is not available instantly or even a priori but only after monitoring, often done periodically (e.g., default 1 min in Prometheus [12]). Within such a monitoring interval, numerous flows may arrive stochastically from users at various ingress nodes and need to be processed by service components even before information about these flows is globally available.

To overcome these limitations, we propose a novel approach for autonomous service provisioning and coordination using model-free deep reinforcement learning (DRL). In our proposed approach, which we call DeepCoord, a centralized DRL agent is trained offline through interaction with the network environment, using its previous actions and experience as feedback. The trained DRL agent then autonomously provisions services online and controls dynamic flow scheduling.

Manuscript received January 28, 2021; revised April 8, 2021; accepted April 26, 2021.

S. Schneider, A. Manzoor, R. Schellenberg, and H. Karl are with the Computer Networks Group at Paderborn University, Germany (email: stefan.schneider@upb.de). H. Qarawlus was with Paderborn University, Germany, and is now with Fraunhofer ISST, Germany. R. Khalili and A. Hecker are with Huawei Technologies Munich Research Center, Germany. We also thank Sven Uthe for his valuable contributions to our first prototype.

This work was supported in part by the German Research Foundation within the Collaborative Research Centre “On-The-Fly Computing" (SFB 901), the German Federal Ministry of Education and Research under Software Campus grant 01IS17046 (RealVNF), and the European Commission under 5G-PPP project FUDGE-5G (H2020-ICT-42-2020 call, grant 957242). The expressed views are the authors’ and do not necessarily represent these projects.

For scalability, the agent does not decide scheduling of each flow individually. Instead, it selects and periodically updates scheduling rules that are deployed at each network node in a distributed fashion and applied purely locally at runtime to incoming flows.

Practical application of DRL is known to be challenging and requires careful design of the corresponding observations, actions, and reward as well as integration into the overall system [13]. To address these challenges, we formulate a partially observable Markov decision process (POMDP) with observations based on realistically available monitoring data that is only intermittently available and that contains only aggregated, slightly delayed, and uncertain information. In particular, DeepCoord can be used without expert knowledge and it requires neither a priori traffic knowledge nor detailed knowledge of the network or involved services. We use continuous actions to allow fine-grained online control of service scaling and placement as well as for dynamic flow scheduling. We design the reward function to support multiple objectives (e.g., QoS) based on available monitoring data. Finally, we propose a framework architecture for integrating DeepCoord with a given network environment through custom adapters and publish an open-source prototype [14]. While DeepCoord is trained in a centralized, offline fashion, the trained agent can handle rapidly arriving stochastic and bursty traffic, generalizes to new and unseen scenarios, and scales to large, real-world network topologies while making decisions within milliseconds.

This paper is an extension of our previous work [15]. In this extended version, we consider QoS requirements in terms of acceptable end-to-end delay (soft and hard deadlines) as well as limited link capacities in addition to nodes' compute capacities, which further restrict the solution space and make service coordination more challenging. We show that DeepCoord adapts to these additional constraints through self-learning without human adjustments. We also improve the support for optimizing multiple objectives. This is challenging since common objectives are often conflicting. For example, reducing end-to-end delay requires processing traffic close to its ingress node, but maximizing throughput requires balancing traffic across the entire network to utilize all available resources. Hence, we propose different options for navigating such trade-offs, including objective weighting or custom utility functions, and present new objective formulations for improving QoS and reducing costs and energy. We illustrate the benefits and trade-offs of these objectives in our extensive evaluation on real-world network topologies. Overall, our contributions are:

- We define the problem of online service provisioning and coordination based on available monitoring information. We propose different methods and example formulations for optimizing multiple, possibly opposing objectives.
- We address the service coordination problem by formalizing the corresponding POMDP, which we use for DeepCoord, our novel, self-learning DRL approach based on deep deterministic policy gradient (DDPG) [16].
- DeepCoord consistently outperforms existing approaches, requiring much fewer resources to ensure high success rates on real-world network topologies. It also generalizes to unseen traffic patterns, learns to optimize multiple objectives, and scales to networks of realistic size while only relying on information that is available in practice. Specifically, we observe that DeepCoord reaches up to 76 % more successful flows and more than 2x higher total utility when optimizing multiple objectives.
- For reuse and reproducibility, we make our code publicly available [14].

II. Related work

Service coordination is relevant in a variety of use cases (e.g., cloud or edge computing and NFV) and has been addressed by many researchers. Mann [17] and Hong and Varghese [2] survey service coordination approaches for cloud computing and edge computing, respectively. Herrera and Botero [5] provide an overview of service coordination approaches in the context of NFV. Our proposed DRL-based service coordination approach is not limited to any specific use case but can be applied across different scenarios, e.g., in cloud or edge computing and NFV. In this section, we discuss related work, first focusing on conventional approaches without DRL, e.g., heuristics or mixed-integer linear programs, in Sec. II-A. In Sec. II-B we compare existing approaches that, similar to our proposed approach, use DRL for self-learning service coordination.

A. Conventional Approaches without DRL

The large majority of existing research builds on conventional approaches like heuristics or solving mixed-integer linear programs. Authors often rely on unrealistic a priori knowledge and focus on a subset of our considered service coordination problem, which includes online service scaling and placement as well as runtime scheduling of incoming flows. For example, multiple authors [6]–[9] place services offline, assuming full a priori traffic knowledge. Other authors [10], [11], [18] consider online service scaling and placement but disregard runtime scheduling of flows.

There are many, slightly varying definitions of "scheduling" in related work [7], [19]–[25]. Here, we define flow scheduling as a runtime decision, where (i.e. at which instance) to process an incoming flow according to its processing needs. Similarly, multiple authors [22], [24], [25] consider runtime scheduling of incoming flows but assume a given fixed placement. More related to our approach, Zhang et al. [21] consider joint online scaling and placement of services as well as flow scheduling. Such joint coordination is important to successfully balance trade-offs [26], [27].

While all aforementioned approaches work well in their considered scenarios, each approach is tightly bound to its underlying assumptions. As we show in our evaluation (Sec. V), applying such conventional approaches to scenarios with slightly different assumptions (e.g., different, stochastic traffic patterns) can easily lead to significantly decreased performance. Adapting to new assumptions requires time-consuming manual adjustments by experts. In contrast, our self-learning and self-adaptive, model-free DRL approach learns by itself to

jointly scale, place, and schedule in varying scenarios without any prior knowledge.

In recent years, multiple authors proposed machine learning for predicting required resources [28]–[30] or upcoming traffic demands [31], which can be combined with online heuristic algorithms for pro-active service coordination [32], [33]. We see this as a promising, complementary research direction that could be combined with DRL in future work.

B. Self-Learning Approaches with DRL

Luong et al. [34] survey recent self-learning DRL approaches for networking. Related to our work, multiple authors address online service placement under changing load [35]–[38], some considering stochastic traffic and QoS [37]. In contrast to our approach, Pei et al. [35] rely on a simulated copy of the network to quickly test, evaluate, and revert different actions to ultimately choose the best one in each time step. Furthermore, the final coordination decisions are made by a separate heuristic. Similarly, Solozabal et al. [39] rely on combining their DRL approach with a heuristic and restrict their approach to service placement in networks forming a star topology. Wang et al. [36] assume up-front traffic knowledge per time step and equal flow scheduling between all instances, which could lead to high end-to-end delays and bad service quality. In contrast to our work, Xiao et al. [37] and Quang et al. [38] process incoming flows sequentially, making individual decisions per flow and service component. To address resulting infeasible or sub-optimal placements, the authors roll back and undo previous decisions [37] or apply a separate heuristic algorithm to fix the DRL agent's proposed solution [38]. Making such expensive decisions per flow would not work in our problem, where we consider rapidly arriving flows and only delayed, aggregated, and partially observed network state.

Instead of per-flow decisions and similarly to how we schedule incoming flows, Xu et al. [40] assign different portions of traffic to different paths. However, they focus on traffic engineering and do not consider service scaling and placement. The authors further assume traffic knowledge ahead of each time step and assist their DRL agent with a heuristic. Nasir and Guo [41] do consider delayed and partial observations but focus on power allocation in wireless networks.

The recent work by Gu et al. [42] is most related to our work. Like us, they consider service coordination with flow scheduling, optimize a generic network utility, and build on DDPG [16], which we shortly describe in Sec. IV-C. Still, there are three main differences: 1) Gu et al. focus on compute costs but neglect resource and communication constraints such as node and link capacities and link delays. In contrast, we consider such constraints to support QoS optimization and scenarios with limited resources (cf. edge computing). 2) They assume traffic knowledge ahead of each time step. 3) They support their DRL approach with a custom heuristic to help with action selection and exploration. The authors show that both too much or too little support by the heuristic degrades performance.

Overall, to the best of our knowledge, our work is the first to consider online service coordination in realistic scenarios with rapidly arriving flows and delayed, only partially observed network state. Our model-free approach works without support from a heuristic, which makes it more versatile and less error-prone. We hence believe our approach to be much better applicable to real-world systems than existing work, especially since we do not rely on a priori information.

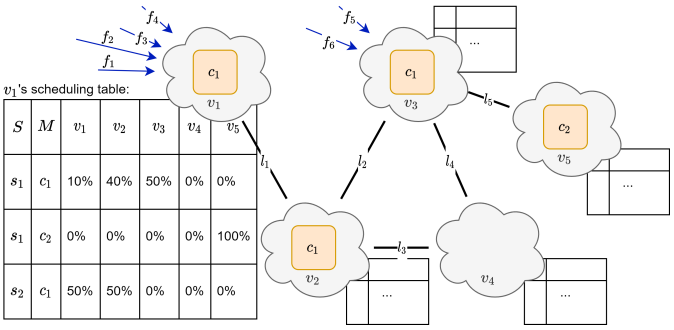


Fig. 1: DeepCoord periodically updates scheduling tables that are deployed at each node in a distributed fashion. Flows continuously arrive at ingress nodes and are scheduled according to these scheduling tables at runtime.

III. Problem Statement

We consider the problem of online service provisioning and coordination in a network of geographically distributed nodes. Here, we intend to precisely define the problem's parameters, decision variables, and objectives but not to provide a full mathematical formulation (e.g., as mixed-integer linear programs). Note that our DRL approach does not require explicit knowledge of the full, detailed network state described here. Instead, it observes only partial and delayed information that is available via monitoring (defined in Sec. IV-B) and implicitly learns to adapt to a given scenario through feedback from its actions.

A. Problem Parameters

The network $G = (V, L)$ consists of nodes V and links L as shown in Fig. 1. Each node $v \in V$ has a compute capacity $\text{cap}_v \in \mathbb{R}_{\geq 0}$. We consider a single generic resource (e.g., CPU), which can be extended easily to multiple resource types. Each link $l \in L$ interconnects two nodes bidirectionally and has a maximum data rate $\text{cap}_l \in \mathbb{R}_{>0}$, which is shared in both directions. It also has a delay $d_l \in \mathbb{R}_{>0}$ that depends, among other factors, on the distance between the connected nodes.

Traffic arrives as many short-lived flows at ingress nodes in the network (for example, f_1–f_6 in Fig. 1), e.g., representing users or sensors requesting a service. Any node can be an ingress node. Each flow $f = (s_f, v_f, t_f, \lambda_f, \delta_f) \in F$ is defined by the service s_f it requests, the ingress node v_f where it arrives, its time of arrival t_f, its requested data rate λ_f, and its duration δ_f. There can be multiple services available in the network, where S is the set of all available services. Each service $s \in S$ consists of a chain of components specified by vector $C_s = \langle c_1, ..., c_{n_s} \rangle$. Services may share components (e.g., s_1 and s_2 both use c_1 in Fig. 1). Furthermore, a service s has

a set of QoS requirements Θ_s, which need to be met to ensure good service quality. As examples for such QoS requirements, we consider $\Theta_s = \{d_s^{\text{soft}}, d_s^{\text{hard}}\}$ with optional soft and/or hard deadlines $d_s^{\text{soft}}, d_s^{\text{hard}} \in \mathbb{R}_{\geq 0} \cup \varnothing$. Flows requesting service s must complete before hard deadline d_s^{hard}, defined relative to flow arrival t_f, or otherwise are dropped automatically. Achieving an end-to-end delay within soft deadline d_s^{soft} is desirable for best service quality but not imperative for flow success.

Service components can be instantiated at multiple different nodes; all instances process flows independently of each other. Set C contains all available components of all services. A flow requesting service s is considered to complete successfully if it traverses instances of all components in C_s in the specified order and within deadline d_s^{hard} (if $d_s^{\text{hard}} \neq \varnothing$).

B. Decision Variables and Network State

We consider scaling and placing services $s \in S$ and scheduling incoming flows $f \in F$ to component instances of the requested service over time T. To this end, we define two decision variables $x_{c,v}(t)$ and $y_{f,c}(t)$. Binary variable $x_{c,v}(t) \in \{0, 1\}$ indicates whether an instance of component c is placed at node v at time t (placement). Instances can be placed at no, one, or multiple nodes (scaling). Variable $y_{f,c}(t) \in V$ indicates at which node $v \in V$ to process a flow f requesting component c of service s_f at time t (scheduling). Our approach centrally controls scaling, placement, and scheduling rules, which are applied locally at each node. We explain details of how we set $x_{c,v}(t)$ and $y_{f,c}(t)$ in Sec. IV-A.

In line with the current serverless trend, we do not explicitly consider intra-node scaling and placement. Instead, we assume that *within* a node v, which may be anything from a single machine to a large data center, the node's operating system or systems like Kubernetes [43] and ElasticNFV [44] start and scale instances of a component c transparently if $x_{c,v}(t) = 1$.

Utilization of node and link resources, i.e., $r_v(t)$ and $r_l(t)$, depend on decisions $x_{c,v}(t)$ and $y_{f,c}(t)$ as well as flow length δ_f and requested data rate λ_f. Consequently, $r_v(t)$ increases with the total data rate of flows processed by instances at v and $r_l(t)$ depends on the total data rate of flows forwarded along link l. Flows are dropped if they cannot be processed or forwarded, e.g., because the resources of a selected node v or link l are already fully utilized or because there is no instance of requested component c. Depending on $x_{c,v}(t)$, component c may not be available at a node v when a flow arrives to be processed there. At this point, fetching, installing, and starting an instance of c on demand would take considerable time, in which the flow would likely already expire.

We further assume that a monitoring system collects and synchronously reports metrics about each node $v \in V$ in fixed intervals of $\Delta > 1$ time steps. As many flows may arrive within Δ, the monitoring system only reports aggregated information over the last interval Δ but no per-flow details. In particular, we assume the monitored network state to merely include the number and aggregated rate of incoming, processed, and dropped flows at v and the average end-to-end delay of completed flows d_{avg} in the last Δ time steps (from $t - \Delta$ to t). This is in contrast to most related work, which assumes complete per-flow and often even a priori knowledge in every single time step.

C. Objectives

We optimize the long-term utility U_T over all T time steps that reflect the desired coordination goals. Utility U_T may consist of a single optimization objective $U_T = o_j$, a weighted (by w_j) sum $U_T = \sum_j w_j o_j$ of multiple objectives, or a more complex custom utility function based on one or multiple objectives. In our evaluation (Sec. V), we consider examples of all three cases, detailed next.

A useful example for a *single optimization objective* is $U_T = o_f$ where o_f is the total amount of successful vs. dropped flows over T time steps:

$$o_f = \frac{F_{\text{succ}} - F_{\text{drop}}}{F_{\text{succ}} + F_{\text{drop}}} \in [-1, 1] \tag{1}$$

It encourages more successful flows F_{succ}, fewer dropped flows F_{drop}, and thus higher throughput.

In practice, operators are often interested in optimizing multiple, possibly competing objectives at once (e.g., for QoS, energy, costs). Here, the *weighted sum of objectives* $U_T = \sum_j w_j o_j$ is useful. We consider three additional objectives, each conflicting with o_f:

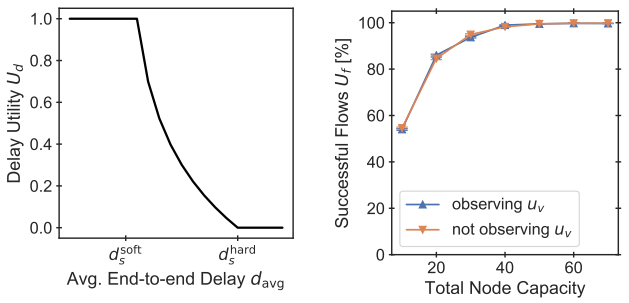
$$o_i = 2 \cdot \frac{-\sum_{c \in C, v \in V} x_{c,v}(t)}{|C| \cdot |V|} + 1 \in [-1, 1] \tag{2}$$

$$o_n = 2 \cdot \frac{-\sum_{v \in V} \mathbb{1}_{\{\sum_{c \in C} x_{c,v}(t) \geq 1\}}}{|V|} + 1 \in [-1, 1] \tag{3}$$

$$o_d = \max\left\{-1, \frac{-d_{\text{avg}}}{D} + 1\right\} \in [-1, 1] \tag{4}$$

Objective o_i minimizes the total number of placed instances (indicated by $x_{c,v}(t)$) to minimize costs (e.g., for licensing). Objective o_n minimizes the number of compute nodes used for at least one instance; $\mathbb{1}_{\{\sum_{c \in C} x_{c,v}(t) \geq 1\}}$ is 1 if an instance is placed at node v and 0 otherwise. This may reduce costs and energy consumption if unused nodes are turned off (e.g., in an edge scenario). Objective o_d minimizes the average end-to-end delay d_{avg} per completed flow in time T for better QoS. If no flow is successful, d_{avg} is undefined and we set $o_d = -1$. To ensure all objectives are in the same range $[-1, 1]$, we scale them by dividing by the respective maximum value. In o_d, we normalize d_{avg} with network diameter D in terms of delay. We further add 1 and cap any values below -1, which may occur if flows traverse the entire network multiple times back and forth due to bad service coordination (i.e., $d_{\text{avg}} > D$). We investigate trade-offs between these objectives and o_f as well as the impact of weights w_j in Sec. V-D. Objectives o_f, o_i, o_n, and o_d are examples for typical optimization goals, but it is also possible to choose and optimize other objectives based on the desired goals and available monitoring information.

Finally, as third and most generic option, we allow defining a *custom utility function* U_T based on any available monitoring information. This allows operators to define arbitrary, complex utility functions based on their business insights. As an example, we consider a custom utility function $U_T = U_f \cdot U_d \in [0, 1]$,



(a) Custom Utility Function (b) Impact of Observing Node Load

Fig. 2: a) DeepCoord supports complex custom utility functions, e.g., for meeting soft deadlines. b) Delayed observations of node load u_v are not useful for DeepCoord.

where $U_f = \frac{F_{\text{succ}}}{F_{\text{succ}} + F_{\text{drop}}} \in [0, 1]$ is the flow success ratio and $U_d \in [0, 1]$ the delay utility shown in Fig. 2a. The delay utility is maximal for an end-to-end delay within the soft deadline d_s^{soft} and then gradually diminishes with increasing delay up to hard deadline d_s^{hard}. In Sec. V-D2, we show that DeepCoord also learns to optimize such a custom utility function to meet soft deadlines for optimal QoS.

IV. DRL Approach: DeepCoord

We propose DeepCoord to address service coordination using model-free DRL. DeepCoord does not know the network topology, link delays, service or per-flow details. Instead, it relies on aggregated yet incomplete, slightly delayed, and uncertain information about incoming flows available through periodic monitoring (updated every Δ time steps). It learns service coordination without expert knowledge from its own experience.

We describe our service coordination approach in Sec. IV-A and formalize a POMDP in Sec. IV-B. Sec. IV-C outlines our DRL framework and algorithm.

A. Joint Scheduling, Scaling, and Placement

We design our approach to work for realistic, dynamic networks with many rapidly arriving flows. Hence, making per-flow coordination decisions centrally at the DRL agent would be highly inefficient and not scalable for large numbers of flows. Moreover, it would require up-to-date, per-flow knowledge, which is not available centrally. Instead, we schedule incoming flows at each node immediately according to rules that are installed at all nodes in the network. DeepCoord updates these rules every Δ time steps, whenever new monitoring data becomes available.

To this end, we introduce *scheduling tables* for each node that indicate where incoming flows should be processed (similar to AWS traffic dials [45] but with dynamic rather than fixed quotas). As illustrated in Fig. 1, each scheduling table contains entries for every service $s \in S$ and every corresponding component $c \in C_s$ (here, $S = \{s_1, s_2\}$, $C_{s_1} = \langle c_1, c_2 \rangle$, $C_{s_2} = \langle c_1 \rangle$). The table entries specify at which destination node to process c by means of a probability distribution over all nodes (not just neighbors). For example in Fig. 1, incoming flows at node v_1 requesting component $c_1 \in C_{s_1}$ of s_1 are scheduled according to the probabilities in v_1's scheduling table. Here, each flow is processed locally at v_1 with 10 % probability, scheduled to be processed at v_2 with 40 %, and scheduled to v_3 with 50 %. Flows belonging to s_1 that finish processing c_1 at v_1 and are then requesting c_2 are all scheduled to v_5. We assume shortest path routing between nodes, e.g., from v_1 to v_5. The same component c_1 also appears in service s_2, where it could require different scheduling. Hence, we consider separate scheduling entries for different services in S. We assume service set S to be rather static and contain all available services, even if they are not currently in use. Scheduling table entries for unused services are simply ignored. If set S does change, e.g., because a completely new service s is released, the scheduling tables and DeepCoord's neural networks have to be restructured to include s. In principle, the learned weights for $S \setminus s$ could be retained in the restructured neural networks.

By deploying these scheduling tables at each node, incoming flows are scheduled immediately (in $O(\log |V|)$) at runtime according to these probabilities. That means, $y_{f,c}(t)$ is set to v_i with probabilities given for v_i and $c \in C_{s_f}$ in a distributed manner. After deciding a destination node for processing a flow according to the scheduling probabilities, the entire flow is sent there. Using separate scheduling tables for each node allows to schedule flows differently depending on where they arrive in the network. In doing so, flows can be scheduled to close-by nodes, reducing end-to-end delay.

We also derive variable $x_{c,v}(t)$ (scaling and placement) from the scheduling tables but update it only every Δ time steps. To avoid dropped flows, we ensure that instances of component c are available at every node v to which flows may be scheduled. Specifically, we start at the ingress nodes with the first component $c_1 \in C_s$ for each service s and set $x_{c_1,v}(t) = 1$ if there is a non-zero probability for $y_{f,c_1}(t) = v$ based on the scheduling tables. Since scheduling probabilities sum up to one, we always place at least one instance per service component to process incoming flows. We then continue in a similar fashion for the next component c_2, checking the scheduling tables of the nodes where we placed instances of c_1. Based on v_1's scheduling table in Fig. 1, we would set $x_{c_1,v_1}(t) = x_{c_1,v_2}(t) = x_{c_1,v_3}(t) = x_{c_2,v_5}(t) = 1$. Following this approach, we can jointly decide flow scheduling probabilities, scaling, and placement by periodically (every Δ time steps) updating the scheduling tables for all nodes. In practice, these updates could be done consistently across the network using SDN technology [46].

B. MDP with Partial Observability

In real networks, the full network state is huge and can only be observed partly through monitoring. Hence, we design a POMDP to create and periodically update the scheduling tables as described in Sec. IV-A. Using a POMDP is novel compared to related DRL approaches, which mostly assume a fully observable MDP (see Sec. II). In a POMDP, an agent interacts with an environment to obtain rewards, which

allows the agent to learn highly-rewarded behavior. Formally, a POMDP $(\mathcal{O}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ is defined by observation space $\mathcal{O}$, i.e., parts of the full network state, the agent's action space $\mathcal{A}$, the environment dynamics $\mathcal{P}$, which are typically unknown, and the reward function $\mathcal{R}$. In our approach described in Sec. IV-A, the agent interacts with the environment every Δ time steps. It receives observations from the last Δ time steps (e.g., through monitoring), applies an action to update the scheduling tables for the next Δ time steps, and, after these Δ time steps, receives a reward together with the next observation. We define $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{R}$ as follows.

Observations $\mathcal{O} = \langle \lambda_{v,s} | v \in V, s \in S \rangle$, where $\lambda_{v,s}$ is the data rate summed up over all flows arriving at ingress v and requesting service s, averaged over the previous interval of length Δ. If v is not an ingress, $\lambda_{v,s} = 0$.

In our previous work [15], DeepCoord also observed node load $u_v = \frac{r_v(t)}{\text{cap}_v} \in [0, 1]$ at v over the last Δ time steps ($u_v = 1$ if $\text{cap}_v = 0$). However, we found that this observation usually does not improve DeepCoord's performance (see Fig. 2b). In fact, the agent can completely change all scheduling rules with a single action, such that the observed node load u_v from the previous interval is no relevant indication for the node load in the next interval, even with constant ingress data rates. Even without this observation, the agent implicitly still learns about the network's capacities. It is rewarded for successful flows and punished for dropped flows when scheduling to nodes or links with insufficient capacity. Thus, we do not include u_v in observations $\mathcal{O}$ here to avoid unnecessary complexity for DeepCoord and further reduce our requirements regarding monitoring.

Actions $\mathcal{A} = \langle p_{v,s,c,v'} | v, v' \in V, s \in S, c \in C_s \rangle$, where $p_{v,s,c,v'} \in [0, 1]$ is the probability for scheduling a flow arriving at node v, requesting component c of service s to be processed at node v'. This results in a probability distribution with $\sum_{v' \in V} p_{v,s,c,v'} = 1$. As different scheduling probabilities are explored in the POMDP, it is unlikely that probabilities are set to exactly 0 %. To avoid sending small fractions of traffic to many nodes, we further process these probabilities as follows. We introduce a threshold p_{thres} and set all probabilities $p_{v,s,c,v'} < p_{\text{thres}}$ to 0 during post-processing. We then normalize each scheduling table row to ensure that the probabilities again sum up to 1, i.e., flows are still scheduled and processed at nodes other than v'. Finally, we apply these processed scheduling tables to the network and also use them for training (see Sec. IV-C).

Reward $\mathcal{R} = U_\Delta$. Here, U_Δ can correspond to any of the three types of utility functions defined in Sec. III-C, i.e., optimizing either an individual objective, a weighted sum of multiple objectives, or a custom utility function. In either case, the reward signal is computed based only on metrics collected in the last Δ time steps. In the example of $U_\Delta = o_f$, U_Δ would only consider the successful and dropped flows in the last monitoring interval Δ, which are most affected by the previous action. Internally, DeepCoord maximizes the sum of discounted rewards to optimize long-term utility. We evaluate DeepCoord with different utility functions for optimizing varying objectives in Sec. V-D.

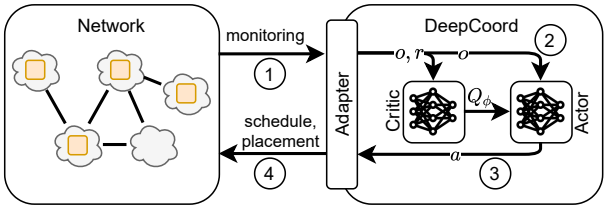


Fig. 3: In our proposed framework, the actor-critic DRL agent iteratively interacts with the network through an adapter.

C. DRL Service Coordination

DeepCoord is based on DDPG [16], which can handle large, continuous action spaces such as $\mathcal{A}$ in our POMDP, unlike previous algorithms like deep Q-learning [47]. DDPG is an off-policy actor-critic algorithm, i.e., it learns from buffered batches of previous experience using neural networks for both actor μ_θ and critic Q_ϕ. The critic approximates the long-term value $Q_\phi(o, a)$ of action a after observation o based on immediate reward r and expected future rewards. Critic Q_ϕ is used to train actor μ_θ. Actions produced by μ_θ represent the probabilities of each node's scheduling table, which should sum up to 1 for each row (Sec. IV-B). To this end, we split the output layer of μ_θ into separate parts for each row in each scheduling table and apply the softmax activation separately.

To ensure fast, consistent, and good service coordination, we first train DeepCoord offline until convergence and then apply the trained agent online (inference). Fig. 3 shows our framework designed for training and applying DRL for service coordination. The network provides monitoring data in regular intervals (step 1). In step 2, an adapter processes the monitoring information, retrieving the relevant observation o and calculating reward r for the previous interval as described in Sec. IV-B. This architecture allows us to connect DeepCoord to different kinds of networks or monitoring systems, simply by implementing a new adapter. In step 3, o and r are used to train critic Q_ϕ and actor μ_θ and to choose the next action a as defined in Sec. IV-B. The adapter uses a to compute the final scheduling tables for all nodes, derives the scaling and placement, and applies it to the network (step 4).

Alg. 1 shows the resulting DRL algorithm for training and inference. DDPG is known for its high training variance depending on the random seed [48]. To mitigate this problem, we propose to train k DRL agents in parallel with different random seeds, e.g., one per available CPU core (ln. 1–2). After training, the best agent can be selected automatically based on the achieved reward. During training, new experience is added to the buffer B and batches b of size N are sampled to train critic and actor (ln. 5–9). For training stability, target critic $Q_{\phi'}$ and actor $\mu_{\theta'}$ are updated slowly according to τ (ln. 10–11). Then, the next action a is selected using the trained actor and adding Gaussian noise $\mathcal{N}$ to encourage exploration (ln. 12). Finally, a is post-processed as described in Sec. IV-A and IV-B to derive the final scheduling, scaling, and placement decisions (ln. 13). We store the processed and actually applied actions in buffer B for training. After training, we use the best trained agent for fast inference during online service coordination (ln. 14). New observations are directly passed to

Algorithm 1 DeepCoord Training and Inference

1: $k \leftarrow$ num. CPU cores available for training ▷ Training
2: **for** k DRL agents in parallel **do**
3: initialize $\mu_\theta, \mu_{\theta'}, Q_\phi, Q_{\phi'}, B$
4: **for all** Δ time steps $\in T$ **do**
5: $o, r \leftarrow$ adapter.process(monitoring)
6: $B \xleftarrow{\text{add}} (o_{\text{prev}}, a_{\text{prev}}, r, o)$
7: $b \leftarrow \text{sample}(B, N)$
8: train Q_ϕ minimizing the Bellman error [16]
9: train μ_θ maximizing $\mathbb{E}_o[Q_\phi(o, \mu_\theta(o))]$
10: $Q_{\phi'} \leftarrow \tau Q_\phi + (1-\tau) Q_{\phi'}$
11: $\mu_{\theta'} \leftarrow \tau \mu_\theta + (1-\tau) \mu_{\theta'}$
12: $a \leftarrow \mu_\theta(o) + \mathcal{N}$
13: network $\xleftarrow{\text{apply}}$ adapter.process(a)
14: select best trained agent (μ_θ, Q_ϕ) ▷ Inference
15: **for all** Δ time steps $\in T$ **do**
16: $o, r \leftarrow$ adapter.process(monitoring)
17: $a \leftarrow \mu_\theta(o)$
18: network $\xleftarrow{\text{apply}}$ adapter.process(a)

the trained actor μ_θ to obtain the next action (ln. 17). For best performance during inference, we do not add noise but exploit the best action. The selected action is then post-processed and applied to the network as before (ln. 18).

Offline training of DeepCoord is time-intensive and depends on random exploration. In contrast, online inference is very fast [49]. Its complexity is defined by the matrix multiplication of the observations and neural network weights, which depend on observation and action space (Sec. IV-B). We empirically evaluate training and inference complexity for varying network sizes in Sec. V-E.

D. Implementation and Deployment

We implemented DeepCoord using Python and published it in an open-source repository [14]. DeepCoord's actor and critic neural networks are built with Keras [50] and TensorFlow [51]. DeepCoord follows the common OpenAI Gym interface [52] and interacts with the network through an adapter that collects and computes the required observations and the reward. The adapter implementation depends on the specific network environment. For training and evaluation, we use a lightweight open-source simulator [53]. For production deployment, systems like Prometheus [12] and Kubernetes [43] could be interfaced by the adapter for centralized monitoring and orchestration. Scheduling rules could be installed and applied in a distributed fashion at all nodes using SDN [54]. The location of the orchestrator and SDN controller could be optimized using established approaches [55], [56] to minimize the overhead of collecting monitoring data and updating scheduling rules in very large scenarios. Moreover, we discuss a hierarchical approach for scalable coordination of very large, multi-domain networks in our related paper [57]. We leave further investigation in this direction for future work.

V. Numerical Evaluation

We evaluate DeepCoord through extensive simulations using real-world network topologies and realistic traffic patterns. We describe the details of our evaluation setup in Sec. V-A. In Sec. V-B, we evaluate how well DeepCoord self-adapts to scenarios with varying load, traffic patterns, node and link capacities, and QoS requirements. For each scenario, we train DeepCoord offline from scratch in a training environment and then evaluate the trained agent in a separate testing environment with different random seeds. The agent adapts to each scenario through self-learning without human intervention or expertise, simply from experience when interacting with each environment. Sec. V-C investigates DeepCoord's generalization capabilities, where it is trained on one scenario and then tested on other new and unseen scenarios without any additional training. While the aforementioned experiments focus on optimizing a single optimization objective, Sec. V-D explores trade-offs when optimizing multiple competing objectives. Finally, Sec. V-E evaluates the scalability of DeepCoord to large, real-world network topologies.

A. Evaluation Setup

1) Evaluation Scenarios: We perform extensive simulations on real-world network topology Abilene [59], which connects nodes at 11 cities across the United States. In Sec. V-E, we also evaluate scalability on three larger real-world network topologies from Europe, China, and across continents, taken from the Internet Topology Zoo [59]. These topologies contain information about the position and interconnection of the network nodes but not about their type (e.g., data center or small edge server) nor about their compute capacity. Since DeepCoord does not distinguish between different node types and is only affected by a node's capacity cap_v, we assign heterogeneous node capacities cap_v between 0 and 2 compute units (e.g., CPU cores) uniformly and independently at random. By default, we use very high link capacities $\text{cap}_l = 1000$ but also consider scenarios with more restricted link capacities (Sec. V-B2). Link delays are based on the distance between connected nodes. While we successfully tested our approach with multiple services, for simplicity, we here focus on and show the results of coordinating a single service s with components $C_s = \langle c_{\text{IDS}}, c_{\text{proxy}}, c_{\text{web}} \rangle$. Instances of each component require resources that increase linearly with increasing total data rate of flows to process [60]. We assume all flows requesting this service to have unit data rate ($\lambda_f = 1$) and flow length ($\delta_f = 1$) but consider scenarios with increasingly complex (and realistic) flow arrival patterns.

In our evaluation, flows arrive over $|T| = 20000$ time steps according to different traffic patterns at the network's ingress nodes. Ingress nodes are selected randomly per network and do not change over time. We further set $\Delta = 100$ time steps, after which DeepCoord receives new observations and applies actions. As described in Sec. IV, this means that information in observations may be delayed by up to 100 time steps. This is more realistic than the common assumption in related work of having up-to-date information at each time step.

2) DRL Hyperparameters: For each scenario, we first train $k = 10$ DRL agents in parallel until convergence (500 episodes). Then, we automatically select the best DRL agent for inference (Sec. IV-C). We train DeepCoord from scratch for each scenario but configure fixed values for all hyperparameters that are used across all scenarios. Thus, no manual adjustments are required for solving different scenarios with DeepCoord.

For both actor and critic, we train dense neural networks with a single fully connected hidden layer (64 nodes, ReLU [61]) using the Adam optimizer [62]. We further configured the following hyperparameters: 1) Discount factor $\gamma = 0.99$. 2) Soft target updates with $\tau = 0.0001$. 3) Learning rate $\alpha = 0.01$ with decay 0.001. 4) Buffer size $|B| = 10000$ with batch size $|b| = 64$. 5) For exploration, we use Gaussian noise with $\mathcal{N}(0, 0.2)$. 6) Threshold $p_{\text{thres}} = 0.1$ (see Sec. IV-B).

3) Baseline Algorithms:

- A heuristic for bidirectional scaling and placement, which we refer to as BSP, from our previous work [11]. BSP jointly optimizes service scaling and placement as well as flow assignment using an iterative destroy-and-repair mechanism.
- Shortest path (SP): For each ingress, SP places exactly one instance per component $c \in C_s$. It follows a simplified first-fit strategy by instantiating the first component at the ingress node and each following component at the neighbor closest to the previous instance. In doing so, SP favors nodes with fewer existing instances and skips nodes without any compute capacity (independent of current utilization).
- Load balance (LB): LB instantiates all components at all nodes with non-zero capacity and schedules flows equally.

SP and LB are similar to the baselines used by Xu et al. [40]. All three algorithms choose actions from action space $\mathcal{A}$—but, unlike DeepCoord, do not learn from these actions.

Applying BSP to our problem directly works poorly. The algorithm assumes that all flows run in parallel and compete for resources, but in our problem, flows arrive sequentially at each ingress and only overlap partially. For a fair comparison, we adjusted the input processing of BSP to estimate the overlapping flows per Δ time steps. This is an example of how built-in assumptions limit the applicability of a model-based algorithm, requiring manual adjustments by experts. We show both the default and adapted version of BSP in our evaluation. Unlike DeepCoord, related DRL approaches (Sec. II) are not available publicly. Thus, a direct comparison is difficult.

4) Execution & Figures: We repeated all experiments with 30 different random seeds on machines with Intel Xeon W-2145 and 32 GB RAM. Each measurement point in the figures of this section (Fig. 4–13) represent the mean over these 30 repetitions. The error bars depict the standard deviation.

B. Self-Adaptation to Varying Scenarios

Here, we focus on maximizing the number of successful flows ($U_T = o_f$, cf. Sec. III-C) in the Abilene network. We systematically vary different problem parameters (traffic, capacities, QoS requirements) and compare the percentage of successful flows after T time steps for each algorithm. We train DeepCoord from scratch for each scenario to evaluate how well it adapts itself to these scenarios.

1) Varying Ingress Nodes and Traffic Patterns: First, we vary the number of ingress nodes from 1 to 5 and choose increasingly complex flow arrival patterns. With more ingress nodes, total traffic increases and the network's capacities become saturated such that flows have to be dropped.

The simplest traffic pattern we consider is *fixed flow arrival*, where flows arrive in fixed intervals (10 time steps) at each ingress. Fig. 4a shows the percentage of successful flows for the different algorithms. As described in Sec. V-A, default BSP performs poorly, but the manually adapted BSP ("BSP Ad.") does much better and only drops flows with more than 3 ingress nodes. SP avoids dropped flows up to 2 ingress nodes and LB always drops many flows. DeepCoord outperforms all other algorithms, processing much more flows successfully in the highly saturated network with 4 and 5 ingress nodes (up to 76 % more successful flows than adapted BSP).

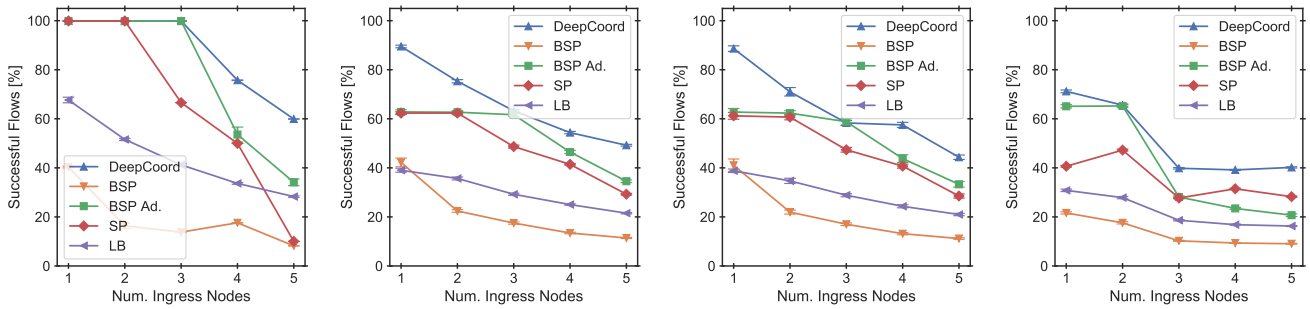
Fig. 4b shows the results for *Poisson flow arrival* (mean inter-arrival time 10 time steps). Due to the randomness in flow arrival, multiple flows may arrive directly after another in bursts, which can easily lead to dropped flows. Again, adapted BSP is slightly better than SP and much better than default BSP. LB still performs worse than the other algorithms. Still, DeepCoord outperforms all algorithms. Compared to adapted BSP, it processes up to 43 % more flows. In particular, it learns to deal with Poisson flow arrival by not fully utilizing all resources of a node but leaving some resources free for handling small bursts.

Next, we consider more realistic flow arrival following a *Markov-modulated Poisson process (MMPP)* [63]. The two-state Markov process randomly switches between flow arrival with mean inter-arrival time 12 and 8 (50 % higher rate) every 100 time steps with 5 % probability. Fig. 4c shows that DeepCoord handles MMPP flow arrival well and, again, outperforms the other algorithms (up to 41 % better than adapted BSP).

Finally, Fig. 4d shows the results for flows following *real-world traffic traces* that were recorded for the Abilene network [64]. To simulate increasing load, we enable an increasing number of ingress nodes where the trace-driven traffic arrives. Here, adapted BSP is better with low load (1–2 ingress nodes) and SP is better with high load (4–5 ingress nodes). DeepCoord handles this real-world traffic well and again outperforms all other algorithms (up to 43 % better than adapted BSP and 45 % better than SP).

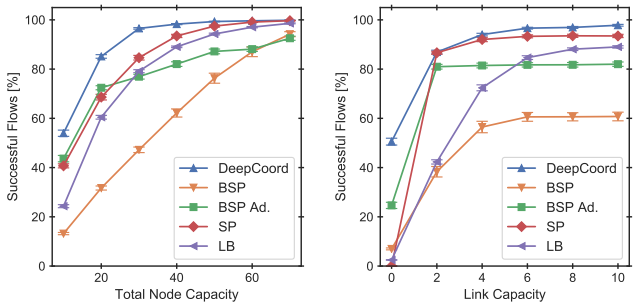
2) Varying Node and Link Capacities: We investigate self-adaptation of DeepCoord to scenarios with varying node and link capacity, again, training from scratch for each scenario. Specifically, we consider MMPP traffic with 4 ingress nodes on the Abilene network, starting with a total node capacity of 10 compute units (similar to Sec. V-B1) and then evenly increasing node capacity. Fig. 5a shows that DeepCoord adapts well to different node capacities and clearly outperforms all other algorithms. In comparison, the other algorithms need at least 67 % more resources to reach high success rates of above 95 %.

Next, we consider scenarios with limited link capacity, varying cap_l from 0 to 10 for each link l. In contrast to all other scenarios, where nodes' compute resources were



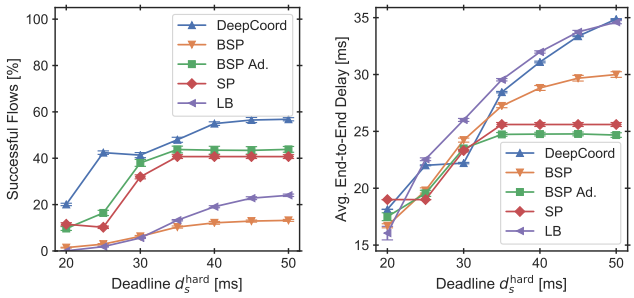
(a) Fixed Arrival (b) Poisson Arrival (c) MMPP Arrival (d) Real-World Traffic Trace

Fig. 4: The figures show the percentage of successful flows for an increasing number of ingress nodes, i.e., increasing load, and increasingly realistic traffic patterns. Compared to other approaches, DeepCoord processes most flows successfully.



(a) Varying Node Capacity (b) Varying Link Capacity

Fig. 5: DeepCoord self-adapts to varying capacities. It leverages increasing node and link capacities to process more flows successfully, outperforming other approaches.



(a) Successful Flows (b) End-to-end Delay

Fig. 6: DeepCoord self-adapts to varying hard deadlines. It leverages higher deadlines to better distribute load and achieve higher flow success rates.

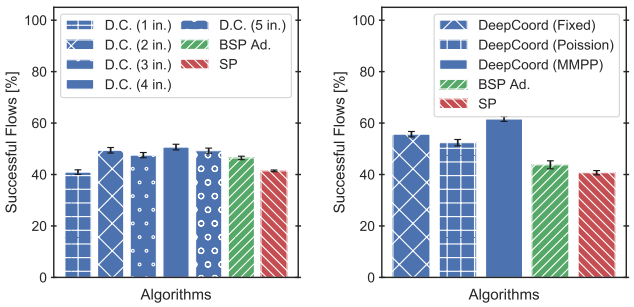
the bottleneck, here, communication between nodes is the bottleneck. Such communication-constraint scenarios are not the main focus of DeepCoord, which uses fixed (shortest) paths when scheduling flows between nodes rather than routing them dynamically. Still, DeepCoord achieves more successful flows than the other algorithms (Fig. 5b) and is the only approach to achieve success rates of more than 95 %. This indicates that, even without dynamic routing, it successfully adapts to varying link capacities by distributing load across different nodes and corresponding paths.

3) Varying QoS Requirements (Hard Deadlines): Here, we explore scenarios with MMPP traffic, 4 ingress nodes, and varying QoS requirements in terms of hard deadlines d_s^{hard}. As flows are dropped automatically if a hard deadline is not met, completing flows that exceed the deadline is not possible. DeepCoord implicitly learns that flows are dropped when their end-to-end delay is too high—without requiring any explicit knowledge about these deadlines. Fig. 6 shows that DeepCoord exploits increasing deadlines by distributing flows to nodes farther away. This leads to an increasing success rate (Fig. 6a) and results in higher end-to-end delay within the allowed deadline (Fig. 6b). Similarly, LB exploits increasing deadlines and tries to balance load across all nodes as far as possible with the given deadlines. Still, it achieves much lower success rates than DeepCoord. In contrast, adapted BSP and SP do not exploit deadlines beyond 35 ms to increase success rates, leading to an increasing gap compared to DeepCoord.

C. Generalization to Unseen Scenarios

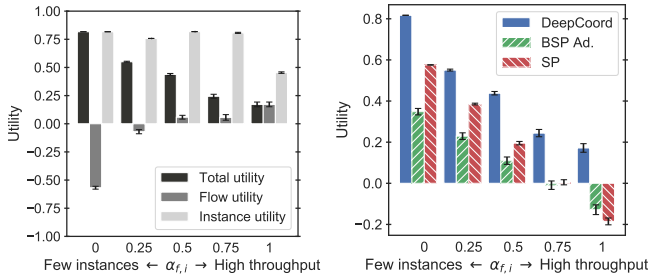
For the different scenarios of Sec. V-B, we always train DeepCoord from scratch but reuse the same hyperparameter settings. This allows to fully automate training and applying DeepCoord to different scenarios. In practice, a trained agent still needs to perform reasonably well when facing a new scenario, e.g., due to changes in load or traffic. Training a new agent optimized for the new scenario can take hours, during which the old agent is still being used. To support such generalization, we define our observations and reward based on generally-available information and normalize observations, actions, and rewards to be in a similar range (Sec. IV-B).

We investigate generalization of DeepCoord to scenarios with unseen load. In particular, we train five different DeepCoord agents on MMPP traffic with 1–5 ingress nodes, respectively, representing scenarios with low to high load. Without any additional training, we then test all five DeepCoord agents on MMPP traffic with 4 ingress nodes. Fig. 7a shows that, as



(a) Unseen Ingress Nodes (b) Unseen Traffic Pattern

Fig. 7: DeepCoord generalizes to new scenarios with previously unseen ingress nodes (a) and traffic patterns (b).



(a) DeepCoord's Total, Flow, and Instance Utility for trade-off $\alpha_{f,i}$ (b) Algorithms' Total Utility

Fig. 8: DeepCoord effectively navigates trade-off $\alpha_{f,i}$ between placing fewer instances and higher throughput.

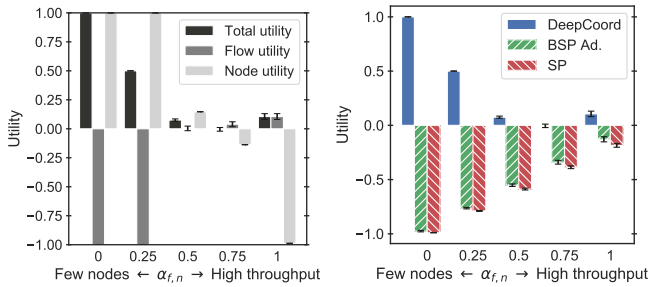
expected, the DeepCoord agent trained and tested on 4 ingress nodes (abbreviated as "D.C. (4 in.)") performs best. However, also the agents trained on 2, 3, and 5 ingress nodes generalize well to unseen traffic from 4 ingress nodes. In fact, they achieve a similar percentage of successful flows as the agent trained and tested on 4 ingress nodes and still outperform adapted BSP and SP. Only the agent trained on a single ingress node leads to slightly worse results when generalizing to much higher load with 4 ingress nodes. Still, its performance is comparable with SP's.

We also study generalization of DeepCoord to scenarios with new traffic patterns. Specifically, we train one agent on fixed flow arrival and another on Poisson flow arrival and confront both with previously unseen MMPP flow arrival. Fig. 7b shows the successful flows for both cases in the Abilene network with 4 ingress nodes. For comparison, we also show results of DeepCoord trained on MMPP traffic and of adapted BSP and SP. The figure indicates that DeepCoord generalizes well to new traffic patterns without significantly reduced successful flows. The generalized agents still outperform adapted BSP and SP.

D. Optimizing Multiple Objectives

In Sec. V-B and V-C, we focus on maximizing successful flows (o_f) as the only optimization objective. In practice, operators often want to optimize multiple objectives. Here, in Sec. V-D1, we evaluate service coordination with multiple objectives, investigating the trade-off between maximizing successful flows and objectives o_i, o_n, and o_d, defined in Sec. III-C. We also consider a more complex, custom objective function for supporting soft deadlines in addition to hard deadlines in Sec. V-D2. In all cases, we consider MMPP traffic with 4 ingress nodes on the Abilene network.

1) Weighted Sum of Objectives: First, we consider utility $U_T = w_f o_f + w_i o_i$ as weighted sum of maximizing successful flows (obj. o_f) and minimizing the number of placed instances (obj. o_i). The two objectives are often conflicting as maximizing successful flows may require balancing the load across more instances. We explore this trade-off by systematically varying $\alpha_{f,i} = w_f = 1 - w_i \in [0, 1]$ and training DeepCoord for each setting. Fig. 8a shows the results in terms of flow utility o_f, instance utility o_i, and total, weighted utility U_T. Clearly, $\alpha_{f,i}$ affects how DeepCoord coordinates services. As desired, higher $\alpha_{f,i}$ leads to better flow utility. At the same time, the agent manages to maintain high instance utility, i.e., placing few instances. Only for $\alpha_{f,i} = 1$, the agent places more instances (i.e., lower instance utility) for even higher flow utility. In the given scenario, there are only enough resources to process some but not all flows successfully ($o_f < 1$) such that the total utility decreases with increasing $\alpha_{f,i}$. Compared to the other algorithms, DeepCoord achieves significantly better total utility for all $\alpha_{f,i}$ values (Fig. 8b), indicating that it learns to navigate this trade-off well.

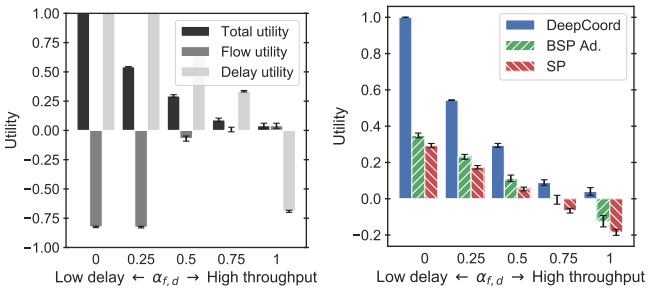


(a) DeepCoord's Total, Flow, and Node Utility for trade-off $\alpha_{f,n}$ (b) Algorithms' Total Utility

Fig. 9: DeepCoord effectively navigates trade-off $\alpha_{f,n}$ between utilizing fewer nodes and achieving higher throughput.

Second, we evaluate the trade-off between maximizing successful flows and minimizing the number of used compute nodes with $U_T = w_f o_f + w_n o_n$ and $\alpha_{f,n} = w_f = 1 - w_n \in [0, 1]$. A compute node can be turned off if no instances are placed there, saving costs and energy. Hence, to turn off a node v and to optimize o_n, DeepCoord has to learn to select actions that do not schedule any traffic (below threshold p_{thres}) to any instance at v. Compared to minimizing the number of instances (obj. o_i), optimizing o_n is more challenging since DeepCoord is not rewarded for removing a single instance as long as there are still other instances placed at the same node. Still, Fig. 9a shows that DeepCoord explores actions effectively and does learn different coordination schemes depending on $\alpha_{f,n}$. With low $\alpha_{f,n}$ (0 or 0.25), DeepCoord drops all flows ($o_f = -1$) in

favor of optimal node utility ($o_n = 1$). As desired, with higher $\alpha_{f,n}$, it learns to process more flows successfully at the cost of utilizing more nodes. Compared to the other algorithms, its overall utility is much higher for all values of $\alpha_{f,n}$ (Fig. 9b). Hence, while $\alpha_{f,n}$ has to be chosen carefully, DeepCoord can successfully optimize both objectives and navigate the trade-off as controlled by $\alpha_{f,n}$.



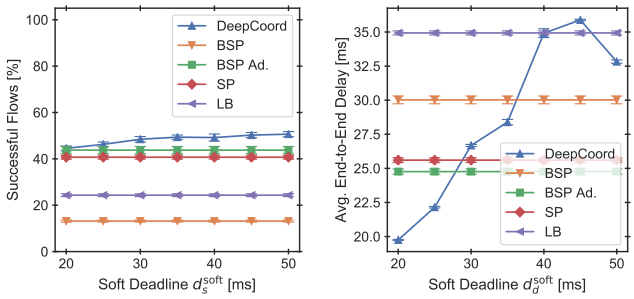
(a) DeepCoord's Total, Flow, and Delay Utility for trade-off $\alpha_{f,d}$ (b) Algorithms' Total Utility

Fig. 10: DeepCoord effectively navigates trade-off $\alpha_{f,d}$ between lower delay (thus, better QoS) and higher throughput.

Finally, we consider the trade-off between maximizing successful flows and minimizing avg. end-to-end delay with $U_T = w_f o_f + w_d o_d$ and $\alpha_{f,d} = w_f = 1 - w_d \in [0, 1]$. Again, o_f and o_d are often opposing objectives as processing more flows successfully may require scheduling them to nodes farther away, i.e., with higher path and end-to-end delay. Again, Fig. 10a shows that DeepCoord learns different coordination schemes corresponding to $\alpha_{f,d}$. As desired, higher $\alpha_{f,d}$ leads to better flow but worse delay utility. With decreasing $\alpha_{f,d}$, it favors shorter delays at the cost of more dropped flows. The agents trained with $\alpha_{f,d} = 0$ and $\alpha_{f,d} = 0.25$ drop most flows but process the remaining successful flows with optimal delay. Again, DeepCoord achieves significantly better total utility for all $\alpha_{f,d}$ values than the other algorithms (Fig. 10b). Overall, DeepCoord learns to effectively optimize multiple, even competing objectives, where the trade-off between these objectives can be controlled conveniently via weights $\alpha_{f,i}$, $\alpha_{f,n}$, and $\alpha_{f,d}$.

2) Custom Utility Function (Soft Deadlines): In addition to optimizing individual objectives or weighted sums of multiple objectives, DeepCoord also supports optimizing custom utility functions. These functions may specify any complex relationship between available monitoring information and resulting utility. As an example, we here consider the custom utility function defined in Sec. III-C. There, the total utility depends on the successful flows and their delay in terms of meeting QoS requirements. The utility is high as long as flows' end-to-end delay is below a given soft deadline and then gradually drops off until a hard deadline is reached (Fig. 2a). Unlike the scenario with hard deadlines in Sec. V-B3, flows are not dropped automatically if they exceed their soft deadline. Hence, when only optimizing objective o_f, DeepCoord would be unaware of and not adapt to these soft deadlines.

Instead, when optimizing the custom utility function, Fig. 11 shows that DeepCoord does successfully learn to take these



(a) Successful Flows (b) End-to-end Delay

Fig. 11: DeepCoord self-adapts to custom utility functions, here the utility function in Fig. 2a for soft deadlines. As a result, it learns to respect soft deadlines and leverages increasing deadlines to improve the flow success rate.

soft deadlines into account. It not only outperforms the other algorithms in terms of successful flows (Fig. 11a) but also adapts to ensure the avg. end-to-end delay stays below the given soft deadline to maximize utility and QoS. In turn, it exploits higher soft deadlines to process more flows successfully (14 % more with $d_s^{soft} = 50$ compared to $d_s^{soft} = 20$). In the scenario here, soft deadlines above $d_s^{soft} = 35$ only enable marginal improvements of successful flows, and further reducing the avg. end-to-end delay below the soft deadline does not improve the utility. Hence, the slightly decreased avg. end-to-end delay for $d_s^{soft} = 50$ compared to $d_s^{soft} = 45$ could be a result of randomized training. The other algorithms do not support QoS optimization with custom utility functions and are unaware of the soft deadlines. Hence, they do not adapt to varying soft deadlines and, overall, process fewer flows successfully than DeepCoord.

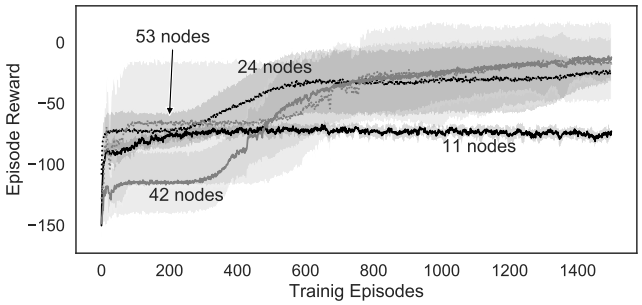


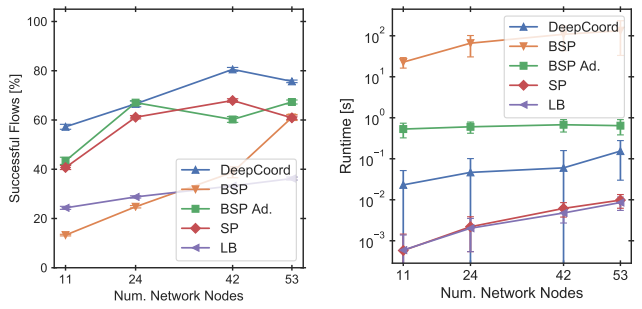
Fig. 12: Learning curves of DeepCoord when training on networks of varying sizes.

E. Scalability

Finally, we evaluate the scalability of our approach to large-scale networks with many nodes. In addition to Abilene (11 nodes), we consider real-world network topologies BT Europe (24 nodes), China Telecom (42 nodes), and TiNet (53 nodes) [59], each with 4 ingress nodes and MMPP flow arrival (as defined in Sec. V-B1).

Fig. 12 shows the learning curves for training DeepCoord offline. The lines show the average episode reward of the $k = 10$ agents during training and the error bands show the standard deviation. As action noise enforces exploration, the reward is much lower and noisier during training than when testing the trained agent. Still, the rapid growth of episode reward within the first 100 episodes indicates that DeepCoord quickly learns a good coordination policy.

The figure also shows that more training may still increase performance significantly, e.g., the reward for 42 nodes leaps around episodes 300 and 600. With much more training, we expect further leaps in performance. Especially large networks require excessive training to explore the large action space and to find an optimal policy. The need for excessive training is not specific to our approach but a well-known problem in deep learning [65]. E.g., DeepMind's famous AlphaGo Zero was trained over almost 5 million games [66]. Due to limited time and resources, we had to restrict training to 1500 episodes (each with $|T| = 20000$). In future work, we will explore recent approaches for more efficient training like distributed DRL [67] and curriculum learning [68].



(a) Successful Flows (b) Algorithm Runtime (log. scale)

Fig. 13: Even in large networks, DeepCoord processes more flows successfully than existing approaches (a) while maintaining short inference runtimes (b).

Despite limited offline training, DeepCoord can compete with or even outperform all baseline algorithms. Fig. 13a compares the algorithms' percentage of successful flows (optimizing only objective o_f). As before, adapted BSP performs comparable to SP and considerably better than the default BSP version. LB performs worse on small networks but processes an increasing number of flows successfully with increasing network size. This is because LB balances traffic equally among all nodes with resources, leading to lower load per node and more successful flows for larger networks. Still, DeepCoord processes as many or even more flows successfully. The difference is especially large for 42 nodes, where there was a particularly large leap in performance during training (see Fig. 12). We believe that considerably more training (e.g., in a commercial setting) could result in similar performance leaps and even better results for 24 and 53 nodes.

In addition to quality, the runtime of online coordination decisions is crucial to quickly adapt to changes. Fig. 13b shows the algorithms' average runtime per coordination decision on a logarithmic scale. While offline training is time- and resource-intensive, applying the trained DRL agent for online inference only requires tens of milliseconds and is much faster than the (adapted) BSP heuristic. Compared to DeepCoord, the simple SP and LB baselines are even faster but at the cost of reduced coordination performance. Overall, DeepCoord scales well to large networks while maintaining reasonable runtimes.

VI. Conclusion

Our DRL approach, DeepCoord, learns quickly during offline training and then autonomously provisions and coordinates services online. In contrast to existing approaches using a priori knowledge for planning, it relies on realistically available, partial and delayed observations with uncertain future traffic and without knowledge of network topology or service structure. It learns without human intervention or expertise and flexibly adapts to different scenarios or optimization objectives, even supporting multiple opposing objectives (throughput, costs, energy, QoS). Hence, we believe our approach is an important step towards truly driver-less, self-learning networks and thus towards higher efficiency, more flexibility, and improved reliability.

Future work could focus on extending and scaling DeepCoord to very large networks, e.g., in a distributed or hierarchical fashion. Here, it seems promising to further investigate complexity, reliability, and implementation options in large, multi-domain networks in combination with SDN.

References

[1] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service mesh: Challenges, state of the art, and future research opportunities," in *IEEE Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2019, pp. 122–1225.

[2] C.-H. Hong and B. Varghese, "Resource management in fog/edge computing: A survey on architectures, infrastructure, and algorithms," *ACM Comput. Surv.*, vol. 52, no. 5, Sep. 2019. [Online]. Available: https://doi.org/10.1145/3326066

[3] J. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," Internet Requests for Comments, RFC Editor, RFC 7665, 2015. [Online]. Available: http://www.rfc-editor.org/info/rfc7665

[4] ITU-T, "Architectural framework for machine learning in future networks including IMT-2020 (Y.3172)," ITU-T, Recommendation, 2019.

[5] J. G. Herrera and J. F. Botero, "Resource allocation in nfv: A comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[6] H. Moens and F. De Turck, "VNF-P: A model for efficient placement of virtualized network functions," in *International Conference on Network and Service Management (CNSM)*. IEEE, 2014, pp. 418–423.

[7] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and S. Davy, "Design and evaluation of algorithms for mapping and scheduling of virtual network functions," in *Conference on Network Softwarization (NetSoft)*. IEEE, 2015, pp. 1–9.

[8] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Optimal virtual network function placement in multi-cloud service function chaining architecture," *Computer Communications*, vol. 102, pp. 1–16, 2017.

[9] T.-W. Kuo, B.-H. Liou, K. C.-J. Lin, and M.-J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, 2018.

[10] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic virtual network function placement," in *IEEE Conference on Cloud Networking (CloudNet)*. IEEE, 2015.

[11] S. Dräxler, S. Schneider, and H. Karl, "Scaling and placing bidirectional services with stateful virtual and physical network functions," in *IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2018, pp. 123–131.

[12] Prometheus, "Documentation," https://prometheus.io/docs/prometheus/latest/configuration/configuration/ (March 18, 2020), 2020.

[13] G. Dulac-Arnold, D. Mankowitz, and T. Hester, "Challenges of real-world reinforcement learning," in *International Conference on Machine Learning (ICML) Workshop on RL4RealLife*, 2019.

[14] S. Schneider, A. Manzoor, H. Qarawlus, R. Schellenberg, and S. Uthe, "DeepCoord GitHub repository," https://github.com/RealVNF/DeepCoord (April 7, 2021), 2021.

[15] S. Schneider, A. Manzoor, H. Qarawlus, R. Schellenberg, H. Karl, R. Khalili, and A. Hecker, "Self-driving network and service coordination using deep reinforcement learning," in *International Conference on Network and Service Management (CNSM)*. IFIP/IEEE, 2020.

[16] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *International Conference on Learning Representations (ICLR)*, 2016.

[17] Z. Á. Mann, "Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms," *ACM Computing Surveys (CSUR)*, vol. 48, no. 1, pp. 1–34, 2015.

[18] C. Fuerst, S. Schmid, L. Suresh, and P. Costa, "Kraken: Online and elastic resource reservations for multi-tenant datacenters," in *IEEE Conference on Computer Communications (INFOCOMM)*. IEEE, 2016.

[19] L. Qu, C. Assi, and K. Shaban, "Delay-aware scheduling and resource optimization with network function virtualization," *IEEE Transactions on Communications*, vol. 64, no. 9, pp. 3746–3758, 2016.

[20] H. A. Alameddine, S. Sebbah, and C. Assi, "On the interplay between network function mapping and scheduling in VNF-based networks: A column generation approach," *IEEE Transactions on Network and Service Management*, vol. 14, no. 4, pp. 860–874, 2017.

[21] Q. Zhang, Y. Xiao, F. Liu, J. C. Lui, J. Guo, and T. Wang, "Joint optimization of chain placement and request scheduling for network function virtualization," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 731–741.

[22] L. Gu, D. Zeng, S. Tao, S. Guo, H. Jin, A. Y. Zomaya, and W. Zhuang, "Fairness-aware dynamic rate control and flow scheduling for network utility maximization in network service chain," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 5, pp. 1059–1071, 2019.

[23] S. Schneider, L. D. Klenner, and H. Karl, "Every node for itself: Fully distributed service coordination," in *International Conference on Network and Service Management (CNSM)*. IFIP/IEEE, 2020.

[24] M. Blöcher, R. Khalili, L. Wang, and P. Eugster, "Letting off STEAM: Distributed runtime traffic scheduling for service function chaining," in *IEEE Conference on Computer Communications (INFOCOMM)*. IEEE, 2020.

[25] Y. Chen and J. Wu, "Flow scheduling of service chain processing in a nfv-based network," *IEEE Transactions on Network Science and Engineering*, 2020.

[26] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica, "FairCloud: Sharing the network in cloud computing," in *ACM SIGCOMM Conference*, 2012, pp. 187–198.

[27] S. Dräxler, H. Karl, and Z. Á. Mann, "JASPER: Joint optimization of scaling, placement, and routing of virtual network services," *IEEE Transactions on Network and Service Management*, vol. 15, no. 3, pp. 946–960, 2018.

[28] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand *et al.*, "Autopilot: Workload autoscaling at google," in *European Conference on Computer Systems*, 2020, pp. 1–16.

[29] S. Schneider, N. P. Satheeschandran, M. Peuster, and H. Karl, "Machine learning for dynamic resource allocation in network function virtualization," in *IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2020, pp. 122–130.

[30] S. Rahman, T. Ahmed, M. Huynh, M. Tornatore, and B. Mukherjee, "Auto-scaling network service chains using machine learning and negotiation game," *IEEE Transactions on Network and Service Management*, 2020.

[31] C. Hardegen, B. Pfülb, S. Rieger, A. Gepperth, and S. Reißmann, "Flow-based throughput prediction using deep learning and real-world network traffic," in *IFIP/IEEE International Conference on Network and Service Management (CNSM)*. IFIP/IEEE, 2019, pp. 1–9.

[32] X. Fei, F. Liu, H. Xu, and H. Jin, "Adaptive vnf scaling and flow routing with proactive demand prediction," in *IEEE Conference on Computer Communications (INFOCOMM)*. IEEE, 2018, pp. 486–494.

[33] X. Zhang, C. Wu, Z. Li, and F. C. Lau, "Proactive VNF provisioning with multi-timescale cloud resources: Fusing online learning and online optimization," in *IEEE Conference on Computer Communications (INFOCOMM)*. IEEE, 2017, pp. 1–9.

[34] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3133–3174, 2019.

[35] J. Pei, P. Hong, M. Pan, J. Liu, and J. Zhou, "Optimal vnf placement via deep reinforcement learning in sdn/nfv-enabled networks," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 2, pp. 263–278, 2020.

[36] X. Wang, C. Wu, F. Le, and F. C. Lau, "Online learning-assisted VNF service chain scaling with network uncertainties," in *IEEE International Conference on Cloud Computing (CLOUD)*. IEEE, 2017, pp. 205–213.

[37] Y. Xiao, Q. Zhang, F. Liu, J. Wang, M. Zhao, Z. Zhang, and J. Zhang, "NFVdeep: adaptive online service function chain deployment with deep reinforcement learning," in *International Symposium on Quality of Service*, 2019, pp. 1–10.

[38] P. T. A. Quang, Y. Hadjadj-Aoul, and A. Outtagarts, "A deep reinforcement learning approach for vnf forwarding graph embedding," *IEEE Transactions on Network and Service Management*, vol. 16, no. 4, pp. 1318–1331, 2019.

[39] R. Solozabal, J. Ceberio, A. Sanchoyerto, L. Zabala, B. Blanco, and F. Liberal, "Virtual network function placement optimization with deep reinforcement learning," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 2, pp. 292–303, 2019.

[40] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *IEEE International Conference on Computer Communications (INFOCOMM)*. IEEE, 2018, pp. 1871–1879.

[41] Y. S. Nasir and D. Guo, "Multi-agent deep reinforcement learning for dynamic power allocation in wireless networks," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 10, pp. 2239–2250, 2019.

[42] L. Gu, D. Zeng, W. Li, S. Guo, A. Y. Zomaya, and H. Jin, "Intelligent VNF orchestration and flow scheduling via model-assisted deep reinforcement learning," *IEEE Journal on Selected Areas in Communications*, 2019.

[43] Cloud Native Computing Foundation, "Kubernetes: Production-grade container orchestration," https://kubernetes.io/ (Jan 31, 2020), 2020.

[44] H. Yu, J. Yang, and C. Fung, "Fine-grained cloud resource provisioning for virtual network function," *IEEE Transactions on Network and Service Management*, 2020.

[45] Amazon Web Services (AWS), "AWS docs (traffic dials)," https://docs.aws.amazon.com/global-accelerator/latest/dg/about-endpoint-groups-traffic-dial.html (March 18, 2020), 2020.

[46] L. Schiff, S. Schmid, and P. Kuznetsov, "In-band synchronization for distributed SDN control planes," *ACM SIGCOMM Computer Communication Review*, vol. 46, no. 1, pp. 37–43, 2016.

[47] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[48] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *AAAI Conference on Artificial Intelligence*, 2018.

[49] R. Hecht-Nielsen, "Theory of the backpropagation neural network," in *Neural networks for perception*. Elsevier, 1992, pp. 65–93.

[50] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[51] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[52] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv preprint arXiv:1606.01540*, 2016.

[53] S. Schneider, A. Manzoor, H. Qarawlus, R. Schellenberg, and S. Uthe, "Service coordination simulator GitHub repository," https://github.com/RealVNF/coord-sim (January 21, 2021), 2021.

[54] Q. Wei, D. Perez-Caparros, and A. Hecker, "Dynamic flow rules in software defined networks," in *European Workshop on Software-Defined Networks (EWSDN)*. IEEE, 2016, pp. 25–30.

[55] S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, D. Hock, M. Jarschel, and M. Hoffmann, "Heuristic approaches to the controller placement problem in large scale SDN networks," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, pp. 4–17, 2015.

[56] D. Zhou, Z. Yan, Y. Fu, and Z. Yao, "A survey on network data collection," *Journal of Network and Computer Applications*, vol. 116, pp. 9–23, 2018.

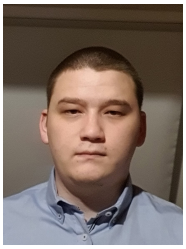
[57] S. Schneider, M. Jürgens, and H. Karl, "Divide and conquer: Hierarchical network and service coordination," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021.
[58] O. Tange *et al.*, "GNU parallel – the command-line power tool," *The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, 2011.
[59] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 9, pp. 1765–1775, 2011.
[60] M. Peuster, S. Schneider, and H. Karl, "The softwarised network data zoo," in *International Conference on Network and Service Management (CNSM)*. IEEE/IFIP, 2019, online Dataset: https://sndzoo.github.io/ (accessed January 21, 2021).
[61] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011, pp. 315–323.
[62] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference for Learning Representations*, 2015.
[63] W. Fischer and K. Meier-Hellstern, "The Markov-modulated poisson process (MMPP) cookbook," *Performance evaluation*, vol. 18, no. 2, pp. 149–171, 1993.
[64] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly, "SNDlib 1.0–Survivable Network Design Library," *Networks*, vol. 55, no. 3, pp. 276–286, 2010.
[65] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in NLP," in *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2019.
[66] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.
[67] S. Schneider, H. Qarawlus, and H. Karl, "Distributed online service coordination using deep reinforcement learning," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2021.
[68] A. Graves, M. G. Bellemare, J. Menick, R. Munos, and K. Kavukcuoglu, "Automated curriculum learning for neural networks," in *International Conference on Machine Learning (ICML)*, 2017.

Stefan Schneider obtained his master's degree in 2017 at the Paderborn University, Germany. He is currently pursuing his Ph.D., working as a research associate at the university's computer networks group. His main research interests are network softwarization, cloud computing, 5G and beyond—particularly in combination with machine learning. He has worked on multiple large research projects with industry partners and has been leading his own research project (RealVNF) in 2018–2021.

Ramin Khalili received his B.Sc. from Shiraz University, his M.Sc. from the Sharif University of Technology, both in Iran, and his Ph.D. in computer networks and distributed systems from UPMC, France. He was with the University of Massachusetts at Amherst, EPFL, and the Telekom Innovation Laboratories in Berlin, before joining the Huawei Research Center in Munich, Germany. Ramin published over thirty scientific papers and received multiple best paper awards during these years.

Adnan Manzoor received the B.Tech. degree in Computer Science from the University of Delhi, India, in 2017. He is currently pursuing the M.Sc. degree in Computer Science from the University of Paderborn, Germany. His main research and professional interests involve network virtualization, reinforcement learning, and natural language processing.

Haydar Qarawlus received his B.Sc. in Information Technology from the American University of Iraq, Sulaimani in 2016. He received his M.Sc. in Computer Science in 2020 from Paderborn University in Germany. His main research interests are network softwarization, software engineering, machine learning, and cloud computing.



Rafael Schellenberg is a student research assistant at the Computer Networks research group at Paderborn University, Germany. He is currently pursuing his bachelor's degree in computer engineering.

Holger Karl heads the Computer Networks research group in Paderborn University. He has two main research interests; the first one is advanced wireless network, e.g., cooperative diversity techniques and resource management in factory-floor automation. His second interest is future Internet, specifically the design and architecture of protocol stacks and unifying concepts like SDN and NFV across different scenario types.

Artur Hecker (MSc Universität Karlsruhe and PhD ENST, Paris) is Director of networking research at the Advanced Wireless Technology Laboratory of Huawei Munich Research Center. From 2006 to 2013, Artur was Associate Professor at Télécom ParisTech, where he was leader of Security and Networking research. Overall, Artur looks back at almost 20 years of entrepreneurial, academic and industry experience in networks, systems and system security.