

## Research article

# Attention-based model and deep reinforcement learning for distribution of event processing tasks

Andriy Mazayev<sup>a,\*</sup>, Faroq Al-Tam<sup>b</sup>, Noélia Correia<sup>a,c</sup>

<sup>a</sup> Centre of Electronics, Optoelectronics and Telecommunications (CEOT), University of Algarve, 8005-139 Faro, Portugal

<sup>b</sup> Capgemini Engineering, Portugal

<sup>c</sup> Faculty of Science and Technology, University of Algarve, 8005-139 Faro, Portugal

## ARTICLE INFO

## Keywords:

Web of Things (WoT)  
 Representational state transfer (REST)  
 application programming interface (APIs)  
 Edge computing  
 Load balancing  
 Resource placement  
 Deep reinforcement learning  
 Transformer model  
 Pointer networks  
 Actor critic

## ABSTRACT

Event processing is the cornerstone of the dynamic and responsive Internet of Things (IoT). Recent approaches in this area are based on representational state transfer (REST) principles, which allow event processing tasks to be placed at any device that follows the same principles. However, the tasks should be properly distributed among edge devices to ensure fair resources utilization and guarantee seamless execution. This article investigates the use of deep learning to fairly distribute the tasks. An attention-based neural network model is proposed to generate efficient load balancing solutions under different scenarios. The proposed model is based on the Transformer and Pointer Network architectures, and is trained by an advantage actor-critic reinforcement learning algorithm. The model is designed to scale to the number of event processing tasks and the number of edge devices, with no need for hyperparameters re-tuning or even retraining. Extensive experimental results show that the proposed model outperforms conventional heuristics in many key performance indicators. The generic design and the obtained results show that the proposed model can potentially be applied to several other load balancing problem variations, which makes the proposal an attractive option to be used in real-world scenarios due to its scalability and efficiency.

## 1. Introduction

Event processing is a crucial element in dynamic and reactive Internet of Things (IoT) applications, as it allows to derive real-time (or near-real-time) conclusions from data. An event processing task, usually mapped into a web request (e.g., HTTP), follows a simple observe-evaluate-actuate pattern. As the name suggests, this pattern involves performing three tasks: observe one or many devices/sensors, evaluate the produced data and, if the user-defined condition is satisfied, notify one or many actuators. With the recent emergence of the Web of Things (WoT) [1–4] concept, where every device has its own internet protocol (IP) address and every device resource is addressable by uniform resource identifier (URI), the observe-evaluate-actuate chain operates over a set of URI endpoints.

Event processing has relied primarily on cloud computing, which virtually has an unlimited amount of storage and computational resources. Centralized processing, alongside with the virtualization of both storage and computational resources, offers flexibility and efficiency by scaling up and adapting the system to different situations. However, with the ever-increasing amount of data being produced and consumed directly at the network edge, centralized (cloud) computing ends up becoming a limitation because

\* Corresponding author.

E-mail address: [amazayev@ualg.pt](mailto:amazayev@ualg.pt) (A. Mazayev).

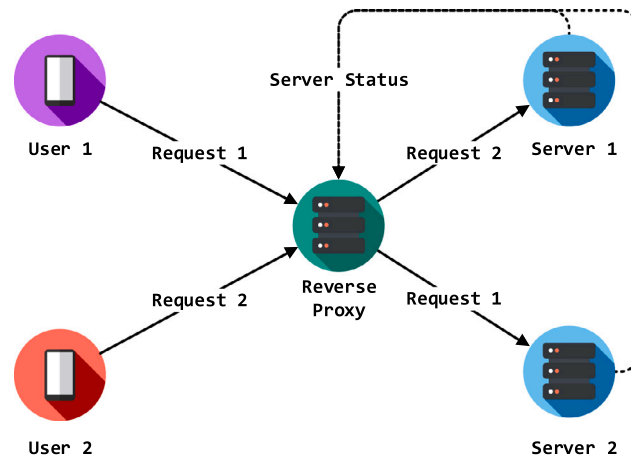


Fig. 1. An overview of reverse proxy.

additional IoT application requirements cannot be satisfied. These requirements are related with delay sensitivity, data volume, uplink costs, non-interruption of service in environments with intermittent connectivity, privacy, and security [5].

The need to satisfy the above-mentioned requirements led to the emergence of a distributed computing paradigm, called edge computing. In this new paradigm, substantial computational and data storage resources are located near the mobile devices, sensors and actuators. Edge computing can be seen as a strategy to provide a uniform computation/storage environment all the way from the core data centers to physical locations near users and data. However, unlike the cloud, edge computing has a more dynamic and distributed computing nature, which results in distributed and ad-hoc event processing chains.

Despite the above-mentioned differences, edge and cloud will have a common background in the underlying technologies. In an Internet-Draft called “IoT Edge Challenges and Functions” [6] Internet Engineering Task Force (IETF) emphasizes that “*virtualization platforms enable the deployment of virtual edge computing functions, including IoT gateway software, on servers in the mobile network infrastructure, in edge or regional data centers*”. At the moment of writing, there are several initiatives that are moving in this direction. As an example, Edge Virtualization Engine (EVE) [7] aims to create an open edge computing engine that enables the development, orchestration and security of cloud-native and legacy applications on distributed edge computing nodes. It also aims to provide support for containers and clusters (e.g., Docker, Kubernetes), virtual machines, unikernels and offer a flexible foundation for IoT edge deployments with choice of any hardware, application and cloud. Another example is the EdgeX Foundry [8] service platform that also aims to provide management, data persistence or streaming near the edge. This project also provides a dedicated rule engine API for the creation of event processing chains. MobileEdgeX [9] is an edge-cloud platform that provides an environment for the developers to deploy software (e.g., as software containers) on the edge. All these initiatives seek to bring orchestration, virtualization, load balancing techniques and technologies from the cloud to the edge. This means that in order to understand how event processing tasks will be handled and distributed across the edge, it is important to see how this is currently being handled by the cloud.

### 1.1. Adequate resource placement and load balancing

In typical application deployments, devices/servers (virtual or physical) are placed behind reverse proxies like NGINX [10], Traefik [11] or Moleculer API Gateway [12]. A reverse proxy, illustrated in Fig. 1, is a type of server that typically sits behind the firewall in a private network and distributes the incoming client requests among the available back-end servers. Common features that reverse proxies provide include load balancing, web acceleration (e.g., caching, SSL encryption), security and anonymity. In case of load balancing there are several balancing strategies that can be used, each focusing on a specific aspect of Quality-of-Service (QoS). NGINX, for example, offers round-robin, least-connected, ip-hash [13]; Traefik at the moment only supports round-robin method [14]; Moleculer API Gateway offers round-robin, random, CPU-usage based and sharding [15]. These load balancing strategies do not provide optimal solutions because these are prohibitively expensive to obtain in real-time. Instead, these strategies trade the quality of solution for the response time, i.e., these strategies are fast but the solutions that they provide can be sub-optimal. Moreover, static optimization strategies are not adequate because the environment is dynamic in terms of available servers and user requests.

The arrival of new user requests is usually a stochastic process [16]. Furthermore, the number of incoming requests can change abruptly. That is, bursts of requests during short periods of time can be followed by periods of slowdown. To address this, containerized environments have a container manager (e.g., Kubernetes) that provides autoscaling of the number of servers, ensuring the creation (or removal) of replicas according to the system load [17]. Similar dynamics are expected at the network edge, and projects like StarlingX [18] solve them by providing tools for management, orchestration and scaling of distributed edge-cloud computing devices.

From the above discussion, it is clear that load balancing and resource planning strategies will play a crucial role in edge computing, together with strategies for the appropriate placement of event processing tasks. Any strategy should be scalable and adaptable to the dynamics of the system: number of incoming requests, number of servers and resources available at that time. In general, the problem of distributing event processing tasks boils down to typical NP-hard problems like the multiple knapsack or bin packing problem [19,20]. Developing handcrafted heuristics for such problems can be challenging, especially under dynamic environments and heterogeneous key performance indicators (KPIs). On the other hand, machine learning (ML) solutions are envisioned as the right answer to an ever increasing complexity of networks and the web [21]. ML approaches are able to learn new distribution strategies which makes them a compelling choice. Moreover, toolsets like Tensorflow Lite<sup>1</sup> that provide means for compressing and deploying ML solutions at mobile phones, micro-controllers and edge Tensor Processing Units (TPU) (e.g., Coral Edge TPU) make ML even more attractive. Bringing ML to the edge can unlock a whole new set of applications that are not possible with cloud solutions [22].

### 1.2. Reinforcement learning as a solution framework

The most successful applications of ML methods (e.g. natural language processing (NLP), computer vision) fall under the umbrella of supervised learning (SL), a process that consists in learning to map a set of inputs to output labels. However, applying supervised ML methods to placement problems, or any other combinatorial problem, is troublesome as it is almost always impossible to obtain the output labels. Moreover, even if the labels are provided and a neural network model is trained in a supervised way, it usually has poor generalization to other problem instances [23].

Reinforcement learning (RL), on the other hand, is an attractive approach to solve combinatorial problems because, unlike SL, it does not require labels. RL can be seen as a trial and error process in which an agent interacts with an environment via a sequence of actions and, in return, receives feedback in the form of reward signals. The goal of the agent is to learn an action selection strategy in a way that the total reward is maximized. A continuous interaction with the environment allows the agent to explore several (millions of) possible environment configurations and to develop a strategy that ensures that the maximum reward is obtained in every possible situation, i.e., to generalize.

Recent attempts in integrating deep learning (DL) with RL showed breakthrough results in video games [24], board games [25, 26], robotics and many more areas [27]. This new learning paradigm is widely known as deep reinforcement learning (DRL). In this work we investigate how DRL can be used by reverse proxy servers to distribute event processing tasks among heterogeneous edge devices available for processing.

### 1.3. Motivation and contributions

The idea of using DRL to solve load balancing and distribution problems has recently attracted many researchers. Surveys show that there are several attempts, with different levels of success, to solve these kind of problems [28,29]. However, the vast majority of these attempts do not solve problems of variable sizes. The problem is treated as a fixed-sized optimization problem where a neural network is trained to map problem instances to solutions. For instance, in the distribution problem illustrated in Fig. 1, the number of incoming requests and the number of available servers are assumed to be fixed. Then, during the development (design and training), the model is trained to learn to map the inputs to outputs. This is followed by testing, where the network's performance is measured on solving problems of the same size that it was trained upon and, finally, reporting the obtained results. However, despite the results reported in the above mentioned surveys, the majority of those models have little practical use in real life deployment because a simple change in input size would make those models invalid.

We argue that the assumption of a fixed problem size is problematic and, usually, does not hold true in dynamic environments like IoT and edge. The number of incoming requests and available nodes is dynamic and changes over time. Therefore, the main limitation of the existing literature that involves DRL, load balancing and distribution problems is the inability of the proposed neural networks to adapt to the dynamics of the environment without retraining. In other words, the problem is the scalability of the proposed neural networks.

This work focuses on developing a scalable DRL model that is able to solve problem instances of variable size without retraining. The main contributions of this article are as follows:

1. A study is made on how event processing tasks can be distributed across edge devices. Three different distribution criteria are studied, each being formalized mathematically.
2. A new neural network architecture is proposed for the distribution of event processing tasks, while ensuring load balancing. The key feature of the proposed model is its scalability, which means that it does not require retraining every time the number of incoming requests or servers, available for processing, change.
3. A comparative study against the optimal solutions and several baseline heuristics is carried out. Obtained results indicate that the proposed model can generate high quality solutions to problems up to five times larger than the ones it was trained upon.

<sup>1</sup> <https://www.tensorflow.org/lite>.

The remainder of this article is organized as follows. In Section 2, existing attempts of using RL in load balancing problems are discussed. A review of neural combinatorial optimization is also carried out. The event task distribution problem is mathematically formulated in Section 3. Section 4 presents the proposed neural network architecture and is organized as follows: (i) Section 4.1 presets a brief RL and DNN background; (ii) Section 4.2 presents the Markov decision process (MDP) representation of the problem; (iii) Section 4.3 describes the detailed design of the proposed neural network architecture. Section 5 describes the training method alongside with the hyperparameters used to obtain the model and evaluates its performance. Finally, in Section 6 conclusions are drawn.

## 2. Related work

RL has been applied in several areas of IoT and edge computing for different purposes, which range from the actuation control (e.g., greenhouse climate control) to resource control (e.g., minimization of communication delay, energy consumption, hardware resources) [28,29]. However, most of the literature uses deep neural network (DNN) architectures that are not able to scale or adapt to the dynamics of the environment, e.g., increase in the number of edge devices, base stations, sensors or number of requests. That is, the action space is fixed, which limits the practical use of DNNs in realistic scenarios. The following subsections provide an overview of the application of RL to IoT, edge computing and load balancing. Then, ongoing research to tackle combinatorial optimization problems, using RL, is presented [30,31].

### 2.1. Reinforcement learning in edge computing

In [32] authors propose a DRL solution to minimize the total cotask completion time. A cotask is a task, generated by an IoT device, that can only be completed when all of its constituent sub-tasks are finished. Authors use a DNN architecture with two heads, one that predicts the offloading location of the task and another for the prediction of the cotask completion time. The obtained results show that the proposed model is able to outperform several baseline heuristics. However, the proposed architecture requires retraining whenever the network or the input size changes.

In [33] an RL method is used to distribute the tasks, produced by different user devices, among a set of edge servers. To deal with the combinatorial action space (any task can be placed at any available edge server), authors introduce a multi-agent algorithm where each Deep Q-Network (DQN) [24] agent makes an action that corresponds to the location where the task will be offloaded. The DQN architecture, used in this work, requires retraining once the state or action space changes, i.e., once an additional edge server is connected to the system.

In [34] mobility load balancing in self-organizing networks is investigated. The goal is to design a handover scheme that transfers a mobile user from its service cell to a neighbor cell that can handle the incoming traffic. The authors use an off-policy deterministic policy gradient (OPDPG) method [35] and propose an asynchronous parallel learning framework to improve the training efficiency in a collaborative manner. The obtained results show that the RL method is able to outperform several baseline heuristics. However, once again, this architecture is designed for fixed-size communication networks, i.e., any change in the number of cells requires retraining the network.

In [36] the authors propose an RL agent to offload computationally intensive tasks, generated by user equipment (UE), to mobile edge computing (MEC) servers, while maximizing the mobile operator revenue and minimizing the energy consumption and time-delays. In the investigated environment the authors considered a queuing model, states of energy harvesting batteries and down-link transmit power costs. The authors conclude that their approach has better performance than the policy-gradient and Q-learning algorithms. The main limitation of this work is the fact that the authors only studied a scenario with a fixed number of MEC servers, which is not always the case in reality. Moreover, any change in the number of MEC servers requires retraining the agent.

Several other works [37–41] used DQN-based models in offloading and load balancing problems. Despite the obtained results, all of these proposals have a common weakness that drastically limits their use in real world deployments. Any change in the state or action space sizes imply retraining the network from scratch. Overall, DQN and its variants are more suitable for static environments with fixed state and action spaces but they have limited uses in dynamic environments

### 2.2. Combinatorial optimization with reinforcement learning

One of the first attempts to solve combinatorial problems with neural networks was presented in [42]. The authors propose a neural architecture, called Pointer Networks (Ptr-Net), which is based on sequence-to-sequence (seq2seq) [43] models with attention mechanisms [44,45] that are commonly used in NLP. Ptr-Net is an encoder–decoder architecture where recurrent neural networks (RNNs), usually long short-term memory network (LSTM) [46] or gated recurrent unit (GRU) [47], are used to process the input sequence at the encoder and to generate the output sequence at the decoder. Although in this work the Ptr-Net was trained in a supervised way, authors showed that a single architecture with the same hyperparameters can be used to solve different combinatorial problems. The fundamental breakthrough of Ptr-Net is its ability to deal with variable output space without the need for retraining the network. Moreover, the authors showed that the Ptr-Net trained on combinatorial problems of small size is capable to generalize to much larger problems without considerable degradation in the quality of generated solutions.

In [23] the authors used Ptr-Net in combination with RL to solve combinatorial problems, such as the traveling salesman problem (TSP) and the knapsack problem. Authors use RL methods because it is difficult to have access to labels (associated with optimal solutions), as combinatorial problems are usually NP-Hard. The obtained results show that the developed RL agent is capable of

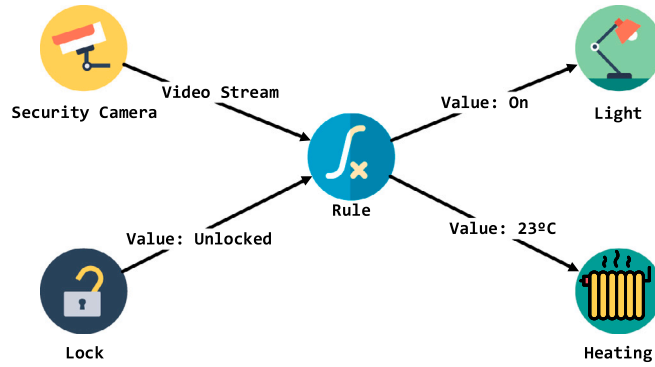


Fig. 2. An example of a Rule with two inputs and two outputs.

finding near-optimal solutions for TSP with up to 100 nodes and for up to 200 items in knapsack problem. In this work the authors also made an empirical analysis between supervised training and RL and reached the conclusion that Ptr-Net networks trained in a supervised way have poor generalization when compared to a Ptr-Net-based RL agent that explores different solutions and observes their corresponding rewards.

In [48] the authors proposed a model, inspired on Ptr-Nets and seq2seq, to solve vehicle routing problem (VRP). Authors removed LSTM from the encoder as they argue that LSTM, and other RNNs, are only necessary when dealing with sequential data. After training, the proposed model was able to generalize well and generate high-quality solutions for all problems sampled from the same distribution that was used during training. Moreover, authors tested the model's ability to handle variable problem sizes. They conclude that their model performs well when training and testing problem sizes are close to each other. However, they report a degradation in performance when the sizes of the testing instances are substantially different from the ones used for training.

The work in [48] is extended by [49], where a model capable of solving several variations of VRP and TSP is proposed. The authors replace LSTM networks by the Transformer [50] and results show that the proposed model is able to achieve (near) optimal solutions for problems of size 20, 50 and 100 nodes.

### 3. The problem of event processing

Prior to presenting the proposed DRL solution, it is necessary to clearly define the problem. This section briefly describes the technological background of event processing, which is then followed by its mathematical formalization.

#### 3.1. Rule synchronization mechanism

There are multiple Internet-Drafts that attempt to standardize event processing in WoT, many of them have been reviewed and analyzed in our previous work [51]. In this article the focus will be on the most recent proposal, called Rule, which was also discussed in [51].

A Rule, illustrated in Fig. 2, is a regular web resource, based on RESTful principles, that allows to create complex many-to-many event processing chains. Multiple sources can be observed (e.g., sensors) and, after condition evaluation, multiple destination endpoints can be updated (e.g., actuators). Being a regular web resource means that Rules can be placed at any device that supports a Rule API and is available for processing. The Rule is a continuously running task that is constantly observing resources and processing the data being sent. To stop it, and to release the resources that it is consuming, the user must send an appropriate request (e.g., HTTP DELETE).

End-users are not expected to create Rules by hand. Applications are expected to be available (e.g., Rules store) where users can find pre-defined Rules, together with related meta-data (e.g., CPU, RAM and storage requirements), i.e., Rule profiles. The inputs and outputs of a pre-configured Rule will have to be defined according to user's need.

Running Rules at the edge, where there will be physical, virtual, and containerized nodes with different computational capabilities, which means that these tasks must be carefully distributed.

**Definition 1 (Rule Distribution (RD) Problem).** Given a set of Rule profiles, decide for the best Rule distribution across devices while taking into account randomly arriving Rule requests, random number of available edge devices (each having its random amount of available CPU, RAM and storage resources) and QoS requirements.

### 3.2. Assumptions and notation

To have a clear understanding of how to solve the RD problem optimally, assumptions and notation have to be clarified. This is followed by the mathematical formalization of the RD problem considering three variants. Such mathematical formalization is important not only to clearly define the problem, but also to have a reference when evaluating the proposed RL-based agent.

**Definition 2 (Node).** A virtual, containerized or physical device exposing a Rule API, being capable of processing a specific set of Rules. The set of nodes able to host Rule  $x$  is denoted by  $\mathcal{N}^x$ , while the overall set of heterogeneous nodes is denoted by  $\mathcal{N}$ .

**Definition 3 (Computational Resource).** Hardware resource (e.g., CPU, RAM, memory) available at nodes. The amount/capacity of  $n$ 's computational resources,  $n \in \mathcal{N}$ , is given by vector  $\mathbf{c}^n = [c_{n,m}]$ , where  $m \in \{1, \dots, M\}$  and  $M$  is the total number of computational resources considered.

**Definition 4 (Rule).** Web resource able to evaluate an expression or performing complex processing over certain inputs. Assuming  $\mathcal{X} = \{x_1, x_2, \dots, x_{|\mathcal{X}|}\}$  to be a set of Rules, each  $x_i$  has a demand vector  $\mathbf{d}^{x_i} = [d_{x_i,m}]$ ,  $m \in \{1, \dots, M\}$ , containing computational resource requirements.

**Definition 5 (Critical Resource).** Given a set of rules  $\mathcal{X}' \subset \mathcal{X}$  under execution at node  $n \in \mathcal{N}$ ,  $n$ 's critical resource is given by  $\Omega^n = \operatorname{argmax}_{m \in \{1, \dots, M\}} \{ \frac{\sum_{x' \in \mathcal{X}'} d_{x',m}}{c_{n,m}} \}$ .

In the following section three variants of the RD problem are formalized. Each variant tries to optimize specific performance indicators: number of allocated Rules, critical resource and/or number of active nodes. The variables that may be involved are the following:

$\omega_x$	One if Rule $x \in \mathcal{X}$ is to be executed, which requires finding a node to place it; zero otherwise.
$\beta_x^n$	One if Rule $x \in \mathcal{X}$ is to be placed at node $n \in \mathcal{N}^x$ ; zero otherwise.
$\Omega^n$	Critical resource at node $n \in \mathcal{N}$ .
$\Omega^{\text{MAX}}$	Most critical resource among all nodes at the network.
$\Theta^n$	One if node $n \in \mathcal{N}$ is in use; zero otherwise.

### 3.3. Mathematical formalization

#### 3.3.1. Greedy optimizer

The performance indicator to be optimized is the number of Rules placed/distributed across nodes, which should be maximized. This is expressed as follows.

– Objective Function:

$$\text{Maximize } \sum_{\{x \in \mathcal{X}\}} \omega_x \quad (1)$$

– Placement of Rules:

$$\sum_{\{n \in \mathcal{N}^x\}} \beta_x^n = \omega_x, \forall x \in \mathcal{X} \quad (2)$$

These constraints ensure that there is a single location for each Rule that is to be executed.

– Computational Resource Limitation:

$$\sum_{\{x \in \mathcal{X}\}} d_{x,m} \times \beta_x^n \leq c_{n,m}, \forall n \in \mathcal{N}, \forall m \in \{1, \dots, M\} \quad (3)$$

where  $d_{x,m}$  is known information. These constraints avoid exceeding the available computational resources of nodes.

– Non-negativity assignment to variables:

$$\omega_x, \beta_x^n \in \{0, 1\}. \quad (4)$$

#### 3.3.2. Critical-aware greedy optimizer

The main performance indicator to be optimized is the number of Rules that are placed/distributed across nodes, which should be maximized while ensuring that load is fairly distributed (most critical resource as a reference). Thus, fair distribution is the secondary performance indicator to be optimized. This kind of optimization is important in cooperative scenarios when edge devices are provided by different owners that trade their computational resources for some reward (e.g., financial benefits). Service providers hold reverse proxy servers, which exposes the Rule API to the end-users. The computational resources are provided by the device owners.

– Objective Function:

$$\text{Maximize } \sum_{\{x \in \mathcal{X}\}} \omega_x + \Omega^{\text{MAX}} \quad (5)$$

This objective function ensures that the largest number of Rules is placed/distributed and, at the same time, that it is done in a fair way, so that Rules have the best (and similar) working conditions.

- Placement of Rules: Expression (2).
- Computational Resource Limitation: Expression (3).
- Obtaining Critical Resource:

$$\Omega^n \leq c_{n,m} - \sum_{x \in \mathcal{X}} d_{x,m} \times \beta_x^n, \forall n \in \mathcal{N}, \forall m \in \{1, \dots, M\} \quad (6)$$

where  $d_{x,m}$  is known information. These constraints finds the critical resource at each node. Finding the most critical resource among all nodes requires:

$$\Omega^{\text{MAX}} \leq \Omega^n, \forall n \in \mathcal{N} \quad (7)$$

- Non-negativity assignment to variables:

$$\omega_x, \beta_x^n \in \{0, 1\}; 0 \leq \Omega^n, \Omega^{\text{MAX}} \leq 1. \quad (8)$$

This mathematical optimization model looks for the largest possible number of Rules that got a place in a node, which is guaranteed by the first component of the objective function, and after reaching this number it ensures that the Rules are distributed in order to free the most critical resource among all nodes (second component). The last component ends up not competing with the first because  $\Omega^{\text{MAX}}$  reaches its maximum,  $\Omega^{\text{MAX}} = 1$ , only when all resources are free (no Rules placed). However, placing Rules increases the value of the objective function further. In short, no increase of  $\Omega^{\text{MAX}}$  is preferred over the placement of a single Rule.

### 3.3.3. Cost-aware greedy optimizer

The main performance indicator to be optimized is the number of Rules that are placed/distributed across nodes, which should be maximized while ensuring that the number of nodes in use (hardware cost) is minimized. Thus, the cost is the secondary performance indicator to be optimized. This optimization is important when the Rule API provider is responsible for both the reverse proxy and nodes, but is using rented hardware to offer its services. This is a very common scenario in cloud computing, so rental cost minimization ends up being also a relevant performance indicator.

- Objective Function:

$$\text{Maximize } \sum_{x \in \mathcal{X}} \omega_x - \frac{\sum_{n \in \mathcal{N}} \Theta^n}{|\mathcal{N}|} \quad (9)$$

This goal ensures that the largest number of Rules is placed/distributed using the lowest number of nodes, for cost minimization.

- Placement of Rules: Expression (2).
- Computational Resource Limitation: Expression (3).
- Obtaining nodes in use:

$$\Theta^n \geq \beta_x^n, \forall n \in \mathcal{N}, \forall x \in \mathcal{X} \quad (10)$$

$$\sum_{x \in \mathcal{X}} d_{x,m} \times \beta_x^n \leq c_{n,m}, \forall n \in \mathcal{N}, \forall m \in \{1, \dots, M\} \quad (11)$$

Constraints (11) avoid exceeding the available resources at nodes.

- Non-negativity assignment to variables:

$$\omega_x, \beta_x^n, \Theta^n \in \{0, 1\}. \quad (12)$$

This mathematical optimization model looks for the largest possible number of Rules that got a place in a node, and then it ensures that the smallest number of nodes are used. This is so because the second component of the objective function is negative, meaning that there is not interest in having  $\Theta^n$  variables set to 1.

The above-mentioned optimization problems provide the optimal solution for each problem instance. However, solvers (e.g., CPLEX [52], Gurobi [53]) that rely on these formalizations to compute optimal solutions can only find results, in a reasonable amount of time, if the problem instance is very small. Therefore, an RL-based solution is proposed RL methods have the ability to understand the system dynamics and, after trained, make appropriate placement decisions in real-time.

## 4. Proposed DRL framework

Prior to presenting the proposed DRL solution designed to tackle the RD problem, formalized in Section 3, this section first presents a brief RL and DNN background, which is then followed by the MDP formulation of RD problem.



#### 4.1. Deep reinforcement learning background

##### 4.1.1. Reinforcement learning

In RL, sequential decision problems can be modeled using Markov decision processes (MDPs). An MDP can be represented by the tuple  $(S, \mathcal{A}, p, r)$ , where  $S$  is the state space and  $\mathcal{A}$  is the action space. The  $p : S \times \mathcal{A} \times S \rightarrow [0, 1]$  is the probabilistic transition model (matrix), where  $p(s'|s, a)$  is the probability of arriving at state  $s'$  from  $s$  after taking action  $a$ , of which  $s', s \in S$  and  $a \in \mathcal{A}$ ;  $r : S \times \mathcal{A} \times S \rightarrow \mathbb{R}$  is the reward function.

An agent learns by interacting with the environment. At time step  $t$ , the agent observes a state  $s_t$  and takes an action  $a_t$ . The environment returns a reward signal  $r(s_t, a_t)$  to the agent and makes a transition to a next state  $s_{t+1}$ . The objective of the agent is to maximize the collected reward (outcome):

$$G_t = \left[ \sum_{k=0}^{\infty} \gamma^k r(s_{t+k}, a_{t+k}) \mid s_0 = s_t \right] \quad (13)$$

where  $\gamma$  is a discount factor used to balance the importance of immediate and long-term rewards.

For the agent to decide which action to take at a given state, it needs to follow a policy. Therefore, the objective of training an agent is to find a policy  $\pi$  that maximizes  $G_t$ . A policy is optimal  $\pi^*$  if following it produces the maximum outcome. In addition, a policy  $\pi$  can be deterministic i.e.,  $\pi(s) = a$ , or stochastic i.e., a probability distribution over the action space  $\mathcal{A}$ .

There are two ways to learn  $\pi$ : value-based and policy-gradient. This work focuses only on the latter.

In value-based learning,  $\pi$  is calculated from a state-value function  $V_{\pi}(s)$  which measures how good it is to be at a given state  $s$ :

$$V_{\pi}(s) = \mathbb{E} [G_t \mid s_t = s] \quad (14)$$

In policy gradient learning,  $\pi$  can be modeled by a parametric function where the objective is to learn a set of parameters  $\theta$  that maximizes an objective function  $J$ . Assume an episodic setting with episode length  $T$  and let  $\tau$  be a sequence of transitions (trajectory)  $\tau = s_0, a_0, \dots, s_{T-1}, a_{T-1}$ . Following the probability general product rule,  $\pi$  can be defined as:

$$\pi(\tau; \theta) = p(s_0) \prod_{t=0}^{T-1} \pi(a_t \mid s_t; \theta) p(s_{t+1} \mid s_t, a_t) \quad (15)$$

where  $p$  is, again, the transition probability density function and  $\theta$  is the set of parameters.

From Expression (15), it is possible to see that the transition from  $s$  to  $s_{t+1}$  depends on the action taken,  $a_t$ , and the probability of reaching  $s_{t+1}$  from  $s_t$  when taking  $a_t$ . Furthermore, following the original assumption about  $\pi^*$ :  $\pi^*$  will generate the optimal trajectory  $\tau^*$  that collects the maximum accumulated reward, the objective function  $J$  can be formulated as:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} [r(\tau)] = \int \pi(\tau; \theta) r(\tau), \quad (16)$$

and the optimal set of parameters can be found using:

$$\theta^* = \arg \max J(\theta) \quad (17)$$

To find  $\theta$ , the product in Expression (15) has to be simplified and differentiated with respect to  $\theta$ . Both can be done using the logarithmic differentiation:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} [\nabla \log(\pi(\tau; \theta)) r(\tau)] \quad (18)$$

Moreover, since  $r$  does not depend on  $\theta$ , we have:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^{T-1} \nabla \log(\pi(a_t \mid s_t; \theta)) G_t \right] \quad (19)$$

Following the gradient ascent rule, the update step is:

$$\theta_{k+1} = \theta_k + \alpha \nabla_{\theta} J(\theta_k) \quad (20)$$

where  $\alpha$  is the learning rate.

The steps of: (i) sampling a trajectory  $\tau \sim \pi_{\theta}$ ; (ii) differentiating with respect to  $\theta$  (Expression (19)); (iii) updating  $\theta$  (Expression (20)); are known as the REINFORCE algorithm [54,55]. This method, however, can suffer from high gradient variance due to trajectory sampling. A common way to reduce the variance is by subtracting a baseline value from the reward term in Expression (18). When the baseline is used, Expression (19) can be rewritten as:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[ \sum_{t=0}^{T-1} \nabla \log(\pi(a_t \mid s_t; \theta)) A(s_t, a_t) \right], \quad (21)$$

where  $A(s_t, a_t)$  is known as the advantage function [56]:

$$A(s_t, a_t) \approx r(s_t, a_t) + \gamma V_{\pi_{\theta}}(s_{t+1}) - V_{\pi_{\theta}}(s_t) \quad (22)$$



and  $V_{\pi_\theta}(s_t)$  is the actual baseline value (Expression (14)).

The method using the Expression (21) is called advantage actor critic (A2C) [57]. As the name suggests, this algorithm relies on an actor to learn the  $\pi_\theta$  and on the critic to learn the baseline values. An efficient way to model both the actor and the critic is by using DNNs. In this case, their losses can be defined as follows:

$$\text{actor loss: } - \sum_{t=0}^{T-1} \log(\pi(a_t|s_t; \theta)) A(s_t, a_t) \quad (23)$$

$$\text{entropy: } - \sum_{t=0}^{T-1} \pi(a_t|s_t) \log(\pi(a_t|s_t)) \quad (24)$$

where the *entropy* term is used to encourage exploration by penalizing agents' overconfidence. For the critic loss, the following expression is used:

$$\text{critic loss: } \frac{1}{T} \sum_{t=0}^{T-1} A(s_t, a_t)^2 \quad (25)$$

#### 4.1.2. Deep neural networks

As noted in Section 2, the models and architectures that are currently used by the RL community to tackle combinatorial problems are well-suited for handling problems with variable state and action spaces. Therefore, this work relies on the Ptr-Net and the Transformer, which are robust and flexible architectures. For brevity purposes, and due to space limitation, most of the mathematical foundations of these architectures are omitted and a quick overview is presented next. For further details please refer to [42,50].

**Pointer network architecture.** The main idea of Ptr-Net is that the output sequence is obtained by *pointing* at elements in the input, hence the name of the architecture. In other words, the output is a sequence of input selections, which makes Ptr-Net suitable for modeling selection problems. For example, this architecture was used to solve classical problems like the TSP and knapsack problem [42,58]. In both cases, the solution is built by selecting the sequence by which cities are visited (in case of TSP) or sequence by which items will be placed into the backpack (in case of knapsack).

This architecture has two main components: the *encoder* and the *decoder*. The encoder's role is to map the input into a feature space, while the decoder generates the pointers. In the encoding mode, the encoder reads the input data, one element at a time, and generates a set of fixed-dimensional vector representations, known as encoder hidden states ( $e_1, \dots, e_n$ ) where  $n$  is the number of input elements. After reaching the end of the input sequence, Ptr-Net switches into a decoding mode. In this mode, the decoder is initialized by the encoder's last hidden state  $e_n$  and a user-defined start of sequence (SoS) symbol. The decoder then produces the first hidden state  $d_1$ , which is used to generate the first pointer (i.e., index) to an element in the input sequence. In the next decoding step,  $d_1$  and the previously pointed element are fed to the decoder to generate a subsequent pointer. This process repeats itself until the decoder points to a user-defined end of sequence (EoS) symbol or a predefined number of pointers is reached. The decoder's hidden states are represented by  $(d_1, \dots, d_m)$  where  $m$  is the size of the output sequence.

The actual pointing mechanism, which we call Ptr-Net head, is computed as follows. First, at any decoding step  $i$  the unnormalized log probabilities (logits)  $u_i$  are calculated:

$$u_i^j = v^T \tanh(W_1 e_j + W_2 d_i), j \in (1, \dots, n) \quad (26)$$

where  $v^T, W_1, W_2$  are learnable parameters of the model.

Next,  $u_i$  is fed to a softmax layer to calculate the probabilities of pointers selection:

$$p_i = \text{softmax}(u_i) \quad (27)$$

where  $p_i$  is the softmax distribution with dictionary size equal to the length of the input. Finally, a pointer is selected by sampling from  $p_i$  via a sampling strategy (e.g., greedy, stochastic). In greedy sampling, for example, the selected pointer is the index of the element with the highest probability in  $p_i$ .

It is worth to note that in the original Ptr-Net proposal [42] both the encoder and the decoder use LSTMs. However, the authors of Ptr-Net have shown later (same year) that RNNs, including LSTM and GRU, are not suited for solving problems where the order of the input does not matter [59]. The main limitation of RNNs is that they are order-sensitive, an undesired behavior when working with sets of data. Moreover, these networks are ineffective in learning long distance relationships between the elements in the input, i.e., finding dependencies between elements that are located far apart in the input sequence [60]. Since RNNs pass the information sequentially, the longer the input sequence, the more likely it is that some information will be lost during sequence processing.

**Transformer architecture.** This architecture, introduced in 2017, presents a new way of computing relationships between the elements in the input. It completely abandons the use of RNNs and, instead, relies entirely on the attention mechanisms [50]. Dropping the recurrence removed the constraints of sequential computation, allowing larger models to be built and a faster training to be achieved. Current state-of-the-art NLP solutions (e.g., GPT-3 [61]) use the Transformer or one of its variants [62].

The core idea introduced by the Transformer is the self-attention mechanism that allows to find inter-dependencies between elements in the input, regardless of their positions in the input sequence. Broadly speaking, self-attention receives as input an embedding representation of  $n$  input elements, denoted by  $(x_1, \dots, x_n)$ , and maps them into  $(z_1, \dots, z_n)$  output embeddings while

preserving the input dimensions. Each  $z_i$  embedding,  $i \in (1, \dots, n)$ , contains information about how input element at position  $i$  is related to all the remaining input elements. This input–output transformation process can be summarized as follows. First, every input element  $x_i$ ,  $i \in (1, \dots, n)$ , is mapped via linear transformation into key  $k_i$ , query  $q_i$  and value  $v_i$  vectors. Next, a logit  $u_i^j$  measuring the compatibility between  $q_i$  and key  $k_j$ ,  $\forall j \in (1, \dots, n)$ , is computed:

$$u_i^j = \frac{q_i k_j^T}{\sqrt{d^{(k)}}}, j \in (1, \dots, n) \quad (28)$$

where  $d^{(k)}$  is the dimension of  $k$ .

Then, just as in Ptr-Net, the compatibility vector  $u_i$  is passed through a softmax layer (Expression (27)) to produce a probability vector  $p_i$ , whose size is equal to the length of the input. Finally, the output  $z_i$  is computed as:

$$z_i = \sum_{j=1}^n p_i^j v_j, \quad (29)$$

where  $p_i^j$  is the  $j$ th weight in  $p_i$ .

The generic input–output mapping, provided by the self-attention mechanism, allow these computation blocks to be stacked to create a higher level of abstraction and, potentially, give the network more generalization capacity. In the original work, authors stacked six self-attention blocks in the encoder and decoder and, at that time, were able to achieve state-of-the-art results in NLP tasks.

Please note that the Transformer was originally developed to solve NLP problems, where the order of the input sequence matters. To that end, the authors introduced a so called *positional encoding*.<sup>2</sup> Here in our work the order does not matter and, for this reason, the positional encoding is not included in our design.

## 4.2. MDP formulation

In the following sections the RD problem, outlined in Definition 1, is formulated as an MDP and described in terms of state, actions, and possible reward functions.

### 4.2.1. State space

The state of the RD problem includes: (i) a set of Rules that arrive at the reverse proxy and need to be distributed across nodes, denoted by  $\mathcal{R}' \subseteq \mathcal{R}$ ; (ii) a set of available nodes capable of hosting Rules in  $\mathcal{R}'$ , denoted by  $\mathcal{N}' \subseteq \mathcal{N}$ . Each Rule is represented by its profile that includes CPU, RAM and storage requirements, while each node is represented by its CPU, RAM and storage resources. Note that the node set, i.e., locations where Rules can be placed, is also part of the action space, which will be explained in the following section.

For an adequate distribution of Rules, reverse proxy must have an up-to-date information about the number of available nodes and their states. In distributed systems, nodes usually share this kind of information either by using passive or active methods. In case of passive methods, nodes expose a specific endpoint, usually `/health`, that responds with its status: an HTTP 200 OK status code informs that the device is alive, which may be followed by information on available resources. Then, the reverse proxy (e.g., Traefik) makes periodic requests to these endpoints and uses the obtained information during the distribution process. In case of active methods, as in MoleculerJS [12], nodes periodically broadcast Heartbeat packets containing information about their health status [63]. Here we assume that a reverse proxy always has an up-to-date information on the available nodes and their CPU, RAM and storage resources. Note that there might be situations where nodes do not have enough resources to serve all requests and, therefore, some of the requests will be rejected. For this reason, we introduce a dummy node  $node_{rej}$  where rejected Rules are placed. This node does not have the ability to process the Rules, as it does not have any computational resources, and its purpose is to allow the RL agent to associate rejected Rules with a dedicated place. The overall state for the RD problem is the following:

$$S \triangleq \{\mathcal{N}' \cup node_{rej}, \mathcal{R}'\}. \quad (30)$$

Designing  $S$  this way allows the RL agent to have a global view of the problem, in order to make appropriate placement decisions. Furthermore, this design can be applied to model other optimization problems (e.g., multiple knapsack problem, multiple vehicle routing problem) whose state is described by two sets and the problem solution can be modeled as a sequence of assignment decisions.

<sup>2</sup> See “Section 3.5 - Positional Encoding” in [50]

#### 4.2.2. Action space

It is assumed that a Rule  $x \in \mathcal{N}'$  can be placed at any available node  $n \in \mathcal{N}'$ , as long as resources are not exceeded. If a Rule has requirements higher than the resources available at the nodes in  $\mathcal{N}'$  then it is rejected, i.e. placed at the  $node_{rej}$ . Therefore, the action space is represented as follows:

$$\mathcal{A} = \mathcal{N}' \cup node_{rej} \quad (31)$$

Placing a Rule at node  $n$  involves updating the amount of available resources at  $n$ . This is done by subtracting Rule resource requirements from node's available resources. Placing a Rule at  $node_{rej}$  does not involve any computation, as the Rule is being rejected.

From the user point-of-view, placing a Rule at any node in  $\mathcal{N}'$  would result in a HTTP 201 Created meaning that the request was accepted and the Rule is running. On the other hand, placing a Rule at the  $node_{rej}$  would result in a HTTP 503 Service Unavailable response, meaning that the system is overloaded at the moment, and cannot process the incoming request. Therefore, the state and action spaces design is consistent with the realistic settings of Rule placement.

It is important to note that this work considers only the presence of edge devices. However, cloud can easily be integrated into the work by considering the  $node_{rej}$  as a cloud node, with virtually unlimited computational resources. This way, a placement at  $node_{rej}$  would mean that the Rule will be executed at the cloud.

#### 4.2.3. Rewards

The agent places Rules, one at a time, and in return receives a reward value. This reward depends on the RD problem variant under consideration (see Section 3).

*Greedy reward.* The reward signal for greedy optimization is simple and represented as follows:

$$r(s_t, a_t) = \begin{cases} 0, & \text{if Rule is rejected} \\ 1, & \text{if Rule is accepted} \end{cases} \quad (32)$$

Placing a Rule at any node in  $\mathcal{N}'$  gives a positive reward to the agent, while placing it at  $node_{rej}$ , i.e., rejecting the Rule, gives zero reward. Since the agent aims to maximize the accumulated reward, it will try to come up with a distribution strategy that maximizes the number of placed Rules.

*Critical-aware greedy reward.* To maximize the placement of the Rules, while ensuring a fair distribution, the following reward is used:

$$r(s_t, a_t) = \begin{cases} -2, & \text{if Rule is rejected} \\ \Omega^{MAX}, & \text{if Rule is accepted} \end{cases} \quad (33)$$

For each rejected Rule the agent is penalized by receiving a negative reward (-2, which has been selected empirically). On the other hand, placing a Rule at node  $n \in \mathcal{N}'$  gives a positive reward equal to the most critical resource among all nodes in  $\mathcal{N}'$ , as described in Expression (7).

Penalizing rejections with negative rewards and rewarding successful Rule placement with positive rewards has two effects. It encourages the placement of the largest number of Rules and, at the same time, it allows the agent to obtain high  $\Omega^{MAX}$  rewards, which brings fairness to the distribution of Rules.

*Cost-aware greedy reward.* The reward for maximizing the Rule placement while minimizing the number of used nodes is represented as follows:

$$r(s_t, a_t) = \begin{cases} -2, & \text{if Rule is rejected} \\ -1, & \text{if Rule placed at an empty node} \\ 0, & \text{otherwise} \end{cases} \quad (34)$$

The minimization of rejected requests is ensured by giving a negative reward, -2, for each rejected Rule. To enforce that a minimum number of nodes is used, the agent receives a negative reward, -1, every time it places a Rule at an empty node, i.e., a node without previously assigned Rules. Placing at a node that already has Rules assigned to it results in a reward equal to zero. This way the agent will try to find a strategy that uses the lowest number of nodes to place all the Rules.

#### 4.3. Model architecture

Fig. 3 is a graphical representation of the proposed model, which has two main components (encoder and decoder) that are detailed in this section. The source code of the model, implemented in Tensorflow [64], is also publicly available.<sup>3</sup>

<sup>3</sup> <https://github.com/AndreMaz/transformer-pointer-critic>.

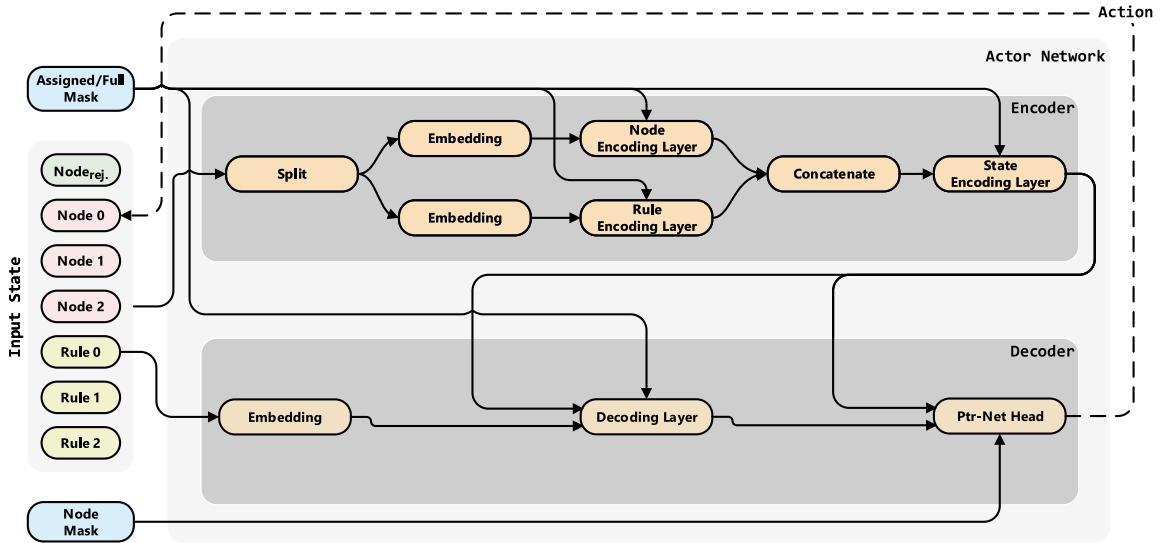


Fig. 3. Model architecture and decoding step example: Rule 0 placed at node 0.

#### 4.3.1. Encoder

The proposed encoder, illustrated in the upper part of Fig. 3, uses self-attention mechanisms in all of its encoding layers. It receives  $y = (y_1, y_2, \dots, y_{n-m}, y_{n-m+1}, \dots, y_n)$  as input, where  $y_1$  is  $node_{rej}$ ;  $y_2$  to  $y_{n-m}$  are the available nodes; and  $y_{n-m+1}$  to  $y_n$  are the Rules.

In the proposed architecture, the input  $y$  is split into two sets: Rules and nodes. Each set passes through a dedicated embedding layer and a self-attention block. The decision of having two separate processing components, one for each set, has two reasons. First, to give the model the ability to select different pre-processing strategies for each type of input. A pre-processing can be, for instance, sorting the rules and nodes using different strategies (as in traditional heuristic algorithms). However, in this case, the pre-processing strategies are not based on any human preconception, and the model learns adequate strategies that suit the QoS factor at hand. The second reason is to make the model flexible to input data of uneven number of features. In other words, to make the model generic and applicable to different problems. For example, for the cost-aware greedy optimization the node requires an additional binary feature in order to represent whether it is empty or not, meaning that the node is described by four features while the Rule is described by three only. Furthermore, the splitting idea can also be applied to more than two input components, which adds more flexibility to the proposed design.

After completing the node and Rule pre-processing, the output is concatenated and passed into a final self-attention block where the network performs further processing. This time it looks for the relationships between the nodes and the Rules.

An interesting property of the proposed encoder is that it generalizes much better than the typical Transformer encoder, especially when the number of Rules or nodes increases. This is mainly due to the flexibility of processing each input type separately using a dedicated location in the encoder. Finally, to improve the training process and enforce problem constraints, the Rules already in place and full nodes are masked at each step. Masking allows the encoder to discard specific elements during the processing stage. The masking is done by modifying the logits calculation in Expression (28) by:

$$u_i^j = \begin{cases} \frac{q_i k_j^T}{\sqrt{d^{(k)}}}, & j \in (1, \dots, n-m), j \neq 1, j \neq \text{empty node} \\ -\infty, & \text{otherwise} \end{cases} \quad (35)$$

The rule and the state encoding blocks use exactly the same expression but operate over different indices. In the case of rule encoding block the Rules that are already assigned are masked.

#### 4.3.2. Decoder

The proposed decoder is illustrated in the lower part of Fig. 3. At each step  $i$ , it receives encoder's output and a single Rule from range  $(y_{n-m+1}, \dots, y_n)$  and generates a pointer to a node  $(y_1, \dots, y_{n-m})$ .

In the decoding layer, the model looks for a (good) contender node for a given Rule. Since the decoder has access to encoder's output, where the model had a global view of the problem, the selection of the contender takes also into consideration the presence of other Rules and the state of the nodes. Finally, the last decoder layer is a Ptr-Net head, described in Section 4.1.2. This head

computes the location where the given Rule will be placed. In order to avoid selecting unfeasible positions (e.g., full nodes), the logits computation of the Ptr-Net head in Expression (26) is modified as:

$$u_i^j = \begin{cases} v^T \tanh(W_1 e_j + W_2 d_i), & j \in (1, \dots, n-m, \dots, n), \\ & j \leq n-m, j \neq 1, \\ & j \neq \text{empty node} \\ -\infty, & \text{otherwise} \end{cases} \quad (36)$$

Furthermore, as in [23], the logits from Expression (36) are clipped to  $[-C_{\text{logit}}, C_{\text{logit}}]$ , where  $C_{\text{logit}} = 10$  in this work, to encourage the exploration:

$$u_i^{(\text{clip})} = C_{\text{logit}} \cdot \tanh(u_i) \quad (37)$$

Then, the clipped logits  $u_i^{(\text{clip})}$  passes through a softmax layer, Expression (27), to produce the probability vector from which the node is selected. In the following decoding step, the resources of the selected node are updated, the assigned Rule is masked and the process repeats for the next Rule in  $(y_{n-m+1}, \dots, y_n)$ .

It is important to outline that, in addition to sequential Rule feeding, we have tested random and neural-network based feeding strategies. However, other than adding additional complexity to the model we noticed no improvements in using any of these strategies. Since the model has a global view of the problem, via the encoder's output, the order by which the Rules are placed has no effect in the final result.

## 5. Performance evaluation

### 5.1. Experimental setup

To evaluate the performance of the proposed agent, a comparison will be made against several greedy heuristics and the optimal solution, generated using CPLEX solver. The optimal relies on the mathematical formalizations presented in Section 3. The main goals are: (i) to assess the performance of the agent for different objective function criteria, and compare against baseline methods; (ii) to evaluate the scalability and generalization capacity of the agent, i.e., evaluate the performance of the agent when it has to solve problems larger than the ones used during training. It worth to mention that performance evaluation does not consider other DRL approaches because, as mentioned in Section 2, models used to solve task distribution and load balancing problems do not have the ability to scale, i.e., models have to be retrained each time the state or action space changes. Therefore, to validate the proposed method, optimal solutions are used as a baseline.

All the development and testing were done on a PC with i7 6700 3.4 GHz CPU with 32 GB of RAM and a single Nvidia 2080Ti with 11 GB GPU.

The simulation environment was configured to generate instances on-the-fly. A uniform distribution, in range  $[0, 1]$ , is used to assign CPU, RAM, and storage resources to all nodes in an instance. A pool of 1000 Rules is made available, for set sampling purposes, and resource requirement values are assigned using a uniform distribution in range  $[0.01, 0.30]$ . This range ensures that the placement of Rules is neither too easy nor artificially hard, while providing some degree of freedom when distributing Rules. Additionally, the environment was configured to produce problem instances of different sizes using the following parameters, allowing the assessment of the agent's performance and scalability:

- *Node-related parameters:*
  - Smallest number of nodes: 10
  - Largest number of nodes: 50
  - Step size: 10
- *Rule-related parameters:*
  - Smallest number of Rules: 10
  - Largest number of Rules: 100
  - Step size: 10

The training process of the agent and baseline heuristics are discussed next, prior to the analysis of results. It should be noted that the choice for relatively simple greedy heuristics, rather than complex heuristics or meta-heuristics, is due to the fact that it is not feasible to run complex approaches in real-time on reverse proxies. As mentioned in Section 1.1, the load balancing methods used in actual implementations are often simple strategies. All strategies, RL agent and baseline heuristics, build a solution in a single pass, by starting from an empty solution and expanding it by placing one Rule at a time.

**Table 1**  
Network(s) and training parameters.

General	
Training steps	100 000
Batch size	128
Discount factor ( $\gamma$ )	0.99
Entropy coefficient ( $c_{\text{entropy}}$ )	0.01
Actor network	
Weight initialization	Xavier uniform [65]
Number of layer stack ( $N_{\text{actor}}^{\text{stack}}$ )	1
Embedding size	128
Number of heads	8
Inner layer dimension	128
Logit clipping ( $C_{\text{logit}}$ )	10
Gradient clipping	L2 Norm (1.0)
Optimizer	Adam [66]
Learning rate ( $\alpha_{\text{actor}}$ )	$1e^{-4}$
Critic network	
Weight initialization	Xavier uniform [65]
Number of layer Stacks ( $N_{\text{critic}}^{\text{stack}}$ )	3
Embedding size	128
Number of heads	8
Inner layer dimension	512
Gradient clipping	L2 Norm (1.0)
Last layers dimensions	128
Optimizer	Adam [66]
Learning rate ( $\alpha_{\text{critic}}$ )	$5e^{-4}$

## 5.2. Training and hyperparameters

The agent was trained on a fixed problem size of 10 nodes and 20 Rules, an empirically selected size that showed to be a good compromise between the training time, approximately 14 h, and the agent's ability to generate robust solutions to all the problem instances during the testing.

The agent is trained using an A2C algorithm [57], summarized in Algorithm 1, with mini-batches of size equal to 128, for better gradient estimates. Two dedicated networks are used: (i) the actor for learning the policy, denoted by  $\pi_\theta$ ; (ii) the critic to estimate the baseline, denoted by  $V_\phi$ . The actor network is illustrated in Fig. 3 and described in detail in Section 4.3. The critic network is the encoder, from Fig. 3, followed by three fully connected layers with linear activation. The last layer of the critic network produces a single scalar value that represents the baseline estimation. During the testing phase, only the actor network is used.

The choice for two separate networks, with no parameter sharing, was made in order to have more freedom to control agent's inference time and the quality of critic's baseline estimations. The actor's encoder uses a single self-attention stack  $N_{\text{actor}}^{\text{stack}} = 1$  in each encoding block, while the critic uses a stack of three  $N_{\text{critic}}^{\text{stack}} = 3$ . Critic with less than three self-attention stacks produced poor baseline estimations. Different learning rates were also used for the actor  $\alpha_{\text{actor}} = 1e^{-4}$  and the critic  $\alpha_{\text{critic}} = 5e^{-4}$  networks, parameters that have shown to be stable and to ensure convergence. Table 1 summarizes all the parameters that were adopted in this work. Please note that the batch size and number of layer stacks in critic network were selected as a trade-off between the quality of the solution and the GPU memory constraints.

---

### Algorithm 1: Actor-Critic training algorithm (adapted from [23]).

---

**Input** :  $T$ =training steps,  $B$ =batch size

**Output**: Trained actor network  $\pi_\theta$

Initialize actor  $\pi_\theta$  network parameters;

Initialize critic  $V_\phi$  network parameters;

**for**  $t = 1$  **to**  $T$  **do**

```

     $s_i \leftarrow \text{SampleProblem}(), \forall i \in \{1, \dots, B\}$ 
     $a_i, r_i \leftarrow \text{SolveProblem}(s_i, \pi_\theta), \forall i \in \{1, \dots, B\}$ 
     $b_i \leftarrow \text{EstimateBaseline}(s_i, V_\phi), \forall i \in \{1, \dots, B\}$ 
     $A \leftarrow \text{ComputeAdvantage}(r_i, b_i)$  // Expression (22)
     $E \leftarrow \text{ComputeEntropy}(\pi_\theta)$  // Expression (24)
     $g_\theta \leftarrow \frac{1}{B} \sum_{i=1}^B A \nabla_\theta \log(\pi_\theta(a_i | s_i)) + c_{\text{entropy}} \times E$ 
     $L_\phi \leftarrow \frac{1}{B} \sum_{i=1}^B \|A\|_2^2$ 
     $\theta \leftarrow \text{ADAM}(\theta, g_\theta)$ 
     $\phi \leftarrow \text{ADAM}(\phi, \nabla_\phi L_\phi)$ 

```

**end**

---

### 5.3. Baseline heuristics

#### 5.3.1. Random insertion

This heuristic randomly distributes the set of Rules across a set of available nodes. More specifically, a Rule and a node are randomly picked from corresponding sets and then the feasibility of the pair is checked, i.e., a validation is performed to check if the resources available at the node are enough to satisfy the requirements of the Rule. If this condition is not met, another node is picked. This process is repeated until all Rules are in place or marked as rejected.

#### 5.3.2. Critical resource insertion

This is a two step heuristic, summarized in Algorithm 2, that first sorts the Rules by their resource requirements and then distributes them across the nodes, possibly giving priority to fairness ( $\Omega^n$ ). The following four variations were generated:

- Descending Rules-Descending Critical (DR-DC)
- Descending Rules-Ascending Critical (DR-AC)
- Ascending Rules-Descending Critical (AR-DC)
- Ascending Rules-Ascending Critical (AR-AC)

DR-DC and AR-DC ( $C_{\text{order}} = \text{'desc'}$ ) prioritize the placement of Rules at locations that maximize the availability at the most critical resource, ensuring fairness. DR-AC and AR-AC ( $C_{\text{order}} = \text{'asc'}$ ), on the other hand, prioritize placement of Rules that minimizes the  $\Omega^n$ , which potentially leads to fewer nodes in use. The  $X_{\text{order}}$  variable controls whether smaller or larger Rules are placed first.

**Algorithm 2:** Critical resource insertion heuristic.

---

**Input :**  $\mathcal{N}'$ =node set,  $\mathcal{X}'$ =rule set,  $X_{\text{order}} \in \{\text{'asc'}, \text{'desc'}\}$  is the rule size sorting criteria,  $C_{\text{order}} \in \{\text{'asc'}, \text{'desc'}\}$  is the critical resource sorting criteria.

**Output:** Feasible solution

$\mathcal{X}' = \text{SortByLargestResource}(\mathcal{X}', X_{\text{order}})$

```

foreach  $x$  in  $\mathcal{X}'$  do
    Initialize critical resource list  $L_{\Omega}$ 
    foreach  $n$  in  $\mathcal{N}'$  do
        // Determine critical resource
         $\Omega^n = \text{ComputeCritical}(n, x)$ 
         $L_{\Omega} \leftarrow (\Omega^n, n)$ 
    end
    // Sort  $L_{\Omega}$  by  $\Omega^n$ 
     $L_{\Omega} = \text{Sort}(L_{\Omega}, C_{\text{order}})$ 
    // Do the first fit
    foreach  $\Omega^n, n$  in  $L_{\Omega}$  do
         $allocated = \text{False}$ 
        if  $\Omega^n \geq 0$  then
            // Place Rule  $x$  at node  $n$ 
             $allocated = \text{True}$ 
            break // Break the loop
        end
        if  $allocated == \text{False}$  then
            Reject  $x$ 
    end
end

```

---

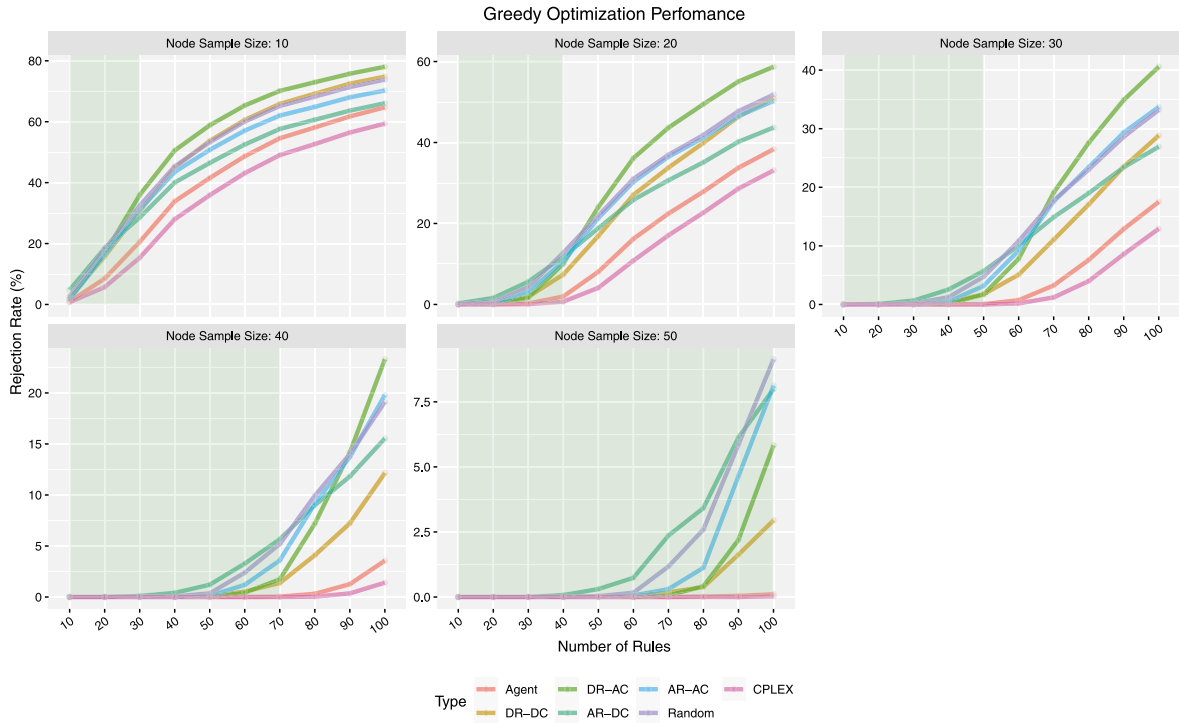
### 5.4. Evaluation

Evaluation consists in solving three different RD variants, mathematically expressed in Section 3.3, and observe how the agent performs in the key performance indicators (KPIs) of each. To ensure the consistency of the agent performance, three different seeds were used to initialize the agent training weights, which essentially results in three distinct versions of the same agent. After the training is finished, each agent solves 100 problem instances for each scenario, generated according to the strategy described in Section 5.1. Therefore, the reported performance is an average of the three agents, i.e., an average of 300 executions. The baseline heuristics, which are deterministic methods, also solve 300 problem instances. Finally, to ensure that results of the CPLEX solver are obtained within a feasible time, a time limit of 60 s is set. With time limit set, CPLEX solver will either return the optimal solution, if it manages to find one, or the best solution found so far. For this configuration CPLEX takes approximately 8 days to solve all the problem instances of each RD variant under the consideration.

#### 5.4.1. Greedy results

The KPI for the greedy optimizer (see Section 3.3.1) is the Rule rejection rate, i.e., the likelihood of a user's request being rejected. Fig. 4 shows the performance of the different solving methods for this particular case. Looking at the plots, it is visible that CPLEX solver has the lowest rejection rate in the all scenarios and, therefore, has the best performance. Moreover, in the green highlighted areas CPLEX was able to find optimal solutions within the time limit.





**Fig. 4.** Rule rejection rate for different problem sizes. Green areas highlight the problem sizes where CPLEX was able to find optimal results for all instances, i.e., all the solution space was searched. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

**Table 2**

Average rejection rate ( $\bar{X}(\text{Rej.})$ ) gap and standard deviation ( $s_X(\text{Rej.})$ ) with respect to the CPLEX solver for the greedy optimization.

Node sample size	Agent		DR-DC		DR-AC		AR-DC		AR-AC		Random	
	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$
10	4.69	1.80	14.40	5.05	17.97	6.89	9.26	2.94	11.95	4.27	14.44	4.66
20	3.18	2.48	10.73	7.70	16.18	11.98	9.64	5.31	12.28	8.23	13.20	8.34
30	1.51	1.95	6.08	6.67	10.49	11.99	7.64	6.51	9.01	9.36	9.27	8.99
40	0.34	0.70	2.36	3.73	4.50	7.61	4.52	5.29	4.58	6.71	4.93	6.56
50	0.01	0.03	0.52	0.98	0.85	1.87	2.10	2.88	1.42	2.75	1.89	3.15

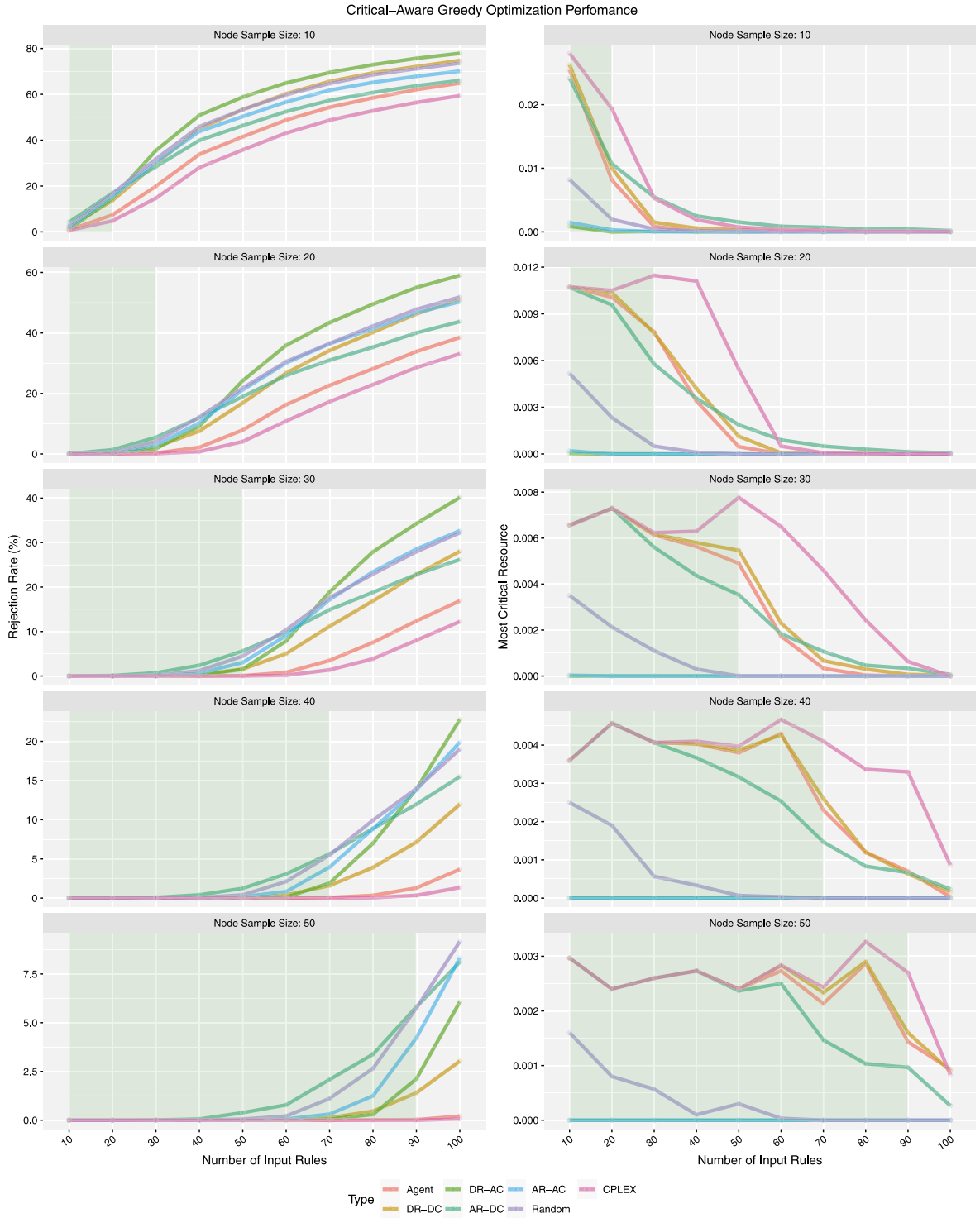
The second best performing method is the proposed agent. It is visible that regardless of the problem size, the proposed agent has the smallest rejection gap with respect to the CPLEX solver, which acts as a lower bound. As an example, for problems with 10 nodes the proposed agent has an average rejection rate 4.69% higher than the CPLEX, as shown in Table 2. As the number of available nodes increases the rejection gap becomes smaller, reaching an average of 0.01% for problems with 50 nodes. In contrast, the best performing heuristic (AR-DC) has an average rejection gap of 9.26% for 10 nodes, when compared to CPLEX. The DR-DC presents the lowest gap, 0.52%, for 50 nodes, which is still larger than the agent's gap.

Results indicate that agent's generalization capacity is maintained even for large problems. Note that during training the agent solved problems with only 10 nodes and 20 Rules. However, results do not show any critical degradation in performance, even for problem instances five times larger than the ones used in training. For example, for the problems with 50 nodes, where CPLEX solves all problem instances to optimality, the agent generates near-optimal solutions.

#### 5.4.2. Critical-aware greedy results

In this scenario, Section 3.3.2, the Rule rejection rate and the most critical resource ( $\Omega^{\text{MAX}}$ ) are the KPIs, which are shown in Fig. 5.

As in the case of greedy optimization, the agent shows a similar trend when considering the rejection rate. On the other hand, looking at the most critical resource at different ranges, the agent struggles to keep up with the CPLEX solver and seem to stay closer the DR-DC heuristic. However, when taking both KPIs into account, it is visible that DR-DC's higher critical resource value is achieved at the expense of a higher rejection rate. Similar behavior happens with the AR-DC heuristic. Considering the KPI gaps



**Fig. 5.** Rule rejection rate (left column) and most critical resource (right column) for different problem sizes. Green areas highlight the problem sizes where CPLEX was able to find optimal results for all instances, i.e., all the solution space was searched. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

between CPLEX and other methods, summarized in Table 3, it is visible that in general the overall  $\Omega^{\text{MAX}}$  differences are negligible but due to lower rejection rate the proposed agent offers a smoother user experience.

**Table 3**

Average  $\Omega^{\text{MAX}}$  ( $\bar{X}(\Omega^{\text{MAX}})$ ) and rejection rate ( $\bar{X}(\text{Rej.})$ ) gaps, alongside with the standard deviation ( $s_X(\Omega^{\text{MAX}})$  and  $s_X(\text{Rej.})$ ), with respect to the CPLEX solver for the critical-aware optimization.

Node sample size	Agent				DR-DC				DR-AC			
	$\bar{X}(\Omega^{\text{MAX}})$	$s_X(\Omega^{\text{MAX}})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\Omega^{\text{MAX}})$	$s_X(\Omega^{\text{MAX}})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\Omega^{\text{MAX}})$	$s_X(\Omega^{\text{MAX}})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$
10	0.00210	0.00352	4.75	1.86	0.00175	0.00292	14.14	5.10	0.00553	0.00966	17.73	7.13
20	0.00173	0.00274	3.21	2.49	0.00155	0.00250	10.77	7.64	0.00498	0.00539	16.04	11.99
30	0.00157	0.00186	1.55	1.98	0.00137	0.00166	6.01	6.65	0.00483	0.00282	10.51	12.07
40	0.00080	0.00101	0.36	0.75	0.00076	0.00100	2.33	3.68	0.00366	0.00108	4.39	7.47
50	0.00020	0.00041	0.02	0.04	0.00015	0.00035	0.50	0.97	0.00252	0.00065	0.85	1.93

Node sample size	AR-DC				AR-AC				Random			
	$\bar{X}(\Omega^{\text{MAX}})$	$s_X(\Omega^{\text{MAX}})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\Omega^{\text{MAX}})$	$s_X(\Omega^{\text{MAX}})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\Omega^{\text{MAX}})$	$s_X(\Omega^{\text{MAX}})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$
10	0.00093	0.00304	9.18	3.01	0.00544	0.00946	11.91	4.13	0.00453	0.00759	14.38	4.69
20	0.00164	0.00291	9.62	5.43	0.00496	0.00537	12.18	8.22	0.00417	0.00465	12.99	8.32
30	0.00172	0.00185	7.57	6.48	0.00483	0.00281	8.89	9.25	0.00413	0.00254	9.12	8.92
40	0.00118	0.00116	4.50	5.30	0.00366	0.00108	4.57	6.74	0.00312	0.00124	4.92	6.56
50	0.00059	0.00081	2.06	2.84	0.00252	0.00065	1.41	2.74	0.00218	0.00074	1.89	3.13

**Table 4**

Average empty node ( $\bar{X}(\text{Empty})$ ) and rejection gaps ( $\bar{X}(\text{Rej.})$ ), alongside with the standard deviation ( $s_X(\text{Empty})$  and  $s_X(\text{Rej.})$ ), with respect to the CPLEX solver for the cost-aware optimization.

Node sample size	Agent				DR-DC				DR-AC			
	$\bar{X}(\text{Empty})$	$s_X(\text{Empty})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\text{Empty})$	$s_X(\text{Empty})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\text{Empty})$	$s_X(\text{Empty})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$
10	-0.03	0.12	4.90	1.52	0.41	0.66	14.19	5.03	0.76	1.10	17.84	6.99
20	0.03	0.78	3.63	2.55	1.79	2.22	10.70	7.61	3.39	3.40	16.05	11.90
30	0.45	1.23	2.06	2.35	4.31	3.36	6.04	6.56	7.86	5.03	10.43	11.92
40	1.23	0.79	0.67	1.19	7.78	2.83	2.32	3.65	13.85	4.79	4.40	7.50
50	1.45	0.66	0.07	0.16	10.67	2.63	0.48	0.91	19.28	5.30	0.85	1.89

Node sample size	AR-DC				AR-AC				Random			
	$\bar{X}(\text{Empty})$	$s_X(\text{Empty})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\text{Empty})$	$s_X(\text{Empty})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$	$\bar{X}(\text{Empty})$	$s_X(\text{Empty})$	$\bar{X}(\text{Rej.})$	$s_X(\text{Rej.})$
10	-0.04	0.56	9.09	2.94	0.80	1.18	11.82	4.12	0.55	0.92	14.28	4.61
20	0.71	2.22	9.61	5.24	3.62	3.67	12.24	8.13	2.66	3.11	13.06	8.28
30	2.72	3.62	7.58	6.38	8.38	5.55	8.97	9.17	6.41	4.77	9.14	8.77
40	6.03	3.11	4.54	5.25	14.80	5.51	4.53	6.72	11.65	4.43	4.91	6.54
50	9.00	2.30	2.05	2.82	20.75	6.06	1.38	2.67	16.44	4.43	1.87	3.08

#### 5.4.3. Cost-aware greedy results

For this RD optimization criteria, described in Section 3.3.3, the Rule rejection rate and the number of empty nodes are the KPIs, which are shown in Fig. 6.

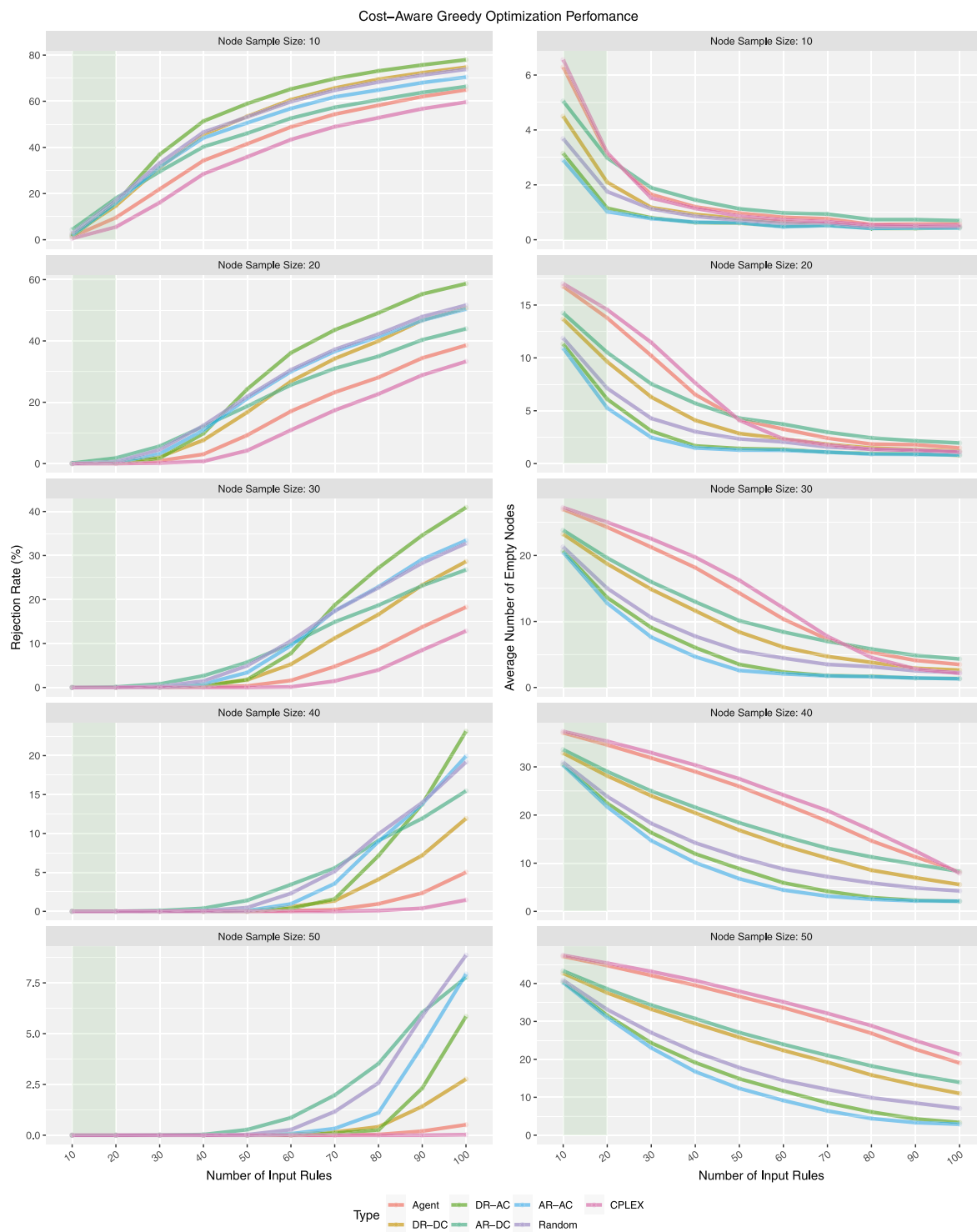
From all the RD problems under consideration, this one was the hardest for CPLEX. This has to do with a greater number of binary variables involved in the mathematical formalization of the problem. The solver did not manage to obtain optimal solutions for problems with more than 20 Rules to distribute. The problem complexity might explain, therefore, the larger agent-CPLEX gap in the rejection rate, when compared with other optimization criteria Fig. 5. These results are summarized in Fig. 5. Nevertheless, the agent shows a trend similar to CPLEX solver in both KPIs and does not show critical performance degradation, as it happens with the heuristics.

When looking at gaps in Table 4, it is possible to see that for 10 nodes the agent uses in average less -0.03 nodes but has a rejection rate is 4.9% higher than the CPLEX. As the number of nodes increases the rejection rate gap decreases at the expense of additional nodes in use. For example, for 50 nodes the agent requires more 1.45 nodes to have almost the same rejection rate as the CPLEX.

In addition, when looking at the results of the heuristics, it is visible that they struggle to keep both KPIs gaps close to the CPLEX and the proposed agent. In fact, from the three scenarios being considered, the cost-aware is where the heuristics struggle the most. This highlights that a change of a single KPI requires redesigning and developing a new heuristic, which is impractical and time consuming. In the case of the proposed agent, which performs well in all scenarios, the only change would be the reward signal.

#### 5.4.4. Inference time

Reporting the time to obtain the solutions is important but hard to measure. It depends on the hardware used (e.g., CPU vs GPU, GPU model), programming language, code implementation/optimization and so on. Nevertheless, a practical approach was taken and the reported times, summarized in Table 5, are averages of 1000 executions obtained on the development machine. For brevity purposes, only the results for 100 Rules are shown, i.e., the upper bound. Also, note that the reported measurements are rough approximations that can vary from execution to execution. Still, they show the trend and the orders of magnitude that each



**Fig. 6.** Rule rejection rate (left column) and empty nodes (right column) for different problem sizes. Green areas highlight where the problem sizes CPLEX was able to find optimal results for all instances, i.e., all the solution space was searched. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

**Table 5**

Time, in milliseconds, to obtain the solution for problems with 100 Rules. Reported times are an average of 1000 executions.

Node sample size	Agent	DC-DC	DC-AC	AR-DC	AR-AC	Random
10	394.82	14.49	14.14	14.22	14.35	12.27
20	398.39	28.53	28.46	28.13	28.60	18.59
30	402.89	42.28	42.61	41.46	42.45	20.05
40	403.31	56.79	57.77	55.80	57.43	20.37
50	404.17	69.86	72.24	69.42	71.80	21.14

method takes to compute the solution. The running times of CPLEX are not reported because, as mentioned earlier, it is not feasible to run it at reverse proxies. Finally, agent's reported times are the average duration of an entire episode, which also accounts the environment's transitions.

Looking at the agent's results it is visible that its time is practically constant, regardless of the number of nodes in the input. This is a result of: (i) using the Transformer architecture that, as mentioned in , does not use RNNs to compute the relationships between the input elements; (ii) GPU parallelization capabilities, allowing to compute the relationships between all the elements in a single pass. It is also worth to mention that, regardless of the tested input size, the model takes approximately 2.5 ms in every decoding step, i.e., to find a location for a single Rule.

Comparing the agent's times with the heuristics, it is visible that they operate at different orders of magnitude. For example, for 10 nodes all the heuristics are around 28 times faster than the agent. However, for 50 nodes, this difference drops down to 6 for the critical resource heuristics and 20 for the random heuristic. This time difference reduction is explained by the complexity of the heuristics:  $O(r \cdot n \log(n))$  for the critical resource heuristic and  $O(r \cdot n)$  for the random heuristics, where  $r$  is the number of Rules and the  $n$  is the number of nodes. Nevertheless, despite the reported time differences, the agent is able to generate high quality or even optimal solutions in 400 ms, which makes the proposed model an appealing option.

Also, it should be noted that, although it is not visible in Table 5, the self-attention mechanism in the Transformer has  $O(m^2)$  complexity, where  $m$  is total number of input elements [62]. The reported constant time for the tested problem instances is ensured by the GPU parallelization. Once the problem size become large enough, and hardware limitations of the GPU are reached, the effects of quadratic complexity become visible in the rapidly growing inference time. However, this limitation can be tackled by replacing the Transformer with a recently introduced variation called the *Linformer* [67], which reduces the self-attention complexity from  $O(m^2)$  to  $O(m)$  while maintaining similar performance in NLP tasks. Overall, the performance and the inference speed of the proposed model can be improved even further by tuning the hyperparameters, using more advanced RL training algorithms (e.g., rolling baseline [49], proximal policy [68]), using more recent variations of the Transformer [62,69] and by doing model optimization<sup>4</sup> and compression [70].

## 6. Conclusions

This article investigated the use of DNN-based strategies for the distribution of event processing tasks across edge devices, considering three different optimization criteria. Each criteria is mathematically formulated, ensuring proper task execution, fair usage of resource and/or cost minimization related to device operation. Furthermore, an MDP formulation is introduced, which is followed by the presentation of the model based on Transformer and Ptr-Net architectures.

Simulation experiments show that the proposed model is capable of producing high quality solutions in all scenarios under consideration. Additionally, it is scalable and capable of solving problems with up to five times larger than the ones that it was trained upon. The latter is a very important ability as it allows to solve instances of different sizes without the need of retraining, which makes it easy to deploy in real world. Existing work applying DNN and RL methods to load balancing tend to neglect model scalability and only work with static environment settings, which is not applicable in real world.

In summary, the proposed model is generic and can be applied to problems that are modeled using two sets of elements and solved by taking sequential assignments decisions. The latter is an important characteristic in load balancing problems as the increasing network complexity will likely generate several problem variations for which there is no good performing heuristic, making RL and machine learning methods attractive and of practical value.

Future research will focus on further tuning the performance of the proposed method while studying its applicability to other combinatorial problems that can be modeled in a similar fashion to the problem investigated in this work.

## Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Andriy Mazayev reports financial support was provided by Portuguese Foundation for Science and Technology.

<sup>4</sup> See [https://www.tensorflow.org/lite/performance/model\\_optimization](https://www.tensorflow.org/lite/performance/model_optimization).

## Acknowledgments

This work was supported by FCT (Foundation for Science and Technology) from Portugal within CEOT (Center for Electronic, Optoelectronic and Telecommunications), by UID/MULTI/00631/2020 project and by FCT (Foundation for Science and Technology), Portugal Ph.D Grant SFRH/BD/138836/2018.

## References

- [1] S. Duquennoy, G. Grimaud, J.-J. Vandewalle, The web of things: interconnecting devices with high usability and performance, in: 2009 International Conference on Embedded Software and Systems, IEEE, 2009, pp. 323–330.
- [2] D. Guinard, V. Trifa, T. Pham, O. Liechti, Towards physical mashups in the web of things, in: Proceedings of the 6th International Conference on Networked Sensing Systems, in: INSS'09, IEEE Press, Piscataway, NJ, USA, 2009, pp. 196–199.
- [3] Web of things (WoT) architecture, [Accessed 02-06-2020]. <https://w3c.github.io/wot-architecture/>.
- [4] OCF Core Specification 2.0.4, [Accessed 02-09-2019]. <https://openconnectivity.org/developer/specifications>.
- [5] J. Hong, Y.-G. Hong, J.-S. Youn, Problem Statement of IoT integrated with Edge Computing, draft-hong-iot-edge-computing-02, Internet Engineering Task Force, 2019, (in preparation).
- [6] J. Hong, Y.-G. Hong, X. de Foy, M. Kovatsch, E. Schooler, D. Kutscher, IoT Edge Challenges and Functions, draft-irtf-t2trg-iot-edge-06, Internet Engineering Task Force, 2022, (in preparation).
- [7] EVE - LF edge, [Accessed 01-03-2021]. <https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/>.
- [8] Edge X foundry, [Accessed 01-03-2021]. <https://www.edgexfoundry.org/>.
- [9] MobileEdgeX, [Accessed 01-03-2021]. <https://mobileedge.com/>.
- [10] NGINX | high performance load balancer, web server and reverse proxy, [Accessed 15-09-2020]. <https://www.nginx.com/>.
- [11] Traefik - open source load balancer and reverse proxy for microservices, [Accessed 15-09-2020]. <https://docs.traefik.io/>.
- [12] Moleculer - Progressive microservices framework for Node.js. <https://moleculer.services/>.
- [13] Using nginx as HTTP load balancer, [Accessed 15-09-2020]. [http://nginx.org/en/docs/http/load\\_balancing.html](http://nginx.org/en/docs/http/load_balancing.html).
- [14] Services - Traefik, [Accessed 15-09-2020]. <https://docs.traefik.io/routing/services/#load-balancing>.
- [15] Load Balancing | Moleculer - Progressive microservices framework for Node.js. <https://moleculer.services/docs/0.14/balancing.html#Built-in-strategies>.
- [16] K.-M. Cho, P.-W. Tsai, C.-W. Tsai, C.-S. Yang, A hybrid meta-heuristic algorithm for VM scheduling with load balancing in cloud computing, *Neural Comput. Appl.* 26 (6) (2015) 1297–1309.
- [17] Autoscaling in Kubernetes, [Accessed 15-09-2020]. <https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/>.
- [18] StarlingX, [Accessed 15-09-2020]. <https://www.starlingx.io/>.
- [19] M.V. Modenesi, A.G. Evsukoff, M.C. Costa, A load balancing knapsack algorithm for parallel fuzzy c-means cluster analysis, in: International Conference on High Performance Computing for Computational Science, Springer, 2008, pp. 269–279.
- [20] A. Trivella, D. Pisinger, Bin-packing problems with load balancing and stability constraints, in: INFORMS Transportation and Logistics Society 2017, 2017.
- [21] N. Feamster, J. Rexford, Why (and how) networks should run themselves, 2017, arXiv preprint arXiv:1710.11583.
- [22] M.S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, F. Hussain, Machine learning at the network edge: A survey, *ACM Comput. Surv.* 54 (8) (2021) 1–37.
- [23] I. Bello, H. Pham, Q.V. Le, M. Norouzi, S. Bengio, Neural combinatorial optimization with reinforcement learning, 2016, arXiv preprint arXiv:1611.09940.
- [24] V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, et al., Human-level control through deep reinforcement learning, *Nature* 518 (7540) (2015) 529–533.
- [25] D. Silver, A. Huang, C.J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al., Mastering the game of go with deep neural networks and tree search, *Nature* 529 (7587) (2016) 484–489.
- [26] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al., Mastering atari, go, chess and shogi by planning with a learned model, *Nature* 588 (7839) (2020) 604–609.
- [27] K. Arulkumaran, M.P. Deisenroth, M. Brundage, A.A. Bharath, Deep reinforcement learning: A brief survey, *IEEE Signal Process. Mag.* 34 (6) (2017) 26–38.
- [28] A. Shakarami, M. Ghobaei-Arani, M. Masdari, M. Hosseinzadeh, A survey on the computation offloading approaches in mobile edge/cloud computing environment: a stochastic-based perspective, *J. Grid Comput.* (2020) 1–33.
- [29] L. Lei, Y. Tan, K. Zheng, S. Liu, K. Zhang, X. Shen, Deep reinforcement learning for autonomous internet of things: Model, applications and challenges, *IEEE Commun. Surv. Tutor.* 22 (3) (2020) 1722–1760.
- [30] Y. Bengio, A. Lodi, A. Prouvost, Machine learning for combinatorial optimization: a methodological tour d'horizon, *European J. Oper. Res.* (2020).
- [31] N. Mazyavkina, S. Sviridov, S. Ivanov, E. Burnaev, Reinforcement learning for combinatorial optimization: A survey, 2020, arXiv preprint arXiv:2003.03600.
- [32] Y.-H. Chiang, T.-W. Chiang, T. Zhang, Y. Ji, Deep-dual-learning-based cotask processing in multiaccess edge computing systems, *IEEE Internet Things J.* 7 (10) (2020) 9383–9398.
- [33] Z. Zhang, C. Li, S. Peng, X. Pei, A new task offloading algorithm in edge computing, *EURASIP J. Wireless Commun. Networking* 2021 (1) (2021) 1–21.
- [34] Y. Xu, W. Xu, Z. Wang, J. Lin, S. Cui, Deep reinforcement learning based mobility load balancing under multiple behavior policies, in: ICC 2019-2019 IEEE International Conference on Communications, ICC, IEEE, 2019, pp. 1–6.
- [35] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, M. Riedmiller, Deterministic policy gradient algorithms, in: E.P. Xing, T. Jebara (Eds.), Proceedings of the 31st International Conference on Machine Learning, in: Proceedings of Machine Learning Research, 32, (1) PMLR, Beijing, China, 2014, pp. 387–395, <https://proceedings.mlr.press/v32/silver14.html>.
- [36] Z. Zhang, F.R. Yu, F. Fu, Q. Yan, Z. Wang, Joint offloading and resource allocation in mobile edge computing systems: an actor-critic approach, in: 2018 IEEE Global Communications Conference (GLOBECOM), IEEE Press, 2018, pp. 1–6, <http://dx.doi.org/10.1109/GLOCOM.2018.8647593>, <https://doi.org/10.1109/GLOCOM.2018.8647593>.
- [37] M.G.R. Alam, M.M. Hassan, M.Z. Uddin, A. Almogren, G. Fortino, Autonomic computation offloading in mobile edge for IoT applications, *Future Gener. Comput. Syst.* 90 (2019) 149–157.
- [38] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, M. Bennis, Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning, *IEEE Internet Things J.* 6 (3) (2018) 4005–4018.
- [39] K. Zhang, Y. Zhu, S. Leng, Y. He, S. Maharjan, Y. Zhang, Deep learning empowered task offloading for mobile edge computing in urban informatics, *IEEE Internet Things J.* 6 (5) (2019) 7635–7647.
- [40] T. Yang, Y. Hu, M.C. Gursoy, A. Schmeink, R. Mathar, Deep reinforcement learning based resource allocation in low latency edge computing networks, in: 2018 15th International Symposium on Wireless Communication Systems, ISWCS, IEEE, 2018, pp. 1–5.
- [41] D. Van Le, C.-K. Tham, A deep reinforcement learning based offloading scheme in ad-hoc mobile clouds, in: IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops, INFOCOM WKSHPS, IEEE, 2018, pp. 760–765.
- [42] O. Vinyals, M. Fortunato, N. Jaitly, Pointer networks, in: Advances in Neural Information Processing Systems, 2015, pp. 2692–2700.

- [43] I. Sutskever, O. Vinyals, Q.V. Le, Sequence to sequence learning with neural networks, in: *Advances in Neural Information Processing Systems*, 2014, pp. 3104–3112.
- [44] D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, 2014, arXiv preprint [arXiv:1409.0473](https://arxiv.org/abs/1409.0473).
- [45] M.-T. Luong, H. Pham, C.D. Manning, Effective approaches to attention-based neural machine translation, 2015, arXiv preprint [arXiv:1508.04025](https://arxiv.org/abs/1508.04025).
- [46] S. Hochreiter, J. Schmidhuber, Long short-term memory, *Neural Comput.* 9 (8) (1997) 1735–1780.
- [47] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio, Learning phrase representations using RNN encoder-decoder for statistical machine translation, 2014, arXiv preprint [arXiv:1406.1078](https://arxiv.org/abs/1406.1078).
- [48] M. Nazari, A. Oroojlooy, L. Snyder, M. Takác, Reinforcement learning for solving the vehicle routing problem, in: *Advances in Neural Information Processing Systems*, 2018, pp. 9839–9849.
- [49] W. Kool, H. Van Hoof, M. Welling, Attention, learn to solve routing problems!, 2018, arXiv preprint [arXiv:1803.08475](https://arxiv.org/abs/1803.08475).
- [50] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A.N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.
- [51] A. Mazayev, N. Correia, A distributed CoRE-based resource synchronization mechanism, *IEEE Internet Things J.* 7 (5) (2019) 4625–4640.
- [52] International Business Machines (IBM) Corporation, IBM ILOG CPLEX Optimizer, 2021.
- [53] Gurobi Optimization, LLC, Gurobi optimizer reference manual, 2021.
- [54] R. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, *Mach. Learn.* 8 (3-4) (1992) 229–256.
- [55] R.S. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction*, second ed., The MIT Press, 2018.
- [56] Z. Wang, N. de Freitas, M. Lanctot, Dueling Network Architectures for Deep Reinforcement Learning, *Tech. Rep.*, 2015, [arXiv:1511.06581](https://arxiv.org/abs/1511.06581).
- [57] V. Mnih, A.P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, in: M.F. Balcan, K.Q. Weinberger (Eds.), *Proceedings of the 33rd International Conference on Machine Learning*, in: *Proceedings of Machine Learning Research*, vol. 48, PMLR, New York, New York, USA, 2016, pp. 1928–1937.
- [58] S. Gu, T. Hao, A pointer network based deep learning algorithm for 0–1 knapsack problem, in: *2018 Tenth International Conference on Advanced Computational Intelligence, ICACI, IEEE*, 2018, pp. 473–477.
- [59] O. Vinyals, S. Bengio, M. Kudlur, Order matters: Sequence to sequence for sets, 2015, arXiv preprint [arXiv:1511.06391](https://arxiv.org/abs/1511.06391).
- [60] S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, et al., Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, in: *A Field Guide to Dynamical Recurrent Neural Networks*, IEEE Press, 2001.
- [61] T.B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, 2020, arXiv preprint [arXiv:2005.14165](https://arxiv.org/abs/2005.14165).
- [62] Y. Tay, M. Dehghani, D. Bahri, D. Metzler, Efficient transformers: A survey, 2020, arXiv preprint [arXiv:2009.06732](https://arxiv.org/abs/2009.06732).
- [63] Registry and discovery | moleculer - progressive microservices framework for node.js. <https://moleculer.services/docs/0.14/registry.html>.
- [64] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G.S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, 2015, Software available from [tensorflow.org](https://www.tensorflow.org).
- [65] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, in: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, JMLR Workshop and Conference Proceedings*, 2010, pp. 249–256.
- [66] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, 2014, arXiv preprint [arXiv:1412.6980](https://arxiv.org/abs/1412.6980).
- [67] S. Wang, B.Z. Li, M. Khabsa, H. Fang, H. Ma, Linformer: Self-attention with linear complexity, 2020, arXiv preprint [arXiv:2006.04768](https://arxiv.org/abs/2006.04768).
- [68] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, 2017, arXiv preprint [arXiv:1707.06347](https://arxiv.org/abs/1707.06347).
- [69] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, D. Metzler, Long range arena: A benchmark for efficient transformers, 2020, arXiv preprint [arXiv:2011.04006](https://arxiv.org/abs/2011.04006).
- [70] Y. Cheng, D. Wang, P. Zhou, T. Zhang, A survey of model compression and acceleration for deep neural networks, 2017, arXiv preprint [arXiv:1710.09282](https://arxiv.org/abs/1710.09282).