Distributed Online Service Coordination Using Deep Reinforcement Learning

Stefan Schneider
Paderborn University, Germany
stefan.schneider@upb.de

Haydar Qarawlus
Paderborn University, Germany
qarawlus@mail.upb.de

Holger Karl
Paderborn University, Germany
holger.karl@upb.de

***Abstract*—Services often consist of multiple chained components such as microservices in a service mesh, or machine learning functions in a pipeline. Providing these services requires online coordination including scaling the service, placing instance of all components in the network, scheduling traffic to these instances, and routing traffic through the network. Optimized service coordination is still a hard problem due to many influencing factors such as rapidly arriving user demands and limited node and link capacity. Existing approaches to solve the problem are often built on rigid models and assumptions, tailored to specific scenarios. If the scenario changes and the assumptions no longer hold, they easily break and require manual adjustments by experts. Novel self-learning approaches using deep reinforcement learning (DRL) are promising but still have limitations as they only address simplified versions of the problem and are typically centralized and thus do not scale to practical large-scale networks.**

To address these issues, we propose a distributed self-learning service coordination approach using DRL. After centralized training, we deploy a distributed DRL agent at each node in the network, making fast coordination decisions locally in parallel with the other nodes. Each agent only observes its direct neighbors and does not need global knowledge. Hence, our approach scales independently from the size of the network. In our extensive evaluation using real-world network topologies and traffic traces, we show that our proposed approach outperforms a state-of-the-art conventional heuristic as well as a centralized DRL approach ($60\,\%$ higher throughput on average) while requiring less time per online decision ($1\,\mathrm{ms}$).

I. Introduction

Services consisting of multiple chained components, each with its own functionality, occur in various practical contexts. Examples are network services comprising chained virtual network functions (VNFs) in network function virtualization (NFV) [1], [2], microservices forming a service mesh in cloud and edge computing [3], [4], or machine learning functions in a pipeline [5]. In all of these cases, an ongoing challenge is to provide these services to users by deploying instances of the service components and routing incoming traffic through them, taking limited compute and link capacities into account. In particular, *service coordination* requires deciding how often to instantiate each component (scaling), where to place these instances in the network (placement), which incoming flows[1] to schedule to which placed instance (scheduling), and how to route these flows through the network (routing). As flows arrive rapidly and demand fluctuates over time, service coordination needs to adjust at runtime.

While this problem is well studied, existing work mostly focuses on long-/medium-term planning based on estimated user demand for an upcoming time interval. Even with recent advances in traffic forecasting [6]–[8], it is still likely that in practice, actual user demands differ from the predicted demand. Hence, any initial plan may no longer work when confronted with the actual user flows arriving at runtime, leading to either under- or over-allocation.

Furthermore, existing work typically proposes conventional approaches like solving mixed-integer linear programs (MILPs) or using heuristic algorithms, which are tailored to specific scenarios by experts and built on rigid models and assumptions. Operational reality easily diverges from such constructed scenarios, e.g., flows typically arrive stochastically without following any rigid pattern or with unexpected distributions. Then, assumptions may no longer hold and the proposed approaches could perform much worse than anticipated or even break completely. Consequently, experts are required again to understand the problem and adjust the approach manually in a time-consuming and error-prone way.

Hence, self-learning approaches have been proposed recently for service coordination. These approaches mainly use (deep) reinforcement learning (DRL) or contextual bandits to learn service coordination directly from experience rather than following hand-written rules. The hope is that, in this way, they can self-adapt to new scenarios or optimization objectives without human intervention or expertise. Existing DRL approaches, however, consider either complementary or simplified subproblems of the full scaling, placement, scheduling, and routing problem. Furthermore, they typically use centralized DRL approaches, where a single global DRL agent observes and controls the entire network. Up-to-date global knowledge of fast-changing information such as demand or resource utilization as well as central control are unrealistic in practical large-scale networks.

To address these issues, we propose a fully distributed self-learning approach that solves the full scaling, placement, scheduling, and routing problem. We deploy separate DRL agents at each node in the network making coordination decisions individually in parallel. These agents are trained offline in a centralized fashion, leveraging experience from all agents, and then coordinate services independently online

[1]We use the terms *flow* and *request* interchangeably here.

in a fast, distributed fashion. Each agent only observes itself and its direct neighbors and only has local control of how incoming flows are processed and forwarded. Hence, our approach requires neither global knowledge nor centralized control. In contrast to typical centralized approaches, the size of observation and action spaces is invariant to the network size and only depends on the network degree, i.e., maximum number of neighbors per node. Since the network degree is typically much smaller than the total number of nodes in the network [9], our approach scales better to realistic, large-scale networks. For example, it neither requires nor explicitly distinguishes different pre-defined tiers like fog/edge/cloud but works in all kinds of networks. Furthermore, our approach generalizes to new, unseen scenarios and is also more robust than typical centralized approaches as there is no single point of failure. In previous work, we presented a centralized DRL approach [10] and fully distributed heuristic algorithms [11]. The approach presented here combines the strengths of both approaches, namely flexibility and self-adaption of DRL and scalability and speed of distributed coordination. Our evaluation shows that it outperforms both approaches consistently and significantly. *Overall, our contributions are:*

- We formally define the service coordination problem of online scaling, placement, scheduling, and routing.
- To solve the problem, we formalize a Markov-decision process (MDP) with partial observability and propose a novel self-learning approach using distributed DRL agents for scalable service coordination in practice.
- We evaluate our approach on real-world network topologies and traffic traces, showing its adaptability, generalization capabilities, and scalability.
- For reproducibility, we publish our code online [12].

II. Related work

Existing work mostly proposes conventional optimization approaches without DRL for solving the service coordination problem, e.g., in cloud or edge computing [4], [13] or NFV [2]. Many authors [14]–[17] consider offline service coordination with full a priori knowledge of user demand. Related work that does consider online coordination [18]–[20] still does not schedule incoming flows at runtime but assumes traffic to arrive in fixed intervals and to be known a priori for each interval. Blöcher et al. [21] do schedule flows dynamically at runtime but assume a given fixed placement and do not consider routing. In contrast, we propose joint scaling, placement, scheduling, and routing at runtime in a distributed fashion without any a priori knowledge.

In previous work [11], we considered scaling, placement, scheduling, and routing at runtime proposing two fully distributed algorithms. Similar to our fully distributed DRL approach proposed here, these algorithms only have local observations and control, making individual coordination decisions at each node in parallel. In contrast to this previous work, we now consider flows to have deadlines that bound their maximum acceptable delay to achieve good Quality of Service (QoS). More importantly, the algorithms of our previous work, as well as all other aforementioned approaches, are hand-written and include built-in assumptions, which may not hold in practice. In contrast, our proposed model-free DRL approach does not follow a hand-written algorithm and has very limited built-in assumptions (e.g., about incoming traffic). Instead, it learns itself from experience how to deal with different traffic patterns effectively. We evaluate and compare our DRL approach against such a heuristic in Sec. V.

Recently, DRL for self-learning coordination has been proposed [10], [22]–[28]. Zhang et al. [22] focus only on placement and use tabular Q-learning, which does not support large and continuous observations or generalization between observations, limiting practical applicability. Pei et al. [23] rely on a simulator to test actions and select the best one in each time step, which would be too slow for online coordination at runtime with rapidly arriving flows—especially when done in a centralized fashion. Wang et al. [24] schedule flows equally to all placed instances, which could lead to high end-to-end delays and bad service quality. More similar to our approach, Quang et al. [25] and Xiao et al. [26] dynamically control individual flows and adjust component placement accordingly. However, the authors do not consider scaling or routing. Furthermore, they propose centralized approaches, assuming global up-to-date knowledge and control of the entire network. In practice, global knowledge can be achieved through monitoring but only with some monitoring delay (e.g., 1 min in Prometheus by default [29]), making it unsuitable for fast per-flow decisions at runtime. Such expensive centralized decisions per flow do not scale to large networks and would be too slow for efficient online scheduling of many rapidly arriving, time-sensitive flows as considered in our scenario.

Other authors [10], [27], [28] propose centralized DRL approaches but avoid scalability issues by installing forwarding rules at all nodes, which are then applied to incoming flows in a distributed fashion at runtime. These rules are then updated periodically by the centralized DRL agent. Compared to our proposed distributed DRL approach, these approaches still have a number of drawbacks: Xu et al. [27] only consider traffic engineering but not scaling and placement of service components. Gu et al. [28] and Schneider et al. [10] consider neither dynamic routing nor link capacities. Xu et al. and Gu et al. further require support from a hand-written heuristic. Moreover, all three approaches optimize coarse-grained rules that are applied to all incoming flows and do not have fine-grained control over individual flows. As we show in our evaluation, such fine-grained control is necessary for precise load balancing and proper dynamic routing. Our proposed distributed DRL approach jointly optimizes scaling, placement, scheduling, and routing. It does not require support from a heuristic and it can control individual flows efficiently by distributing decisions over different DRL agents for each node.

To the best of our knowledge, we propose the first distributed self-learning approach for the online service coordination problem. Compared to existing work, our approach is more powerful as it includes dynamic scaling, placement, scheduling, and routing at runtime and it is more efficient due

to its fully distributed architecture.

III. Problem Statement

We consider online service coordination, including scaling, placement, scheduling, and routing, in a network of geographically and topologically distributed nodes. In this section, we formalize the problem's parameters, decision variables, and objective. It is our intention here to precisely define the problem, not to provide a full mathematical formulation (e.g., as MILP) for feeding into a solver. Note that our proposed DRL approach is only explicitly aware of a small, locally observable subset of the problem parameters (defined in Sec. IV-B1) but can adapt to the scenario at hand through feedback from its actions.

A. Problem Parameters

We denote the underlying substrate network as undirected graph $G = (V, L)$. We do not assume the network to be predivided into tiers like fog/edge/cloud but rather require the coordination scheme to work in all kinds of networks. Each node $v \in V$ has a generic compute capacity $\text{cap}_v \in \mathbb{R}_{\geq 0}$. Adding more resource types, e.g., to distinguish CPU, memory, GPU, etc., is straightforward by considering a vector of capacities per node. Each link $l = (v, v') \in L$ connects two nodes v, v' bidirectionally and has a link delay $d_l \in \mathbb{R}_{\geq 0}$ as well as a maximum data rate $\text{cap}_l \in \mathbb{R}_{\geq 0}$, which is shared in both directions. We denote $L_v \subseteq L$ as the set of outgoing links at a node v and $V_v \subseteq V$ as set of direct neighbors that can be reached via these links. Furthermore, $V^{\text{in}} \subseteq V$ is the set of ingress nodes and $V^{\text{eg}} \subseteq V$ the set of egress nodes.

At ingress nodes, traffic arrives in the form of many partially overlapping flows from users requesting services. Each flow $f = (s_f, c_f, v_f^{\text{in}}, v_f^{\text{eg}}, \lambda_f, t_f^{\text{in}}, \delta_f, \tau_f) \in F$ is defined by

- its requested service s_f and the currently requested component $c_f \in C_s \cup \{\varnothing\}$, indicating the flow's progress within the service (cf. network service header [30]),
- its ingress and egress node $v_f^{\text{in}} \in V^{\text{in}}$ and $v_f^{\text{eg}} \in V^{\text{eg}}$,
- its data rate λ_f, which traversed instances may change,
- its arrival time t_f^{in} and duration δ_f,
- its deadline τ_f (relative to t_f^{in}), which is the maximum acceptable delay for traversing all requested components and reaching the egress node (more detail later).

A requested service $s = (n_s, C_s) \in S$ consists of a chain of n_s components $C_s = \langle c_1, ..., c_{n_s} \rangle$, which flows need to traverse in order. Set C contains all components from all services. After traversing and processing at an instance of a requested component $c_f = c_i \in C_s$, a flow directly requests the next component $c_f = c_{i+1} \in C_s$. After traversing the last component in the service chain, $c_f = \varnothing$, indicating that the flow is fully processed and needs to be routed to its egress.

Components can be instantiated and placed across different nodes in the network zero, one, or multiple times, where all instances are identical and independent from each other. Processing a flow f at an instance of component c incurs a delay d_c. For processing f, each instance of c requires resources $r_c(\lambda_f)$ relative to the flow's data rate λ_f. We assume a component-specific function $r_c(\lambda)$ is known or can be learned (e.g., using benchmarking and supervised learning [31]).

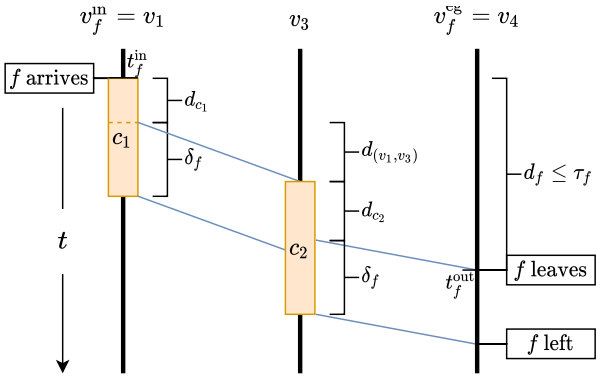


Fig. 1: Message sequence chart for flow $f = f_1$ in Fig. 2

Fig 1 illustrates the different delays a flow f experiences from entering the network at time t_f^{in} to leaving it at t_f^{out}, based on the example of $f = f_1$ in Fig. 2. Arriving at ingress v_1, f traverses an instance of c_1, incurring processing delay d_{c_1}. We assume processing to start directly for incoming flows rather than waiting for the entire flow to arrive completely before processing. For simplicity, we do not distinguish processing of individual packets but model flows as a continuous stream (cf. fluid approximation [32]). Our approach is neither explicitly aware of nor tied to these assumptions but adapts to a given environment through feedback from its actions. After traversing c_1, f heads to neighbor v_3 for processing c_2, adding the corresponding link delay $d_{(v_1,v_3)}$. With the flow duration δ_f in the example here, the first packets of the flow are already processed by c_2 at v_3 while later packets are still at v_1. Finally, end-to-end delay $d_f = t_f^{\text{out}} - t_f^{\text{in}}$ denotes the time from f's arrival to when it starts departing via the egress. Regarding QoS, d_f must be within the maximum acceptable delay τ_f. We denote $\tau_f^t = \tau_f - (t - t_f^{\text{in}})$ as remaining time from time t until f's deadline. Once $\tau_f^t = 0$, the flow expired and is dropped automatically, freeing any currently blocked resources.

B. Decision Variables

To solve the online service coordination problem, scaling, placement, scheduling, and routing need to be determined at runtime. We introduce binary decision variable $x_{c,v}(t) \in \{0, 1\}$ to indicate whether an instance of component c is placed at node v at time t (scaling and placement). An instance can be placed at multiple nodes, but we assume at most one instance per component and node. Internally, the node's operating system or a system like Kubernetes [33] may spawn more instances and handle the allocation to the node's internal resources, e.g., depending on whether it is a single, small machine or a large data center. We assume this intra-node coordination to be transparent and focus on inter-node coordination.

We introduce another decision variable $y_{f,c,v}(t) \in V_v \cup \{v\}$ to indicate how incoming flows are scheduled and routed. Particularly, $y_{f,c,v}(t) = v$ means that flow f requesting

component c at node v and time t is processed locally at v. This requires $x_{c,v}(t) = 1$, i.e., an instance of c needs to be available at v. Processing flow f increases the instance's resource consumption by $r_c(\lambda_f)$. The total resource consumption at a node must not exceed its capacity, i.e., $r_v(t) = \sum_{c \in C, f \in F} x_{c,v}(t) \mathbb{1}_{\{y_{f,c,v}(t)=v\}} r_c(\lambda_f) \leq \text{cap}_v$, where $\mathbb{1}_{\{y_{f,c,v}(t)=v\}}$ is an indicator variable that is 1 if $y_{f,c,v}(t) = v$ and 0 otherwise. When exceeding this capacity, flows cannot be properly processed at v and are dropped.

If $y_{f,c,v}(t) = v' \in V_v$, flow f is not processed locally at node v but forwarded to neighbor v' along link $l = (v, v') \in L_v$. To avoid dropping the flow, the link's capacity must not be exceeded, i.e., $r_l(t) = \sum_{f \in F} \mathbb{1}_{\{y_{f,c,v}(t)=v'\}} \lambda_f \leq \text{cap}_l$. When flow f arrives at neighbor v' at time t', again, the flow is either processed locally at v' or sent to one of its neighbors, depending on $y_{f,c,v'}(t')$.

C. Optimization Objective

Our goal is to set $x_{c,v}(t)$ and $y_{f,c,v}(t)$ such that as many flows as possible are processed successfully. A flow f is successful after routing from ingress v_v^{in} to egress v_v^{eg} while traversing instances of all components of the requested service s_f and completing within its deadline τ_f. Objective o_f is the percentage of successful flows and should be maximized:

$$\max o_f = \frac{|F_{\text{succ}}|}{|F_{\text{succ}}| + |F_{\text{drop}}|} \tag{1}$$

Maximizing this objective implicitly requires proper scaling and placement to ensure instances of requested components are available on a suitable path from ingress to egress. It also requires scheduling flows to instances at different nodes to balance load and avoid exceeding compute capacities. Similarly, link capacities need to be considered during routing. To ensure QoS and avoid timed-out and dropped flows, flows need to complete within their deadline. Consequently, flows with short deadline should be routed via shorter paths.

IV. Distributed DRL Approach

We propose a fully distributed deep reinforcement learning (DRL) approach to coordinate services online, maximizing objective o_f. We deploy separate DRL agents at each node in the network, where each agent only has local observations and control. In Sec. IV-A, we explain how these distributed agents jointly scale and place services as well as schedule and route incoming flows at runtime. Sec. IV-B formalizes the partially observable Markov decision process (POMDP), specifying the observation and action space as well the reward function for our DRL approach. Finally, the overall framework for training and inference is discussed in Sec. IV-C.

A. Joint Scaling, Placement, Scheduling, and Routing

At each node v, a separate DRL agent controls incoming flows independently from agents located at other nodes. Whenever a flow f arrives at a node v, v's DRL agent needs to decide $y_{f,c_f,v}(t)$, i.e., whether to process the flow locally at an instance of requested component c_f hosted at v or to forward f to a neighbor. By setting $y_{f,c_f,v}(t)$, the agents directly control flow scheduling and routing.

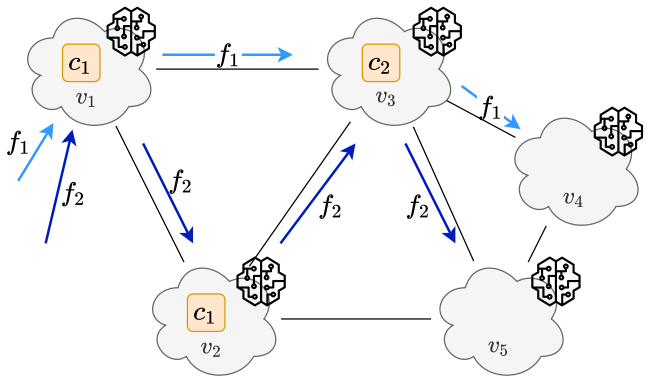


Fig. 2: Example with separate DRL agents at each node coordinating two incoming flows f_1, f_2.

We jointly derive the scaling and placement from $y_{f,c_f,v}(t)$, by setting $x_{c_f,v}(t) = 1$ if $y_{f,c_f,v}(t) = v$. Hence, if the agent decides to process f locally at node v, we ensure an instance of c_f is available at v or automatically start a new instance. Starting a new instance incurs additional component startup delay $d_{c_f}^{\text{up}}$ during which flow f has to wait. Unused instances of a component c_f are removed after a timeout δ_{c_f}, which can be configured based on how costly idle instances of c_f are.

To illustrate this process, consider the example in Fig. 2. Here, a flow f_1 and, shortly after, another flow f_2 arrive at ingress node v_1 and both request service s with $C_s = \langle c_1, c_2 \rangle$. Flow f_1 has egress $v_{f_1}^{\text{eg}} = v_4$ and f_2 has egress $v_{f_2}^{\text{eg}} = v_5$. When f_1 arrives at v_1 at time t_1, the node still has enough free resources to host an instance of $c_f = c_1$, such that v_1's DRL agent decides to process f_1 locally, setting $y_{f_1,c_1,v_1}(t_1) = v_1$ and $x_{c_1,v_1}(t_1) = 1$. When f_2 arrives shortly after at time t_2, v_1's resources are already fully utilized such that v_1's DRL agent decides to forward f_2 to its neighbor v_2, setting $y_{f_2,c_1,v_1}(t_2) = v_2$. At v_2, the corresponding DRL agent decides to process f_2 locally.

In the following, f_1 finishes processing c_1 at v_1, then requests the next component c_2, and is forwarded to neighbor v_3. At v_3, the corresponding DRL agent decides to instantiate c_2 and process f_1 locally. Similarly, f_2 is sent to process c_2 at v_3. After processing, the DRL agent at v_3 sends each flow to its egress node, where the flows depart successfully.

This example illustrates how the distributed DRL agents coordinate incoming flows individually and in parallel to the other nodes' agents. Depending on their own resource utilization as well as the utilization and location of their neighbors, they decide for each flow whether to process it locally or to forward it to a suitable neighbor, setting both decision variables $x_{c,v}(t)$ and $y_{f,c,v}(t)$ jointly at runtime.

B. MDP with Partial Observability

The complete network state depends on a multitude of parameters as indicated in Sec. III and cannot be fully observed by our DRL approach. Instead, each agent only has local and

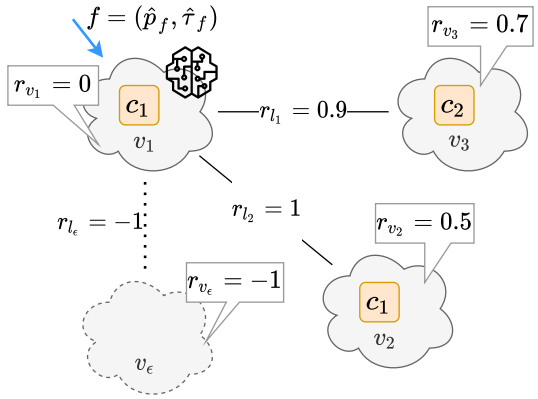


Fig. 3: Example observations of the DRL agent at node v_1 in Fig. 2. Node v_ϵ is a dummy node to ensure a consistent size of observations across all agents. Parametrization with time t omitted for simplicity.

partial observations that are realistically available in practice. To solve the problem using DRL, we formalize the corresponding partially observable Markov decision process (POMDP) as tuple $(\mathcal{O}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, consisting of observations $\mathcal{O}$, actions $\mathcal{A}$, typically unknown environment dynamics $\mathcal{P}$, and reward function $\mathcal{R}$. We define $\mathcal{O}, \mathcal{A}$, and $\mathcal{R}$ as follows.

1) Observations $\mathcal{O}$: An agent's observations are restricted to local information about the incoming flow f, current node v itself, and its neighbors. Specifically, $\mathcal{O} = \langle F_f, R_v^L, R_v^V, D_{v,f}, X_v \rangle$ consists of different parts for flow attributes F_f, link utilization R_v^L, node utilization R_v^V, delays to egress $D_{v,f}$, and available instances X_v. We normalize all observations to be in range $[-1, 1]$ (or $[0, 1]$) as detailed below. Ensuring that all observations are in a similar range is important for effective training and generalization of deep neural networks [34]. Otherwise, the DRL agent may become "blind" to the weak signals of observations with a small range, which are drowned out by stronger signals of other observations with much larger ranges.

a) Flow Attributes F_f: Vector $F_f = \langle \hat{p}_f, \hat{\tau}_f \rangle$ consists of two relevant flow attributes. The progress inside the service chain of a flow f is indicated by $\hat{p}_f \in [0, 1]$ (cf. service index in SFC [30]). It starts at $\hat{p}_f = 0$ when the flow arrives and progresses towards $\hat{p}_f = 1$ with every traversed component. With respect to deadlines, the agent observes $\hat{\tau}_f = \frac{\tau_f^t}{\tau_f} \in [0, 1]$, which is the remaining time to the flow's deadline normalized by deadline τ_f itself (defined relative to flow arrival t_f^{in}). Hence, it starts at $\hat{\tau}_f = 1$ and gradually decreases towards 0 over time.

b) Link Utilization R_v^L: Vector $R_v^L = \langle \frac{r_l(t) - \lambda_f}{\max_{l' \in L_v} \text{cap}_{l'}} | l \in L_v \rangle \in [-1, 1]^{\Delta_G}$ contains the free resources on all outgoing links L_v of node v, normalized by the maximum capacity of links in L_v. Subtracting the flow's data rate λ_f shifts the value to be ≥ 0 if and only if a link can forward the flow.

To allow combining experiences from all agents (details in Sec. IV-C), the size of observation and action space needs to be identical for all agents. Hence, we define $|R_v^L| = \Delta_G$ according to network degree Δ_G, i.e., the maximum number of neighbors in the network. If a node has less than Δ_G neighbors, we add dummy neighbors v_ϵ to V_v and set their link utilization to -1, indicating that they do not exist. Fig. 3 shows the observations of the DRL agent at v_1 in the example network of Fig. 2. Here, a dummy node v_ϵ is added to v_1's two neighbors to match the network degree $\Delta_G = 3$.

c) Node Utilization R_v^V: Similar to R_v^L, vector $R_v^V = \langle \frac{r_{v'}(t) - r_c(\lambda_f)}{\max_{v'' \in V} \text{cap}_{v''}} | v' \in V_v \cup \{v\} \rangle \in [-1, 1]^{\Delta_G + 1}$ contains the free compute resources at v and its neighbors if they were to process the flow at the requested component. Again, we normalize by the maximum node capacity such that the observation is in $[0, 1]$ if there are enough resources for processing the flow and in $[-1, 0)$ otherwise. Here, we divide by the maximum capacity over all nodes (not just neighbors). While a flow has to traverse one of the outgoing links, it can be processed on any node in the network. This normalization helps the DRL agent to identify nodes with high available absolute resources (not just relative to the neighborhood). In case flow f is already fully processed ($c_f = \varnothing$), we set $r_{c_f}(\lambda_f) = 0$. As before, we ensure $|R_v^V| = \Delta_G + 1$ (including v itself) by adding observations of $r_{v_\epsilon}(t) = -1$ for dummy nodes v_ϵ if necessary.

d) Delays to Egress $D_{v,f}$: Vector $D_{v,f} = \langle \max \left\{ -1, \frac{\tau_f^t - d_{v,v',v_f^{\text{eg}}}}{\tau_f^t} \right\} | v' \in V_v \rangle \in [-1, 1]^{\Delta_G}$ defines the shortest path delays from current node v to flow f's egress via each neighbor v' in relationship to the remaining time τ_f^t to f's deadline. This information helps the agent forward f to neighbors that are in the direction of its egress node. If the observation is below 0 for a neighbor v', there is no chance that forwarding via v' will be successful. Assuming a fixed network topology and link delays, the shortest paths and their path delays $d_{v,v',v_f^{\text{eg}}}$ can be precomputed and accessed in constant time during runtime. Again, we pad $D_{v,f}$ with -1 to ensure length $|D_{v,f}| = \Delta_G$.

e) Available Instances X_v: Binary vector $X_v = \langle x_{c_f,v'}(t) | v' \in V_v \cap \{v\} \rangle \in \{0, 1\}^{\Delta_G + 1}$ indicates whether an instance of requested component c_f is currently available at v and its neighbors. After traversing the last component in the service, $x_{c_f,v'}(t)$ is always zero. Again, we pad X_v with -1 to ensure length $|X_v| = \Delta_G + 1$.

2) Actions $\mathcal{A}$: The DRL agents take actions whenever a flow f arrives at their node. The action space $\{0, 1, ..., \Delta_G\}$ is the same for all agents depending on the network degree Δ_G. An action $a \in \{0, 1, ..., \Delta_G\}$ by a DRL agent at node v specifies how to set $y_{f,c_f,v}(t)$. If $a = 0$, the flow is processed locally, i.e., $y_{f,c_f,v}(t) = v$. An action $a > 0$ sends the flow to v's a-th neighbor $v_a \in V_v$. An action is only valid for $a \leq |V_v|$. If a node has fewer neighbors than Δ_G, an action $|V_v| < a \leq \Delta_G$ points to a non-existing neighbor and is invalid. In this case, flow f is dropped and the agent receives a high penalty. Based on the observed -1 values for non-existing dummy neighbors, agents should be aware of which neighbors really exist and only forward flows there.

If a DRL agent selects $a = 0$ even though flow f is already fully processed ($c_f = \varnothing$), f stays at the node for one time step and the agent is queried again. Doing so reduces the remaining time τ_f^t to f's deadline and incurs a small penalty. Based on this penalty and on observation $\hat{p}_f$ (Sec. IV-B1a), agents should learn to forward processed flows directly to their egress without keeping them unnecessarily.

3) Reward $\mathcal{R}$: Ultimately, we want the DRL agents to learn a coordination scheme that processes as many flows successfully as possible. Hence, we give a large positive reward of $+10$ when a flow completes. Conversely, we penalize the agent with a reward of -10 when dropping a flow.

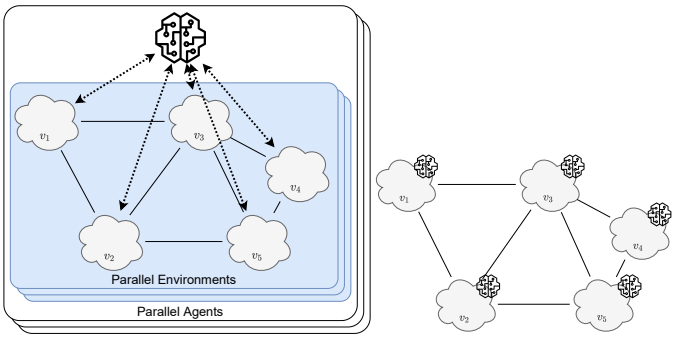
Successful completion of a flow requires proper coordination, taking available neighbors, compute and link resources, link delays, and flow deadlines into account. Hence, when starting training with a random policy, it is very unlikely that a series of random actions leads to a flow completing successfully. Consequently, rewards of $+10$ will initially be very sparse, preventing effective training.

One way to address the challenge of sparse rewards is through reward shaping [35]. To this end, we add additional weaker reward signals that indicate whether an action seemed useful or not—even before a flow is completed or dropped. Specifically, we give a small positive reward of $+\frac{1}{n_{s_f}}$ whenever a flow successfully traverses an instance, where n_{s_f} is the length of the requested service chain. This encourages the DRL agents to host component instances and process flows locally when possible. Furthermore, we give a small penalty of $-\frac{d_l}{D_G}$ whenever a DRL agent sends a flow along a link l. This penalty corresponds to the link's propagation delay normalized by the network's diameter D_G (in terms of path delay). It encourages the DRL agents to forward flows along fewer links and links with shorter delays when possible. We give a similar penalty of $-\frac{1}{D_G}$ when a DRL agent keeps a flow ($y_{f,c_f,v}(t) = v$) that is already fully processed.

These additional rewards help nudge the DRL agents towards a useful policy and can improve training. Yet, it is important these auxiliary rewards are significantly smaller than the rewards/penalties for successful and dropped flows. Otherwise, they limit the agents' ability to find "creative" but successful policies. Too strong auxiliary rewards can even encourage unwanted behavior, e.g., if processing two flows half-way is more rewarding than fully completing one flow.

C. DRL Service Coordination Framework

As defined in Sec. IV-B, we approach service coordination through local observations and actions that control incoming flows for each node. We propose a DRL framework in which we train offline, combining experience from all nodes in a logically centralized deep neural network as illustrated in Fig. 4a. After training converges, this single, central neural network can make decisions for any node in the network. It distinguishes different nodes based on different observations and takes suitable actions accordingly. We can now deploy separate DRL agents at each network node and copy the neural network to each of these agents (Fig. 4b). This allows each agent to locally control incoming flows highly efficiently based only on local observations. While each DRL agent has a copy of the same neural network and thus follows the same policy, this policy is trained to distinguish flows at different nodes and handle them correspondingly.



(a) Centralized training (b) Distributed inference

Fig. 4: a) Centralized training procedure with multiple copies of the network environment and the DRL agent. b) Fully distributed inference with trained DRL agents at each node.

1) Design Choices: Two natural alternatives to our approach are either centralized observations and control for all network nodes at once or distributed training and inference using different neural networks for each node. A centralized approach observing and controlling all nodes at once is more in line with current related work (Sec. II). Yet this would require a significantly larger observation and action space, e.g., a concatenation of our observations and actions for each node, which typically requires more training until convergence. Furthermore, centralized inference would not scale to large networks with rapidly incoming flows, where thousands of decisions may be necessary per millisecond for the network as a whole. Besides, it would require up-to-date global knowledge, which is not realistically available, but at best partial and delayed through monitoring. Distributed training and inference using separate neural networks is a promising alternative approach, yet makes training challenging. E.g., when training distributed neural networks, agents at nodes that are seldom traversed by flows would barely be trained at all, possibly leading to bad policies for these nodes. Hence, our proposed centralized training with distributed inference tries to combine the benefits of both approaches, e.g., by leveraging experience from all agents, giving us more data for effective training. To support continuous online training during inference, DRL agents could update their neural network locally and then synchronize the gradient updates with all other nodes (cf. federated learning [36], [37]). Lest online inference is blocked, training and sharing of updates should happen asynchronously.

However, our approach also comes with the challenge of properly designing the POMDP such that the trained neural network can effectively generalize between similar situations but still distinguish semantically different situations. For example, to keep the action space small, action i means selecting neighbor i, not necessarily node i. Hence, the same action i

Algorithm 1 DRL Training and Inference

1: initialize π_θ, V_ϕ, b ▷ Training
2: $l \leftarrow$ num. parallel training environments
3: **for** l environments in parallel **do**
4: **while** $t \leq T$ **do**
5: **if** flow f arrives at node v **then**
6: $o_t, r_t \leftarrow$ adapter.process(f, v, V_v, L_v, G)
7: $b \xleftarrow{\text{add}} (o_{t-1}, a_{t-1}, r_t, o_t)$
8: $a_t \leftarrow \pi_\theta(o_t)$
9: $x_{c_f,v}(t), y_{f,c_f,v}(t) \leftarrow$ adapter.process(a_t)
10: **if** b is full **then**
11: train V_ϕ using temporal difference updates [39]
12: train π_θ maximizing $\mathbb{E}[\sum_i \gamma^i r(o_{t+i}, a_{t+i})]$
13: Select best agent (π_θ, V_ϕ) ▷ Inference
14: Deploy a copy π_θ^v of π_θ at each node $v \in V$
15: **while** $t \leq T$ **do**
16: **if** flow f arrives at node v **then**
17: $o_t, r_t \leftarrow$ adapter.process(f, v, V_v, L_v, G)
18: $a_t \leftarrow \pi_\theta^v(o_t)$
19: $x_{c_f,v}(t), y_{f,c_f,v}(t) \leftarrow$ adapter.process(a_t)

can lead to very different results when executed by DRL agents at different nodes. To help the DRL agents distinguish between different neighbors and selecting a suitable one, our observations include all relevant information about neighbors' resources, available instances, and distance to the egress node (in terms of shortest path delay). This should help the DRL agents learn to select neighbors with sufficient resources and towards the egress node—independent from the exact node IDs and thus generalizing across DRL agents at different nodes.

2) Algorithm: We selected the actor-critic using Kronecker-factored trust region (ACKTR) algorithm [38] to train our DRL agents. ACKTR is an extension to the well-known asynchronous advantage actor-critic (A3C) [39], which is an on-policy, actor-critic approach leveraging multiple parallel environment copies during training for more diverse training data. Similar to trust region policy optimization (TRPO) [40] and proximal policy optimization (PPO) [41], ACKTR ensures that the learned policy is updated gradually during training, avoiding abrupt and potentially destructive changes in the learned behavior.

Alg. 1 shows the high-level algorithm for centralized offline training (ln. 1–12) and then for distributed online inference (ln. 13–19). ACKTR trains two deep neural networks for the actor π_θ and the critic V_ϕ using mini-batches b. The two neural networks are initialized randomly and trained over l parallel copies of the network environment (ln. 1–3). Clearly, $l = 1$ if the network environment cannot be duplicated. Whenever a flow arrives at a node (ln. 5–9), the DRL agent obtains current observation o_t and previous reward r_t from the network through an adapter. It adds the experience to mini-batch b and selects the next action by passing o_t to its actor π_θ. Based on selected action a_t, the corresponding coordination decisions are taken as described in Sec. IV-A and IV-B2.

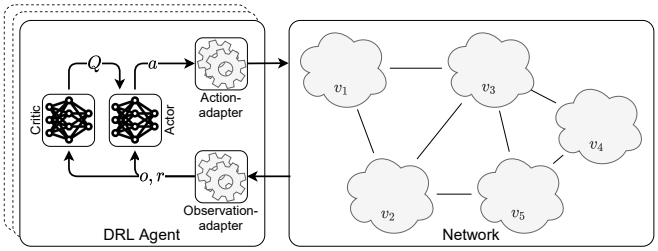


Fig. 5: Implementation consisting of the DRL agent interacting with the network through action- and observation-adapters.

Once mini-batch b is full (i.e., contains a predefined number of experiences), critic and actor are trained (ln. 10–12). Critic V_ϕ estimates the long-term value $V_\phi(o)$ of observing o and following the current policy. It is trained using bootstrapping and temporal difference as detailed in [38], which is a standard technique for reinforcement learning [42]. The value estimate $V_\phi(o)$ is necessary to calculate the advantage, i.e., the relative value of each action a after observing o, which is, in turn, needed to train the actor π_θ. The actor is trained to maximize the long-term return, i.e., the discounted (by γ) sum of future rewards, using the natural gradient method [38]. This training procedure is repeated for a configurable number of training episodes until the agent converges.

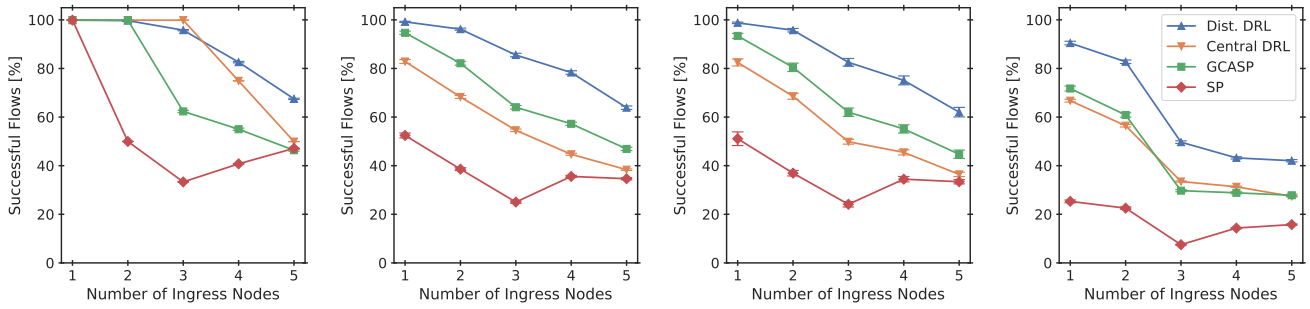
As the random seed used for training can have a significant impact on convergence [43], we train multiple agents (in parallel) using k different random seeds. After training, we automatically select the best agent with the highest reward for online inference (ln. 13). The neural network of this agent is copied to each network node to facilitate fast, distributed inference by a local DRL agent at each node (ln. 14–19). While offline training can be time-intensive, depending on the number of training episodes, online inference is very fast [44]. With a fixed number of hidden units, time and space complexity for inference is in $O(\Delta_G)$, i.e., linear in the network degree.

3) Implementation: We implemented a prototype of our DRL approach as illustrated in Fig. 5 and published it on GitHub [12]. We built the DRL agent on Tensorflow [45] and the `stable-baselines` framework [46] using Python. As light-weight network simulator, we used `coord-sim` [47]. To enable the interaction between the DRL agent and the network, we implemented the OpenAI Gym interface through adapters. These adapters interface the network environment to retrieve relevant observations (e.g., from monitoring data), calculate the reward during training, and apply the selected actions.

V. Numerical Evaluation

A. Evaluation Setup

We evaluate our proposed fully distributed DRL approach and compare its performance against state-of-the-art approaches in a variety of different scenarios. In Sec. V-B, we evaluate how well our DRL approach adapts to varying



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

Fig. 6: Our distributed DRL approach processes most flows successfully with increasing load at different traffic patterns.

ingress nodes and traffic patterns, comparing the percentage of successful flows (as defined in Sec. III-C) over $T = 20000$ time steps. Similarly, we evaluate its adaptability to varying flow deadlines in Sec. V-C, just by retraining the DRL agent for each scenario but without changing any hyperparameters or making manual adjustments. In Sec. V-D, we investigate how well our DRL approach generalizes to previously unseen scenarios without any retraining. Finally, Sec. V-E evaluates the scalability of our approach on large real-world network topologies.

1) Base Scenario: We consider different variations of a base scenario using real-world network topology Abilene [9], which connects 11 cities in the US. We assign compute resources to nodes randomly uniformly between 0 and 2, link capacities between 1 and 5, and derive link delay from the distance between connected nodes. We consider a video streaming service s with $C_s = \langle c_{\text{FW}}, c_{\text{IDS}}, c_{\text{video}} \rangle$. While we successfully tested our approach with multiple services, we focus on a single service in our evaluation for simplicity. All components $c \in C_s$ have a processing delay of $d_c = 5\,\text{ms}$ and require resources linear to their load. Flows requesting s have unit data rate ($\lambda_f = 1$) and length ($\delta_f = 1$) and deadline $\tau_f = 100$, but we consider scenarios with increasingly complex flow arrival patterns (Sec. V-B) and varying deadlines (Sec. V-C). We consider a single egress v_8 but vary between 1 and 5 ingress nodes (v_1–v_5).

2) DRL Hyperparameters: We use the following hyperparameter settings for our DRL approach: 1) Deep neural networks with 2x 256 hidden units (and `tanh`-activation [49]) for both actor and critic, trained with the RMSprop optimizer [50]. 2) Discount factor $\gamma = 0.99$. 3) Initial learning rate $\alpha = 0.25$. 4) $k = 10$ training seeds and $l = 4$ parallel training environments (cf. Sec. IV-C). 5) ACKTR-specific parameters: Entropy loss 0.01, loss on V_ϕ 0.25, and Fisher coefficient 1.0, max. gradient 0.5, Kullback-Leibler clipping 0.001. We selected these hyperparameter settings based on the ACKTR default settings [46] and manual tuning to achieve good performance across scenarios. Tuning hyperparameters automatically for each scenario is time and resource-intensive but could potentially further improve performance.

3) Compared Algorithms: We compare our proposed fully distributed DRL approach against the following baseline and state-of-the-art approaches from our previous work, where the code is publicly available:

- A centralized DRL approach [10], which uses and periodically updates forwarding rules at all nodes that are applied to incoming flows at runtime. It handles partial and delayed observations of the global network state, which are available via periodic monitoring.
- A fully distributed heuristic algorithm, GCASP [11]. Similar to our proposed approach, each node observes and controls incoming flows locally.
- A simple greedy baseline, "SP", which tries to process all flows along the shortest path from ingress to egress.

4) Experiment execution: We ran experiments on machines with Intel Xeon W-2145 CPUs and 128 GB RAM. Figures show the mean and standard deviation over 30 random seeds.

B. Varying Traffic Patterns

We evaluate our fully distributed DRL approach on the base scenario (Sec. V-A1) with varying flow arrival patterns and an increasing number of ingress nodes (v_1–v_5) and thus increasing load. The simplest pattern we consider is *fixed flow arrival* with flows arriving every 10 time steps at each ingress node. Fig. 6a shows the corresponding percentage of successful flows (legend in Fig. 6d), which naturally decreases with increasing load and limited node and link capacities. While all approaches process almost all flows successfully with one ingress node, only the self-learning DRL approaches achieve around 100 % successful flows for up to three ingress nodes. For four and five ingress nodes, our proposed distributed DRL approach outperforms all other algorithms and successfully processes up to 35 % more flows than the central DRL approach. While the central DRL approach relies on shortest path routing, our distributed DRL approach explicitly optimizes routing jointly with scaling, placement, and scheduling. In doing so, it takes link capacities into account and balances load across different paths to minimize dropped flows. The simple SP baseline relies on sufficient resources along the shortest path and thus easily drops flows. Ingress nodes v_1–v_3

are co-located in the given network such that their shortest paths to the egress overlap and compete for shared resources. Ingress v_4 and v_5 are farther away such that their shortest paths do not overlap and, instead, allow SP to utilize more resources for more successful flows. Similar to SP, the fully distributed heuristic, GCASP, favors processing flows along the shortest paths but dynamically reroutes flows when necessary, avoiding bottlenecks and searching for compute resources.

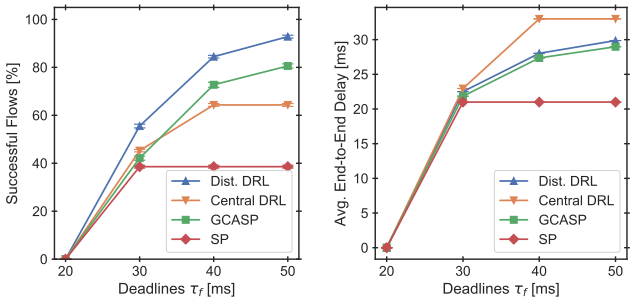
Next, we consider stochastic flow arrival following a *Poisson process* with exponentially distributed inter-arrival times (mean 10 time steps). Fig. 6b shows the corresponding results. Here, the central DRL approach performs worse because its centralized observations are always slightly outdated – as they would be for any centralized approach in practice! Hence, it cannot effectively react to small bursts where multiple flows arrive directly after another. Instead, the same forwarding rules are applied to all incoming flows, which easily leads to dropped flows when considering stochastic flow arrival. In contrast, the fully distributed architecture based on local observations and control of both our proposed distributed DRL approach and GCASP lets them control individual flows quickly at runtime. This allows them to react to short bursts, distributing individual flows among available resources. Still, our distributed DRL clearly and consistently outperforms all other algorithms and is 37 % better than GCASP on average.

Fig. 6c shows very similar results for yet more realistic flow arrival following a *Markov-modulated Poisson process (MMPP)* [51]. The corresponding 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. Our distributed DRL approach successfully adapts to this traffic pattern and outperforms all other approaches (GCASP by 39 % on average).

Finally, Fig. 6d shows the results for flows following real-world traffic traces that are publicly available for the Abilene network [52]. Here, the central DRL and GCASP lead to comparable results. Again, our distributed DRL is considerably better and even increases its lead over both approaches with 60 % higher success rates on average.

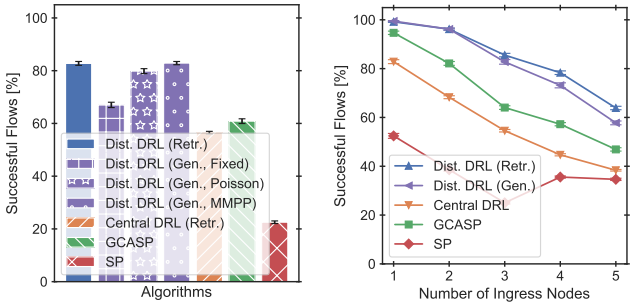
C. Varying Deadlines

We now consider the base scenario (Sec. V-A1) with fixed ingress nodes (v_1, v_2), Poisson flow arrival and instead systematically vary flows' deadlines ($\tau_f \in \{20, 30, 40, 50\}$). Fig. 7 shows the resulting percentage of successful flows and corresponding avg. end-to-end delay of completed flows. With deadline $\tau_f = 20$, all flows are dropped since flows need more than 20 ms to complete, even in the best case where flows are processed along the shortest paths. Indeed, the SP heuristic attempts to process all flows along the shortest paths, leading to a fixed avg. end-to-end delay of 21 ms for deadlines $\tau_f = 30$ and above. Further increasing the deadlines does not increase the percentage of successful flows for SP. In contrast, the other algorithms exploit increasing deadlines and then also use longer paths to balance load. Overall, our proposed distributed DRL approach balances the load more effectively



(a) Successful Flows (b) Avg. End-To-End Delay

Fig. 7: Our distributed DRL adapts to varying flow deadlines.



(a) Unseen Traffic Patterns (b) Unseen Network Load

Fig. 8: Our distributed DRL generalizes to unseen scenarios.

taking the given deadlines into account, processing more flows successfully than any other algorithm (21 % more than GCASP on average).

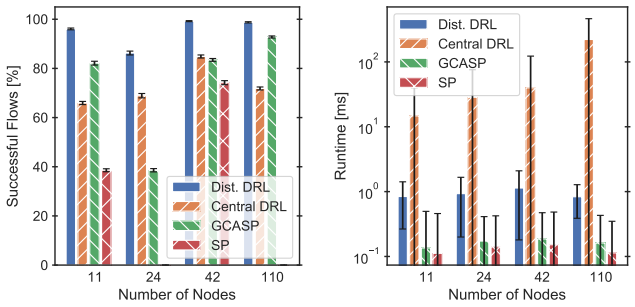
D. Generalization to Unseen Scenarios

In the previous sections, we evaluate how well our distributed DRL approach adapts to changing scenarios (traffic load, traffic patterns, deadlines) just by retraining but without human interaction. Here, we go one step further and investigate how well our trained DRL agent generalizes to previously unseen scenarios without any retraining. In practice, it is important that the DRL agent continues to coordinate services reasonably even if the scenario suddenly changes. A new DRL agent may be retrained periodically (or even continuously) to optimize for the current scenario but until training convergence, the incumbent DRL agent still has to coordinate decently. To facilitate generalization, we designed our observation space to include generally available and useful information and normalized all values to be in a similar range (Sec. IV-B1).

Here, we explore generalization to previously unseen traffic patterns with two ingress nodes in the base scenario (Sec. V-A1). To this end, we train our distributed DRL approach on fixed flow arrival and test it without retraining on unseen real-world traces. Similarly, we test generalization of

TABLE I: Real-world network topologies [9].

Network	Nodes	Edges	Degree (Min./Max./Avg.)
Abilene	11	14	2 / 3 / 2.55
BT Europe	24	37	1 / 13 / 3.08
China Telecom	42	66	1 / 20 / 3.14
Interroute	110	158	1 / 7 / 2.87



(a) Successful Flows (b) Inference Runtime

Fig. 9: Our distributed DRL scales to large networks.

the DRL agent trained on Poisson and MMPP traffic. Fig. 8a shows the percentage of successful flows when generalizing these different DRL agents to unseen trace-driven traffic. For comparison, the figure also shows the success ratio of the DRL approach trained and tested on these traces and of the other algorithms. The success rates of the generalizing DRL agents ("Gen." in Fig. 8a) are very close to the performance of the retrained agent ("Retr.") and still clearly outperform the other algorithms, indicating that our proposed distributed DRL generalizes well to unseen traffic patterns.

Next, we investigate generalization to unseen network loads by training our distributed DRL on the base scenario with two ingress nodes (and Poisson traffic) and testing it with 1–5 ingress nodes, representing increasing load. Fig. 8b shows the resulting successful flows in comparison with a DRL agent that is retrained for each scenario and with the other algorithms. Again, the generalizing DRL agent ("Gen.") processes almost as many flows successfully as the DRL agent that is retrained for each scenario ("Retr.") and still clearly outperforms all other algorithms.

E. Scalability

Finally, we evaluate the scalability of our distributed DRL on large real-world network topologies. Specifically, we consider the BT Europe, China Telecom, and Interroute networks in addition to Abilene [9] (Table I). Particularly the China Telecom network is highly skewed in terms of node degree, which influences the size of our observation and action space (Sec. IV-B). As before, we consider Poisson traffic at two ingress nodes (with node IDs v_1 and v_2), one egress (v_8) and randomly uniform node capacities (between 0 and 2) and link capacities (between 1 and 5).

Fig. 9a shows the resulting percentage of successful flows on these networks. Despite the large size and node degree skewness of these networks, our distributed DRL achieves almost perfect results with close to $100\,\%$ successful flows. In the BT Europe network, link capacities between ingress and egress are scarce leading to more dropped flows for all algorithms. SP fails completely on BT Europe (24 nodes) and Interroute (110 nodes) since there are insufficient resources on the shortest paths. Our distributed DRL outperforms the central DRL by $31\,\%$ and GCASP by $42\,\%$ on average. At the same time, Fig. 9b (log. scale) shows that the inference time of our distributed DRL is on the order of $1\,\mathrm{ms}$. In particular, it is magnitudes faster than the central DRL and, unlike the central DRL, invariant to the network size.

VI. Conclusion

We present a fully distributed self-learning and self-adaptive DRL approach for autonomous service coordination, i.e., joint scaling, placement, scheduling, and routing. Our evaluation shows that our proposed DRL approach adapts to varying scenarios with different traffic patterns, load, QoS requirements (deadlines), and network topologies—without requiring any expert knowledge or human intervention. It also generalizes to new, unseen scenarios and scales to large real-world network topologies while making decisions within roughly $1\,\mathrm{ms}$. Thanks to the fully distributed DRL architecture, our approach is more flexible, better scalable, and generally more successful than current state-of-the-art solutions. Overall, we believe that this makes it a much better approach for autonomous service coordination in practice.

Acknowledgments

This work was supported in part by the German Research Foundation within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901) and the German Federal Ministry of Education and Research through Software Campus grant 01IS17046 (RealVNF).

References

[1] 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

[2] 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.

[3] 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.

[4] 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

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

[6] E. I. Vlahogianni, J. C. Golias, and M. G. Karlaftis, "Short-term traffic forecasting: Overview of objectives and methods," *Transport reviews*, vol. 24, no. 5, pp. 533–557, 2004.

[7] P. Cortez, M. Rio, M. Rocha, and P. Sousa, "Internet traffic forecasting using neural networks," in *IEEE International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2006, pp. 2635–2642.

[8] B. Yu, H. Yin, and Z. Zhu, "Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting," in *International Joint Conference on Artificial Intelligence*, 2018, pp. 3634–3640.

[9] 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.

[10] 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.

[11] 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.

[12] H. Qarawlus, "Distributed drl GitHub repository," https://github.com/RealVNF/distributed-drl-coordination (December 23, 2020), 2020.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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, pp. 1562–1576, 2018.

[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] 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, pp. 255–260.

[20] 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.

[21] 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.

[22] Z. Zhang, L. Ma, K. K. Leung, L. Tassiulas, and J. Tucker, "Q-placement: Reinforcement-learning-based service placement in software-defined networks," in *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1527–1532.

[23] 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*, 2019.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] P. Quinn, U. Elzur, and C. Pignataro, "Network service header (NSH)," IETF, Internet Request for Comments RFC 8300, 2018.

[31] 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.

[32] H. Chen and A. Mandelbaum, "Discrete flow networks: Bottleneck analysis and fluid approximations," *Mathematics of operations research*, vol. 16, no. 2, pp. 408–446, 1991.

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

[34] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," ser. Proceedings of Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 448–456. [Online]. Available: http://proceedings.mlr.press/v37/ioffe15.html

[35] A. Trott, S. Zheng, C. Xiong, and R. Socher, "Keeping your distance: Solving sparse reward tasks using self-balancing shaped rewards," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019, pp. 10 376–10 386.

[36] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[37] W. Luping, W. Wei, and L. Bo, "CMFL: Mitigating communication overhead for federated learning," in *International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 954–964.

[38] Y. Wu, E. Mansimov, R. B. Grosse, S. Liao, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," in *Advances in neural information processing systems (NeurIPS)*, 2017, pp. 5279–5288.

[39] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2016, pp. 1928–1937.

[40] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning (ICML)*, 2015, pp. 1889–1897.

[41] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[42] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[43] 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.

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

[45] 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.

[46] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines," https://github.com/hill-a/stable-baselines, 2018.

[47] S. Schneider, A. Manzoor, H. Qarawlus, R. Schellenberg, and S. Uthe, "Service coordination simulator GitHub repository," https://github.com/RealVNF/coord-sim (November 18, 2020), 2020.

[48] O. Tange *et al.*, "GNU parallel – the command-line power tool," *The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, 2011.

[49] B. L. Kalman and S. C. Kwasny, "Why tanh: Choosing a sigmoidal function," in *International Joint Conference on Neural Networks (IJCNN*, vol. 4. IEEE, 1992, pp. 578–581.

[50] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[51] W. Fischer and K. Meier-Hellstern, "The Markov-modulated poisson process (MMPP) cookbook," *Performance evaluation*, vol. 18, no. 2, pp. 149–171, 1993.

[52] 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.