Self-Driving Network and Service Coordination Using Deep Reinforcement Learning

Stefan Schneider*, Adnan Manzoor, Haydar Qarawlus,
Rafael Schellenberg, Holger Karl
Paderborn University
{stschn, adnan904, qarawlus, srafael, hkarl}@mail.upb.de

Ramin Khalili* and Artur Hecker
Huawei
{ramin.khalili, artur.hecker}@huawei.com
*Corresponding authors

***Abstract*—Modern services comprise interconnected components, e.g., microservices in a service mesh, that can scale and run on multiple nodes across the network on demand. To process incoming traffic, service components have to be instantiated and traffic assigned to these instances, taking capacities and changing demands 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 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 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, it significantly improves flow throughput and overall network utility on real-world network topologies and traffic traces. It also learns to optimize different objectives, generalizes to scenarios with unseen, stochastic traffic patterns, and scales to large real-world networks.

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 or chained virtual network functions (VNFs) in network function virtualization (NFV) [3]. Each component can run on any node in the network and scale flexibly according to the current demand. Service coordination requires online decisions how to scale and where to instantiate each component as well as how to schedule incoming flows to these instances.

While service coordination has been the focus of intensive study [2], [4], most existing work 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. 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 and rely on carefully designed models tailored to specific scenarios and built on corresponding assumptions. Applying them to scenarios with slightly different assumptions or new optimization objectives (e.g., 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 practice, complete knowledge about current load and incoming traffic is not available instantly or even a priori but only after monitoring, often done periodically (e.g., default 1 min in Prometheus [5]). 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). We train a DRL agent offline, where it learns just from interaction with the network environment, using its previous actions and experience as feedback. The trained DRL agent then autonomously provisions services without built-in expert knowledge. It decides and periodically adapts rules for dynamic runtime flow scheduling and derives service scaling and placement accordingly in a centralized, online approach. In doing so, it does not require a priori traffic or complete network knowledge. Instead, it relies on monitoring data that is only intermittently available and that contains only aggregated, slightly delayed, and uncertain information. The trained agent can handle stochastic and bursty traffic, optimize QoS, generalize to new scenarios,

978-3-903176-31-7

and scale to large, real-world network topologies while making decisions within milliseconds. It does so not by deciding on each flow individually but rather by determining and updating rules for each node in the network. These rules are then used purely locally to decide how to treat an actual flow.

Overall, our contributions are:

- We define the problem of online service provisioning and coordination based on available monitoring information.
- We address the problem by formalizing a partially observable Markov decision process (POMDP), which we use for our novel, self-learning DRL approach.
- Our DRL approach consistently outperforms state-of-the-art heuristics, requiring much less 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.
- For reproducibility, we make our code publicly available [6].

II. Related work

Many researchers have addressed service scaling and placement, often in cloud or edge computing [2], [7] or NFV [4], using established optimization approaches without DRL. For example, multiple authors [8]–[11] place services offline, assuming full a priori traffic knowledge. Other authors [12]–[14] consider online service scaling and placement but disregard runtime scheduling of flows. Blöcher et al. [15] do schedule flows dynamically at runtime but assume a given fixed placement. In contrast, we dynamically scale and place services and schedule flows jointly online. Such joint coordination is important to successfully balance trade-offs [16], [17].

In recent years, multiple authors used machine learning for traffic prediction [18] and pro-active service coordination [19], [20]. We see this as promising, complementary research direction that could be combined with DRL in future work.

Recently, first DRL approaches related to our work have been proposed. Multiple authors address online service placement under changing load [21]–[24], some considering stochastic traffic and QoS [23]. In contrast to our approach, Pei et al. [21] 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. Wang et al. [22] 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. [23] and Quang et al. [24] 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 [23] or apply a separate heuristic algorithm to fix the DRL agent's proposed solution [24]. 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. [25] 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 [26] do consider delayed and partial observations but focus on power allocation in wireless networks.

The recent work by Gu et al. [27] is most related to our work. Like us, they consider service coordination with flow scheduling, build on DDPG [28], and optimize a generic network utility. Still, there are three main differences: 1) Gu et al. focus on compute costs, whereas we concentrate on compute capacities (cf. edge computing) and consider link delays for optimizing QoS. 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.

III. Problem Statement

We consider the problem of online service provisioning and coordination in a network of geographically distributed nodes.

A. Problem Inputs

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 compute resource (e.g., CPU), which can be extended easily to multiple resource types. Each link $l \in L$ interconnects two nodes and 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 small flows at ingress nodes in the network (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. After traversing the requested service, flows may leave the network at any node. 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). Components can be

instantiated at multiple different nodes, where 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.

Note that these problem inputs are typically not known completely when solving the problem. In practice, link delays, flow or service characteristics may be unknown or uncertain.

B. Decision Variables and Network State

We consider centralized 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). We explain 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, the node's operating system or systems like Kubernetes [29] start and scale instances of a component c transparently if $x_{c,v}(t) = 1$.

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 include the number and aggregated rate of incoming, processed, and dropped flows at v, the average end-to-end delay of completed flows d_{avg}, as well as the peak resource usage $r_v(t) \in [0, \text{cap}_v]$ at v 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.

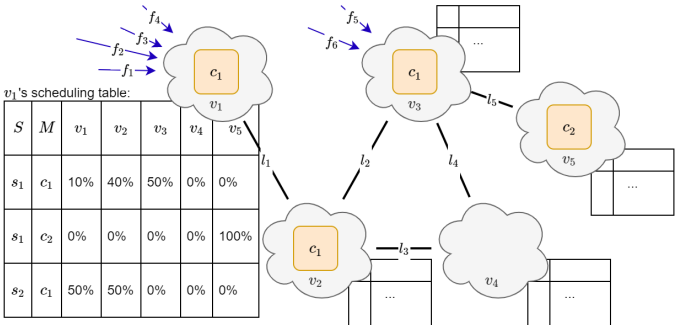
Used resources $r_v(t)$ depend on decisions $x_{c,v}(t)$ and $y_{f,c}(t)$ as well as flow length δ_f and requested data rate λ_f. Consequently, $r_v(t)$ increases with the total data rate of flows processed by instances at v. Flows are dropped if they cannot be processed, e.g., because there is no instance of requested component c or because v's resources are already fully utilized.

C. Objectives

We optimize the long-term utility $U_T = \sum_i w_i o_i$ over all T time steps, which consists of weighted (by w_i) objectives o_i that reflect the desired coordination goals. Here, we maximize utility $U_T = w_f o_f + w_d o_d$ of two objectives o_f and o_d with

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

$$o_d = \max\left\{-1, \frac{-d_{\text{avg}}}{D} + 1\right\} \in [-1, 1] \tag{2}$$



S	M	v_1	v_2	v_3	v_4	v_5
s_1	c_1	10%	40%	50%	0%	0%
s_1	c_2	0%	0%	0%	0%	100%
s_2	c_1	50%	50%	0%	0%	0%

Fig. 1: Flows continuously arrive at ingress nodes and are scheduled according to each node's scheduling table.

Objective o_f is the total amount of successful vs. dropped flows over T time steps. It encourages more successful flows F_{succ} and thus higher throughput. Objective o_d minimizes the average end-to-end delay per completed flow d_{avg} in time T. Objectives o_f 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.

To ensure that o_d and o_f are in the same range $[-1, 1]$, 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 the impact of weights w_f and w_d in Sec. V-D.

IV. DRL Approach

We address service coordination using model-free DRL. Our DRL agent does not know the network topology, link delays, service or per-flow details. Instead, it relies on incomplete, slightly delayed, and uncertain information 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 partially observable Markov decision process (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 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. The DRL agent 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 [30] 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. 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.

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 $c \in C_{s_f}$. 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 [31].

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 Markov decision process with partial observability (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 $\mathcal{O} = \langle \{\lambda_{v,s} | v \in V, s \in S\}, \{u_v | v \in V\} \rangle$, where $\lambda_{v,s}$ is the average ingress data rate and u_v is the node load during the last Δ time steps. Specifically, $\lambda_{v,s}$ 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, $\lambda_{v,s} = 0$. Node load $u_v = r_v(t)/\text{cap}_v \in [0, 1]$ is the resource usage at v over the last Δ time steps, normalized to the capacity of node v. If $\text{cap}_v = 0$, we set $u_v = 1$ to indicate that the node cannot be used.

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, we use $U_\Delta = w_f o_f + w_d o_d$ as defined in Sec. III-C but referring only to the utility of the last Δ time steps. The DRL agent maximizes the sum of discounted rewards to optimize the long-term utility. To avoid that the agent simply drops all flows when trying to optimize o_d, we initialize $o_d = -1$.

C. DRL Service Coordination

Our DRL algorithm is based on deep deterministic policy gradient (DDPG) [28], which can handle large, continuous action spaces such as $\mathcal{A}$ in our POMDP. DDPG is an off-policy actor-critic algorithm, i.e., it learns from buffered batches of previous experience using neural networks for both

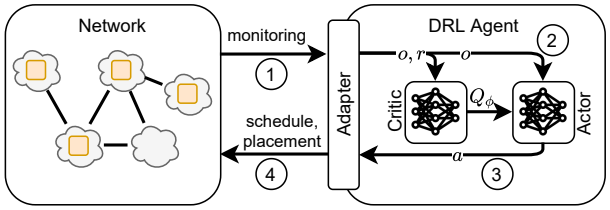


Fig. 2: DRL service coordination framework.

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 the DRL agent offline until convergence and then apply the trained agent online (inference). Fig. 2 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 to connect our DRL agent 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 [32]. As a simple countermeasure, 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 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).

Algorithm 1 DRL 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: $b \leftarrow$ sample(B, N)
8: train Q_ϕ minimizing the Bellman error [28]
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)

Offline training of our DRL agent is time-intensive and depends on random exploration. In contrast, online inference is very fast [33]. 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.

We implemented our DRL approach using Python and publish it in an open-source repository [6]. For production deployment, systems like Prometheus [5] and Kubernetes [29] 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 [34].

V. Numerical Evaluation

A. Evaluation Setup

1) Evaluation Scenarios: We perform extensive simulations on real-world network topology Abilene [36], 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. Each network has heterogeneous node capacities cap_v between 0 and 2 compute units (e.g., CPU cores), assigned uniformly and independently at random. Link delays are based on the distance between connected nodes. While we successfully tested our approach with multiple services, for simplicity, we focus on coordination of 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 linear to their processed data rate. 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 the DRL agent 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 retrain 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 our DRL approach.

For both actor and critic, we train dense neural networks with a single fully connected hidden layer (64 nodes, ReLU [37]) using the Adam optimizer [38]. 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 state-of-the-art heuristic, BSP, for joint scaling, placement, and flow assignment from our previous work [14].
- 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. [25]. All three algorithms choose actions from action space $\mathcal{A}$—but, unlike our DRL approach, 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 our approach, related DRL approaches (Sec. II) are not available publicly. Thus, a direct comparison is difficult.

4) Execution: We repeated all experiments with 30 different random seeds on machines with Intel Xeon W-2145 and 32 GB RAM. Figures show the mean and standard deviation.

B. Maximizing Successful Flows

Here, we focus on maximizing successful flows ($w_f = 1$ and $w_d = 0$ in U_T) in the Abilene network. As evaluation parameters, 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. As evaluation metric, we compare the percentage of successful flows after T time steps for each algorithm.

The simplest traffic pattern we consider is *fixed flow arrival*, where flows arrive in fixed intervals (10 time steps) at each ingress. Fig. 3a shows the percentage of successful flows for the different algorithms (legend in Fig. 3d). 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. Our DRL approach outperforms all other algorithms. It successfully processes all flows even in the highly saturated network with 4 ingress nodes and drops fewer flows with 5 ingress nodes (83 % more successful flows than adapted BSP).

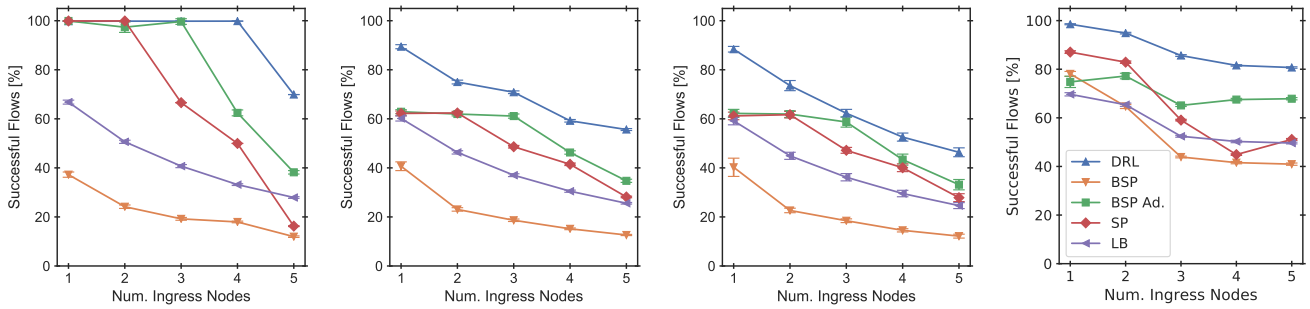
Fig. 3b shows the results for *Poisson flow arrival* (mean: 10 time steps). Due to the randomness in flow arrival, multiple flows may arrive directly after another in bursts, which can easily lead to dropped flows. Again, adapted BSP is slightly better than SP and much better than default BSP. LB still performs worse but more comparable to the other algorithms. Still, our DRL approach outperforms all algorithms. Compared to adapted BSP, it processes up to 60 % 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 realistic flow arrival following a *Markov-modulated Poisson process (MMPP)* [39]. 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. 3c shows that our DRL approach handles MMPP flow arrival well and, again, outperforms the other algorithms.

Finally, Fig. 3d shows the results for flows following *real-world traffic traces* that were recorded for the Abilene network [40]. To simulate increasing load, we enable an increasing number of ingress nodes where the trace-driven traffic arrives. Our DRL approach handles this real-world traffic well and clearly outperforms all other algorithms.

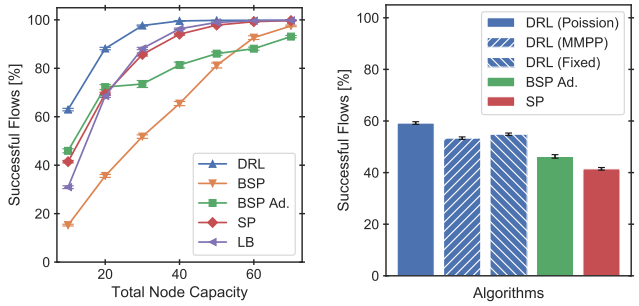
C. Generalization to Unseen Scenarios

For the different scenarios of Sec. V-B, we always retrain the DRL agent but reuse the same hyperparameter settings.



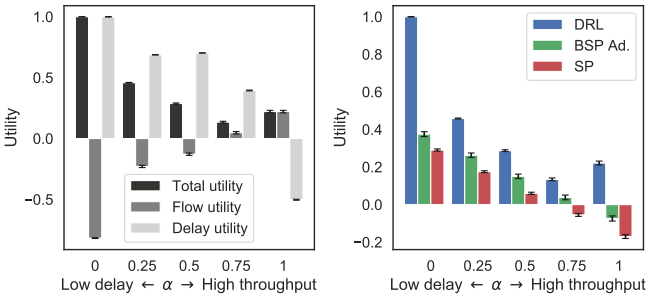
(a) Fixed Arrival (b) Poisson Arrival (c) MMPP Arrival (d) Real-World Traffic Trace

Fig. 3: Our DRL approach processes most flows successfully with increasing load at different flow arrival patterns.



(a) Changed Node Capacity (b) Unseen Traffic Pattern

Fig. 4: Generalization to new, previously unseen scenarios.



(a) DRL: Total, Flow, Delay Utility (b) Algorithm Comparison

Fig. 5: Utility with varying objective trade-off α.

This allows to fully automate training and applying the DRL agent 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 traffic or capacity. 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 based on generally available information and normalize observations, actions, and rewards to be in a similar range (Sec. IV-B).

We investigate generalization of a DRL agent to scenarios with gradually increasing node capacity. In particular, we train the DRL agent on Poisson traffic, 4 ingress nodes, and a total node capacity of 10 compute units and test it with evenly increased node capacity. Fig. 4a shows that our DRL agent generalizes well to different capacities and still outperforms all other algorithms. In comparison, the other algorithms need at least 50 % more resources to reach high success rates of 99 %.

We also study generalization of a trained DRL agent to scenarios with new traffic patterns. Specifically, we train one agent on fixed flow arrival and another on MMPP flow arrival and confront both with previously unseen Poisson flow arrival. Fig. 4b shows the successful flows for both cases in the Abilene network with 4 ingress nodes. For comparison, we also show results of a DRL agent trained on Poisson traffic and of adapted BSP and SP. The figure indicates that our trained DRL agent 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 optimizing the number of successful flows ($w_f = 1, w_d = 0$). In practice, operators often want to optimize multiple objectives. Here, we evaluate service coordination with varying objective weights w_f and w_d. These weights determine the trade-off $\alpha = w_f = 1 - w_d \in [0, 1]$ of many successful flows (and high throughput) vs. short delays, which are often opposing objectives.

Fig. 5a shows results of DRL agents trained with varying α in the case of Poisson traffic and 4 ingress nodes. The figure shows the flow utility o_f, delay utility o_d, and total, weighted utility U_T. Clearly, α affects how our DRL approach coordinates services. As desired, higher α leads to better flow but worse delay utility. With decreasing α, it favors shorter delays at the cost of more dropped flows. The agent trained

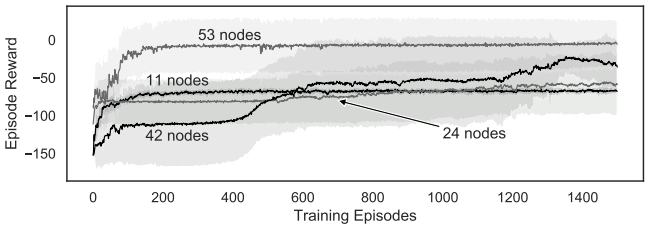
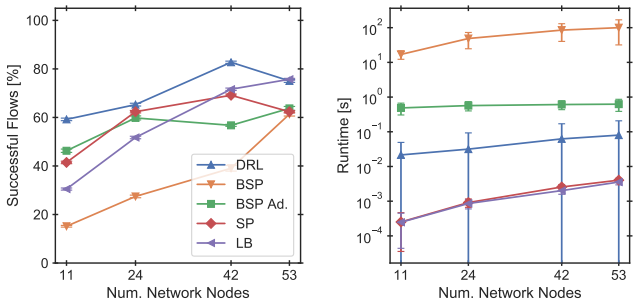


Fig. 6: DRL learning curves for networks of varying sizes.



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

Fig. 7: Comparison of service coordination quality (successful flows) and runtime with increasing network size and capacity.

with $\alpha = 0$ drops most flows but processes the remaining successful flows with optimal delay. Compared to the other algorithms, DRL achieves better total utility for all α values (Fig. 5b). Hence, while α has to be chosen carefully, our DRL agent can successfully optimize multiple weighted objectives.

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) [36], each with 4 ingress nodes and Poisson traffic arrival (mean inter-arrival time: 10 time steps).

Fig. 6 shows the learning curves for training our DRL approach 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 the DRL agent 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 500 and 1300. With much more training, we expect similar leaps in performance for 53 nodes. 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 [41]. E.g., Deepmind's famous AlphaGo Zero was trained over almost 5 million games [42]. 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 and curriculum learning [43].

Despite limited offline training, our trained DRL agent can compete with or even outperform all baseline algorithms. Fig. 7a compares the algorithms' percentage of successful flows (with $\alpha = 1$). 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, our DRL approach processes as much or even more flows successfully. The difference is especially large for 42 nodes, where performance leaped two times during training (see Fig. 6). We believe that considerably more training (e.g., in a commercial setting) could result in similar performance leaps and even better results for 53 nodes.

In addition to quality, runtime of online coordination decisions is crucial to quickly adapt to changes. Fig. 7b 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 online only requires milliseconds and is much faster than the (adapted) BSP heuristic. The trained DRL agent scales well to large networks without significant increase in runtime.

VI. Conclusion

Our DRL approach 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. Hence, we believe our approach is an important step towards truly driverless, self-learning networks and thus towards higher efficiency, more flexibility, and improved reliability.

Acknowledgments

We 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 the 5G-PPP project FUDGE-5G (H2020-ICT-42-2020 call, grant 957242). The expressed views are those of the authors and do not necessarily represent these projects.

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] 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.

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

[6] S. Schneider, A. Manzoor, H. Qarawlus, and S. Uthe, "DRL network service coordination GitHub repository," https://github.com/RealVNF/deep-rl-network-service-coordination (June 22, 2020), 2020.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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), to appear*. IEEE, 2020.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

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

[36] 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.

[37] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011, pp. 315–323.

[38] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *International Conference for Learning Representations*, 2015.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.