Comparing Actor-Critic and Neuroevolution Approaches for Traffic Offloading in FaaS-powered Edge Systems

Emanuele Petriglia
e.petriglia@campus.unimib.it
University of Milano-Bicocca
Milan, Italy

Federica Filippini
federica.filippini@unimib.it
University of Milano-Bicocca
Milan, Italy

Giacomo Pracucci
g.pracucci@campus.unimib.it
University of Milano-Bicocca
Milan, Italy

Marco Savi
marco.savi@unimib.it
University of Milano-Bicocca
Milan, Italy

Michele Ciavotta
michele.ciavotta@unimib.it
University of Milano-Bicocca
Milan, Italy

ABSTRACT

In a computing context characterized by a complex and interconnected network of heterogeneous devices, which generate enormous amounts of data requiring exchange and near-real-time processing, the collaboration between Edge Computing and Function as a Service (FaaS) models holds significant potential to enhance the flexibility, cost-effectiveness, and responsiveness of applications. However, traditional FaaS encounters challenges in distributed edge environments due to dynamic traffic demands and resource limitations. Effective methodologies must be developed to address the load management issue, which involves determining the allocation of incoming requests to each node and deciding whether to process them locally, reject them, or offload them to neighboring nodes with available resources. This paper investigates and compares various approaches for managing incoming requests in a Decentralized FaaS environment. On the one hand, it considers Actor-Critic Reinforcement Learning algorithms, namely Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). On the other hand, it examines the NeuroEvolution of Augmenting Topologies (NEAT) method. Experimental validation underscores the promising results of PPO, which ensures an average rejection rate of less than 4%.

CCS CONCEPTS

• **Computing methodologies → Reinforcement learning**; **Genetic algorithms**; • **Computer systems organization →** *Peer-to-peer architectures.*

KEYWORDS

Function as a Service, Edge Computing, Load Balancing, Reinforcement Learning

ACM Reference Format:
Emanuele Petriglia, Federica Filippini, Giacomo Pracucci, Marco Savi, and Michele Ciavotta. 2024. Comparing Actor-Critic and Neuroevolution Approaches for Traffic Offloading in FaaS-powered Edge Systems. In *Workshop on Serverless at the Edge (SEATED '24), June 3–7, 2024, Pisa, Italy.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3660319.3660331

1 INTRODUCTION

The Cloud Computing paradigm is widely acknowledged as a fundamental core supporting modern services and applications characterized by high workloads, with a projected market size exceeding USD 2200$ billion in 2030 [5]. The Cloud has revolutionized the allocation, distribution, and consumption of computational resources. Typically based on a pay-as-you-go pricing model, it offers flexible, scalable, and readily accessible resources, eliminating the need to manage their complexity. In this context, the widespread adoption of the Internet of Things (IoT) paradigm has recently led to the deployment of millions of small devices, resulting in complex and interconnected networks where vast amounts of data are exchanged almost in real-time. This trend has spurred the blossoming of the Edge Computing paradigm, which relocates the computation process from centralized servers to the periphery of the network, in close physical proximity to the end devices. This decentralization helps minimize latency, facilitating real-time interactions [13].

Function as a Service (FaaS) is a cloud service model that, when integrated with Edge Computing, enables each edge node to execute short-lived code in response to events without managing the infrastructure associated with building and maintaining microservices applications. This approach significantly enhances the flexibility, cost-effectiveness, and responsiveness of applications [2]. However, when dealing with distributed Edge environments, traditional FaaS is inadequate due to dynamic traffic demands and limited resources [16]. Effective solutions must be devised to address the load management problem, determining the amount of incoming requests each node should serve locally, reject, or offload to neighbors with higher resource availability. To this end, Ciavotta et al. proposed DFaaS [4], a fully decentralized FaaS platform designed to automatically distribute traffic load across autonomous edge nodes in the same network. DFaaS relies on a peer-to-peer network to share information about node states and traffic, enabling suitable agents to make informed decisions regarding the offloading strategy to pursue. While numerous literature proposals rely on heuristic or meta-heuristic approaches to tackle the task offloading problem in edge or edge-cloud environments [3, 12], Reinforcement Learning (RL) is gaining traction in this context [11, 14, 17, 33] due to its ability to continuously and automatically adapt decisions to environmental conditions.

This work presents a preliminary comparison among three distinct approaches for implementing DFaaS agents: two Reinforcement Learning (RL) methods, specifically Proximal Policy Optimization (PPO)[27] and Soft Actor-Critic (SAC)[9], and the NeuroEvolution of Augmenting Topologies (NEAT) algorithm [28]. Despite NEAT not being conventionally categorized within the RL taxonomy, it can be regarded as an RL-based approach due to its incorporation of automatic evolution in Deep Neural Networks (DNNs) based on environment characteristics and responses to agent decisions [24]. Our experimental findings demonstrate the effectiveness of PPO in load management for FaaS-based applications, achieving an average rejection rate of less than 4%.

The paper is organized as follows: Section 2 briefly overviews the state of the art. Section 3 presents the working scenario of DFaaS, while the traffic offloading problem is formulated in Section 4 to be exploited in the RL context. Section 5 details the proposed algorithms, while Section 6 describes their experimental validation. Finally, conclusions and future work are discussed in Section 7.

2 RELATED WORK

This section reviews state-of-the-art proposals in the context of tasks offloading in edge and edge-cloud environments, considering both heuristic [3, 12] and RL-based approaches [17, 33]. Indeed, Machine Learning and, particularly, RL are increasingly popular in this context since they can take quick offloading decisions even in partially-observable environments [11, 15].

The proposal by [3] introduces a dynamic priority-based computation scheduling and offloading algorithm, employing (i) a multiple knapsack-based heuristic algorithm for managing high-priority tasks, and (ii) a task weight and data size-based computation scheduling and offloading algorithm for medium-priority tasks in a mobile edge computing environment. An edge-cloud offloading mechanism based on a Knapsack Potential Game to derive an optimal offloading ratio for each edge server, aiming to balance the cost-effectiveness of the overall system is proposed by [12]. The work of Liu et al. [17] focuses on minimizing the system deadline violation ratio considering incoming load and delay constraints, utilizing a deep deterministic policy gradient-based learning algorithm to determine the optimal offloading policy for mobile applications with task-dependency requirements in a mobile edge computing environment. [33] proposes a multi-objective RL algorithm based on double deep Q-network to dynamically approximate the optimal offloading decision in the context of Internet of Vehicles.

While all the mentioned approaches highlight the promising results achieved by RL methods in tackling offloading problems, none specifically focuses on FaaS systems, which are the main target of our work. As far as FaaS is concerned, [6] proposes a load-balancing algorithm designed for FaaS applications, which aims to balance locality awareness, load distribution, and randomness. The authors employ consistent hashing, establishing a stable mapping between objects and servers to enhance locality while simultaneously addressing issues related to server loads, cold-start overheads of different functions, and bursty traffic. While closely related to our work in managing the load of serverless applications, [6] focuses on general servers and does not consider the specific challenges posed by Edge environments.

3 DECENTRALIZED FAAS (DFAAS)

Figure 1 illustrates the reference scenario of DFaaS [4], comprising a network of autonomous FaaS-enabled edge nodes distributed peripherally and geographically. Each node is equipped with a platform based on OpenFaaS [22], facilitating the execution of serverless functions. Function execution requests (i.e., HTTP requests) generated by a client connected to the nearest access point are received by a single edge node. As depicted in the figure, requests can be autonomously forwarded to other edge nodes when necessary, such as in cases of node overload.

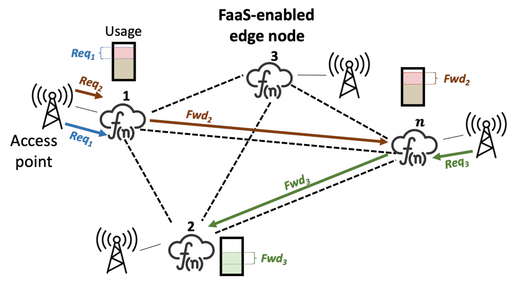


Figure 1: Reference DFaaS scenario [4].

Figure 2 illustrates the high-level architecture of the DFaaS platform. Each node consists of three primary components: an agent, a proxy, and a FaaS platform. The *agent* maintains the peer-to-peer network, monitoring the local node state by collecting metrics and predicting incoming load for individual function classes in the next time slot. Moreover, it negotiates resources with neighboring nodes to handle the load, either forwarding or accepting requests. Upon resource negotiation, the *proxy* component (implemented via HAProxy [10]) configures traffic redirection or acceptance. The agent communicates this configuration to the proxy, which then forwards requests to other nodes or to the local *FaaS platform* for execution, selecting, instantiating, and executing the corresponding function. Internally, the FaaS platform integrates a gateway and a queue for managing requests that cannot be immediately processed.

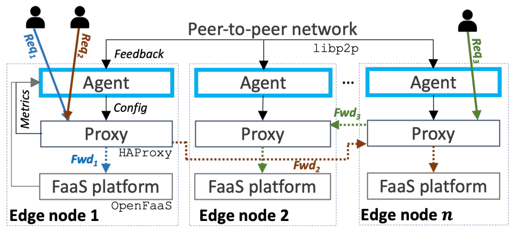


Figure 2: DFaaS platform: proposed architecture [4].

4 REINFORCEMENT LEARNING PROBLEM

This study investigates and compares AI-powered offloading strategies, focusing on the management of request acceptance for local processing, rejection, or redirection to neighboring nodes to ensure balanced and continuous request processing. To tackle the complexity of this subject, we examine state-of-the-art Actor-Critic RL and e NeuroEvolution approaches, noting that for the purposes of this paper, NeuroEvolution is regarded as an integral component of reinforcement learning methodologies.

RL has recently become popular in the literature to address scaling and offloading problems in edge computing scenarios [11, 15]. These scenarios are highly dynamic and typically too complex to be effectively modeled analytically. RL has the ability to learn from interactions with the environment and automatically adapt over time, deciding the proportion of requests to process locally, to forward to other nodes or to reject ensuring an effective resource utilization, reduced latency, and improved user experience.

In general, RL algorithms consist of one or multiple agents that, in each time step t, automatically learn how to map the current state $s(t)$ to an action $a(t)$, based on the interaction with an environment that reacts to the agent decisions by providing a numeric reward signal $r(t)$. The state-action mapping, called policy, is iteratively updated balancing exploration (i.e., selecting a not-yet-chosen action to visit new states with possibly higher reward) and exploitation (i.e., using the accumulated experience to choose the current best action) [29]. The reward associated with each state-action pair usually depends also on the transition that occurs in the environment, i.e., on the new state $s' = s(t+1)$ the agent will observe in the next instant. We write therefore $r(t) = r(s, a, s')$. It is crucial to note that the next state s', albeit related to s and to the selected action a, may be at least partially unpredictable, due to environment dynamics that cannot be modeled or observed efficiently. The global expected return is defined as $G = \sum_{k=0}^{\infty} \gamma^k r(t+k+1)$, where the parameter γ is used to discount the expected reward in future time steps, thus limiting the impact of distant choices on the learned policy.

In this work, we considered a single-agent RL system whose main goal is to prioritize local processing of requests without reaching a congestion state, while also learning to reject or forward requests when necessary. Albeit preliminary, our analysis aims at investigating whether RL can effectively tackle the offloading problem in this setting, with the ultimate goal of designing a multi-agent or federated RL method to manage and distribute the incoming load in the DFaaS platform.

The agent distributes the incoming load based on the node state and other system information, encoded in the so-called *observation*, by taking appropriate *actions* that are rewarded to guide the decisions towards reducing the number of rejections and the permanence in a congestion state. The state space (or space of observations) $\mathcal{S}$, action space $\mathcal{A}$ and reward function $r(s, a, s')$ for our problem are characterized in the following.

State space. Each state $s \in \mathcal{S}$ includes four variables: the number of incoming requests, the capacity of the local processing queue, the forwarding capacity (i.e., the number of requests that neighboring nodes can accept), and the node state (i.e., if we are observing congestion). In particular, we assume that a node is in a congested (or overloaded) state if either the local processing queue is full or the total number of requests redirected to it by neighboring nodes exceeds its forwarding capacity. As already mentioned in Section 3, a node in a congested state will reject all requests until the resource availability is restored.

Action space. The agent decision involves determining the number of requests to process locally, forward to neighboring nodes and reject. Formally, we consider a continuous action space $\mathcal{A}$ whose elements are tuples $a = (p_{loc}, p_{fwd}, p_{rej})$ where each p_j is the percentage of requests to be processed, offloaded or rejected, respectively ($p_{loc} + p_{fwd} + p_{rej} = 1$). Figure 3 illustrates an example of action taken when the incoming load includes 100 requests: 70 of them are accepted locally, 20 are forwarded and 10 are rejected.

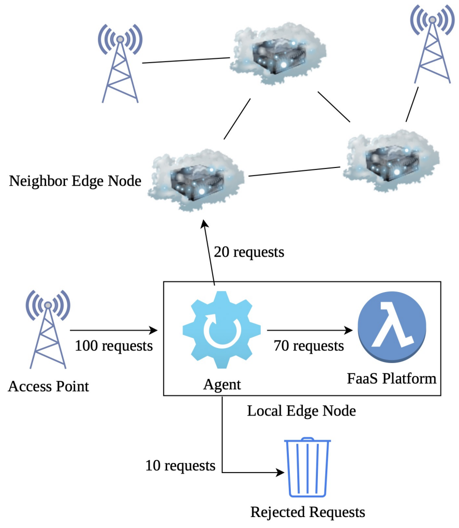


Figure 3: An example of agent action.

Note that in this preliminary work we do not tackle explicitly the problem of identifying the destination node for the forwarded requests. We may assume that this further decision is delegated to a suitable heuristic as in [4].

Reward function. The reward function is the sum of three terms, related to the corresponding elements of the agent action and weighted according to the values of p_{loc}, p_{fwd} and p_{rej}, respectively. Each term is further multiplied by a factor to adjust the importance of the actions based on the node state, i.e., on whether it is congested or not. Indeed, if the system is not congested, the agent should prefer to process the requests locally, but without overloading the node. The reward for local processing is given proportionally to the *queue factor* φ_{queue}, which measures the queue capacity on a scale from 0 (full) to 1 (empty). The reward for forwarding requests considers the φ_{queue} and the *forwarding factor* φ_{fwd}, which measures the availability of the neighboring nodes from 0 (minimum availability) to 1 (maximum availability); it increases as the incoming load and forwarding capacity increase. Finally, rejected requests are penalized rather than rewarded: the penalty is determined by φ_{queue} and φ_{fwd}, and it decreases as the incoming load increases and the forwarding capacity decreases.

Note that, if the node is overloaded, the agent should decide not to process requests locally. A corresponding penalty is applied, forcing the agent to reject incoming requests in order to prevent further aggravation of the congested stage. An additional fixed penalty is defined to discourage the agent from reaching this state.

Details about the algorithms and approaches we considered are provided in the following section.

5 ALGORITHMS

Conventional tabular RL approaches encounter limitations when dealing with large or continuous action and state spaces, as those considered in this context [19]. Therefore, we investigate three approaches, two of which are based on Actor-Critic Deep Reinforcement Learning (DRL), and one on NeuroEvolution of Augmenting Topologies [7]. DRL is a branch of RL that combines reinforcement learning and deep learning, utilizing neural networks to approximate the values achieved by the agent in different states according to its policy. Among the existing DRL methods, we consider the Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) approaches. Like other policy-gradient methods, they are designed to directly learn the optimal policy without explicitly approximating the value of each possible action. This characteristic is crucial in our context, where the action space is continuous, making a direct estimate of the action values infeasible [8]. Furthermore, both PPO and SAC are based on the *Actor-Critic* architecture, where two DNNs interact to collectively design the best policy. In particular, the network denoted as the *actor* decides, in each state, which action to perform, while the *critic* evaluates its decisions by providing an error signal used to optimize future choices.

Neuroevolution refers to the application of evolutionary algorithms to the automatic design, training and optimization of DNNs: across multiple iterations (or generations), multiple solutions are explored by selecting, combining and mutating DNN characteristics as, e.g., the number and connections between neurons, rather than updating the DNN weights as in more traditional approaches. We compare the performance of PPO and SAC with the NeuroEvolution of Augmenting Topologies (NEAT) algorithm, which proved to be effective in producing a good-quality DNN faster than other RL methods in some scenarios [30].

We realized a custom implementation of PPO and SAC using PyTorch [23], while NEAT is implemented using neat-python [18]. Gymnasium [32] was used to define the environment. Details about the three algorithms and related changes to adapt to our context are provided in the following sections.

Proximal Policy Optimization. PPO is a policy-gradient, model-free RL algorithm based on the Actor-Critic architecture. The algorithm optimizes a surrogate objective function using stochastic gradient ascent [27]. PPO is designed to be scalable for large models and parallel implementation, and robust regarding the choice of hyperparameters. Due to its robust performance and minimal need for tuning, it has been designated as the default algorithm in prominent RL frameworks such as StableBaselines3 [26].

Several variants of this algorithm have been proposed; in this work, we consider in particular the *PPO-Clip* algorithm, which introduces the concept of a trust region to prevent, through a suitable parameter *clip* ϵ, sudden policy updates that may result in incorrect optimizations. It is characterized by two additional parameters: *Generalized Advantage Estimation (GAE)* λ, an interpolation factor that weights the differences among the advantages estimated over multiple time steps, and *Entropy Coefficient*, which regulates the balance between exploration and exploitation.

Soft Actor-Critic. SAC is a RL algorithm designed for continuous action spaces. Its primary characteristic is entropy regularization: the policy is trained to maximize a trade-off between the reward function and entropy, which quantifies the randomness in the policy [9]. The high entropy within a SAC policy promotes extensive exploration of the space and mitigates premature convergence to suboptimal policies. SAC operates as an off-policy algorithm capable of learning from past experiences and trajectories generated by various policies. The algorithm utilizes three neural networks: a state value function V, a soft Q-function Q, and a policy function π, each separately approximated and optimized to facilitate convergence. A *soft update* parameter τ is employed to regulate the frequency at which the target network weights are updated based on the weights of the policy network.

Context-specific adaptations to PPO and SAC. The rationale behind opting for a custom implementation of Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) lies in the necessity to define an appropriate approach for characterizing the action probability distribution. In general, algorithms designed for continuous action spaces cannot generate a policy by learning the probability of selecting a specific action a in the current state $s(t)$, as is typically done when the set $\mathcal{A}$ has a finite (and possibly not excessively large) number of elements. Instead, the agent learns to estimate, based on the current state, the parameters (e.g., mean and standard deviation) of a probability distribution over actions [29, 8]. Traditional PPO and SAC algorithms typically adopt a Gaussian probability distribution [27, 9], possibly with slight modifications if the action space is bounded. However, this choice is unsuitable in our context, where each action $a = (p_{loc}, p_{fwd}, p_{rej})$ must satisfy the specific condition $p_{loc} + p_{fwd} + p_{rej} = 1$ (see Section 4).

To address this, following proposals from other literature [31], we opted to utilize the Dirichlet probability distribution [21], which is highly effective in characterizing a decision space where actions are chosen in the simplex [20]. The agent will learn the concentration parameter α, which determines the distribution's shape. Since this parameter must be strictly positive, we modeled the actor network of both PPO and SAC by incorporating an exponential activation function in the last layer. It's noteworthy that an alternative choice for a probability distribution ensuring $p_{loc} + p_{fwd} + p_{rej} = 1$ is the Gaussian-softmax. However, this would introduce training biases due to the normalization of actions (to enforce $p_{loc} + p_{fwd} + p_{rej} = 1$), the non-injectivity of the softmax function, and its invariance with respect to translations [31].

NeuroEvolution of Augmenting Topologies. Traditional DL methods involve updating the DNN weights through methods such as gradient descent. In contrast, NEAT evolves both the weights and the network topology [28], starting from a minimal network and allowing it to evolve and increase in complexity only when deemed beneficial for the considered problem. Abstracting from natural evolutionary processes and genetics, NEAT encodes information about the DNN structure and connection weights as the genotype, while characterizing the phenotype (i.e., the concrete expression of the genotype) as the specific network built and trained based on this general structure. To address the issue of competing conventions [25], which arises when damaged networks result from a crossover between genomes with different encodings but representing the same solution, NEAT utilizes historical markers known as innovation numbers, which enable tracking the origin of each gene, thus facilitating a coordinated and coherent evolution of the network topology. NEAT comprises several parameters that require tuning to optimize its performance; among others, the mutation probability of activation functions, the probability of adding or removing nodes and connections from the network, and the mutation probability of connection weights.

6 EXPERIMENTAL ANALYSIS

This section presents the results of the experimental analysis conducted to compare the proposed algorithms in a simulated scenario. Specifically, Section 6.1 outlines the experimental setup and methodology employed, detailing the definition of our evaluation metrics. Section 6.2 further elaborates on the obtained results.

6.1 Experimental setup and methodology

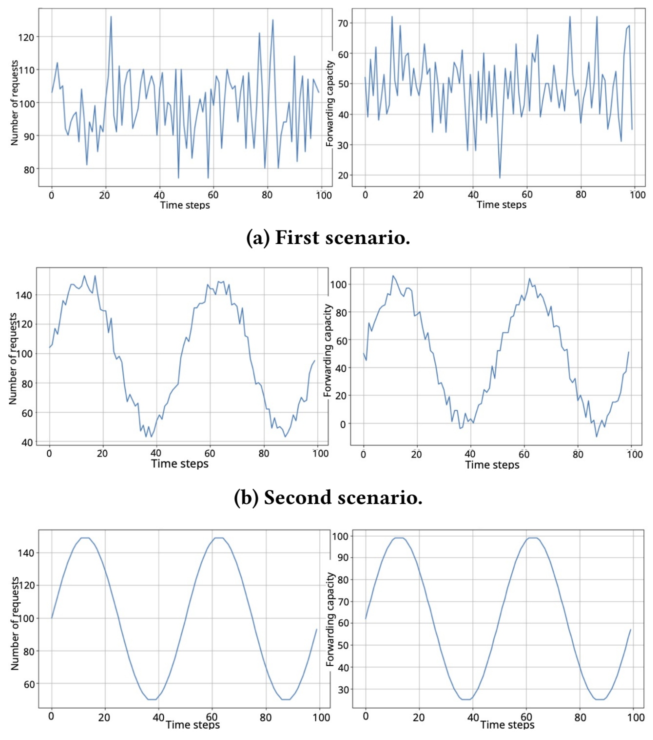
The objective of the experimental analysis is to assess the efficacy and robustness of the PPO, SAC, and NEAT algorithms within a simulated DFaaS environment. Specifically, our focus lies in comparing the generalization capabilities of these three methods and assessing whether an agent trained under a specific load setting can effectively handle situations characterized by varying distributions of incoming requests and forwarding capacity. To achieve this goal, we designed three distinct scenarios, each involving a total of $n = 10k$ incoming requests, delineated as follows: The first scenario, illustrated in Figure 4a, represents the most unpredictable conditions, with request and forwarding capacity values generated according to a Gaussian distribution. The second and third scenarios, depicted in Figures 4b and 4c, employ a sinusoidal function. In the former, we introduce random noise to increase the complexity level encountered by the agent.

We considered three categories of requests, characterized by varying CPU and memory demands, which are randomly sampled from Gaussian distributions defined by the following parameters:

A) between 1 and 10 CPU shares (with mean 5 and standard deviation 2.5), defined as the portion of CPU that can be assigned to process each function, and between 1MB and 25MB of RAM (mean 12.5MB, standard deviation 2.5);
B) between 11 and 20 CPU shares (with mean equal to 15 and standard deviation 2.5) and between 26MB and 50MB of RAM (with mean 38MB and standard deviation 2.5);
C) between 21 and 30 CPU shares (with mean equal to 25 and standard deviation 2.5) and between 51MB and 75MB of RAM (with mean 63MB and standard deviation 2.5);

During the generation process, each one of the n incoming requests is extracted from class A, B or C with uniform probability. Given the preliminary nature of this work, we only considered a single agent deployed on a single edge node, with a local processing queue capacity of 100 requests, and a maximum availability of 1000 CPU shares and 8000MB of RAM.

For each algorithm, the experiments were performed in two different settings: one with the standard hyperparameter values



(a) First scenario.

(b) Second scenario.

(c) Third scenario.

Figure 4: Number of requests (left) and forwarding capacity (right) over time in the three designed scenarios.

suggested by the original authors in the respective papers, and one with tuned hyperparameters, determined by a Bayesian optimization on 100 execution trials using the Optuna framework [1]. Only the most impactful hyperparameters were tuned, as shown in Table 1. It is relevant to note that, due to its evolutionary nature, which embeds stochastic processes of mutation and selection, NEAT does not consistently produce similar results in different executions even when the environment and hyperparameters are kept constant. This variation influences the comparison between standard and tuned hyperparameters, which may be less significant than what is observed for PPO and SAC.

For each scenario and algorithm, we conducted 5 training experiments with 5 different seeds, each repeated using both standard and tuned hyperparameters, resulting in a total of 90 runs. Each session of PPO and SAC training consisted of 1000 episodes including 100 steps each, resulting in a total of $100k$ training steps per session. NEAT began with 100 neural networks, and each session lasted for 100 generations.

At the conclusion of each training session, the algorithms' performance was assessed utilizing four principal metrics: the reward average and standard deviation, the number of steps in a congestion state, and the number of rejected requests. Average reward is a paramount indicator to consider, as it inherently encapsulates other metrics (refer to Section 4). Improvement in any tracked metric correlates with enhanced reward. The standard deviation of the reward

Table 1: Training hyperparameters for each algorithm. If a tuned column cell is empty, the standard value is used.

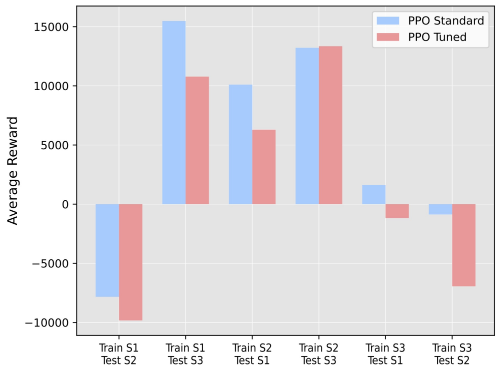
Method	Hyperparameter	Standard	Tuned
PPO [27]	γ	0.99	0.91
	GAE λ	0.95	—
	Clip ϵ	0.2	—
	Entropy Coefficient	0.01	—
	Learning Rate	0.0003	0.0007
	Nodes in hidden layers	256	167
SAC [9]	γ	0.99	0.90
	τ	0.05	0.003
	Learning Rate	0.0001	0.0081
	Nodes in hidden layers	256	206
NEAT [28]	Connection add probability	0.50	0.52
	Node add probability	0.30	0.50
	Weight mutate rate	0.80	0.88
	Survival threshold	0.20	—

is computed based on the average reward across all training runs with varying seeds, considering a single scenario and algorithm. A low value signifies consistent performance and predictability of the algorithm under diverse initial conditions. Finally, a reduced number of rejected requests indicates efficient request distribution by the agent, thereby ensuring a high level of service quality.

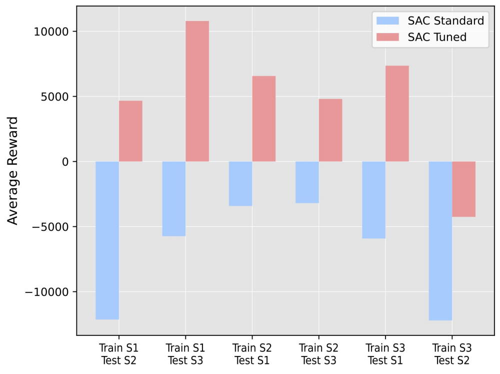
6.2 Experimental results

In line with our earlier discussion, our primary aim in this experimental evaluation is to compare how well the PPO, SAC, and NEAT algorithms can adapt to different scenarios. Therefore, all the results presented below were obtained by training the agents on one of the three scenarios outlined in Figure 4 and then testing them on the remaining scenarios.

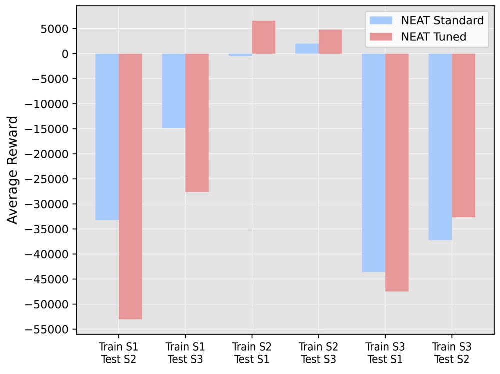
Figure 5 illustrates the average reward across all combinations of training and test scenarios for each algorithm, shedding light on the effects of hyperparameter tuning. Looking at Figure 5a, we can see that PPO performs consistently well when trained on scenario 2, even when tested on scenarios 1 and 3 (third and fourth sets of bars). Interestingly, in these cases, tweaking the hyperparameters does not seem to improve the average reward much, possibly because it leads to overfitting to the specific conditions encountered during training. Similar trends are observed for other combinations of training and testing scenarios, albeit with lower overall performance. This can be attributed to the distinct load distributions characterizing scenarios 1 and 3, making it challenging for the trained agent to generalize across different settings. In contrast, SAC performs significantly better with tuned hyperparameters compared to standard settings (see Figure 5b), demonstrating robust generalization capabilities. However, the average reward is lower than that of PPO, and the SAC algorithm leads to an increase in the number of rejected requests and the number of steps in a congested state during the training process, as illustrated in Figure 6a. Finally, the generalization capabilities of NEAT (Figure 5c) are generally poor, and no clear benefit is observed from hyperparameter tuning except in the two instances where the method is trained in scenario 2. As previously mentioned, this is generally the best training setting, since the load is less biased towards either a stochastic or a purely deterministic pattern.



(a) PPO average reward.



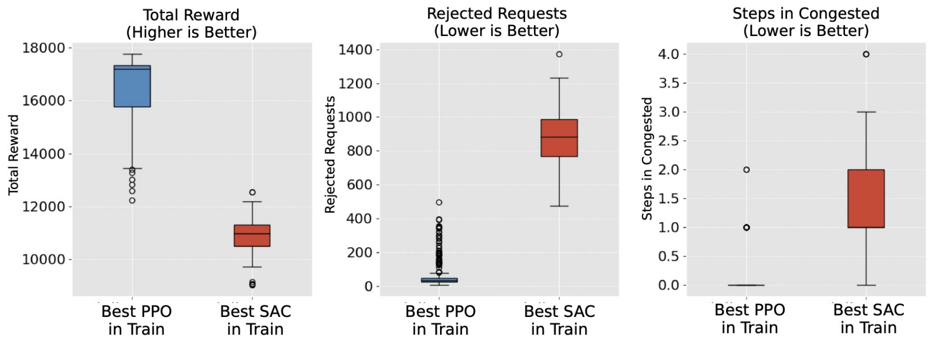
(b) SAC average reward.



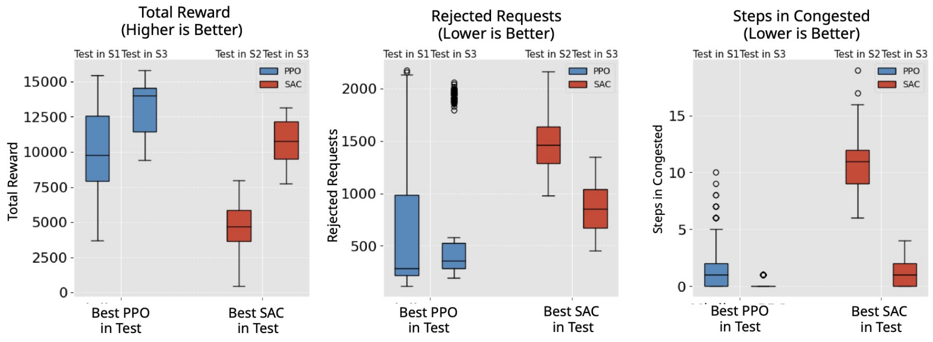
(c) NEAT average reward.

Figure 5: Impact of hyperparameters tuning.

The comparison between PPO and SAC across the three considered metrics is presented in Figure 6, encompassing both training and test results. The results consistently demonstrate the superior performance of PPO, which effectively manages a higher number of requests per episode on average (10, 000), with a rejection rate of less than 4%. Moreover, the PPO agent exhibits prompt responsiveness in restoring optimal working conditions during congested



(a) Training results.



(b) Test results.

Figure 6: Comparision between PPO and SAC.

situations, as evidenced by the moderate number of steps in a congested state.

As previously observed, the PPO agent trained in the second scenario emerges as the top performer. Although the agent trained in the third scenario demonstrates the highest average reward during testing within the same setting, its performance declines significantly when tested in a different scenario, indicating clear overfitting attributed to the simplistic nature of the observed load distribution. Conversely, the right plot of Figure 6 displays that while the results achieved by an agent trained and tested on scenario 2 are slightly lower, it learns a policy that generalizes remarkably well to unseen settings. A similar pattern, though less pronounced, is also observable in SAC.

NEAT is the algorithm exhibiting the poorest performance across all settings, as highlighted in Figure 8 (representing the third scenario). This is partially attributable to the tendency of the agent to prioritize forwarding requests over local processing. The queue factor φ_{queue} (light blue bars in the figure), defined in Section 4, is always very close to 1, showing an almost empty queue in all timesteps; while this situation should encourage the agent in prioritizing the local computation, the number of forwarded requests is often significantly high. The promising results achieved by the agent trained in the second scenario (with a positive average reward, as shown in Figure 5c) are negatively balanced by a rejection rate of around 30–40%.

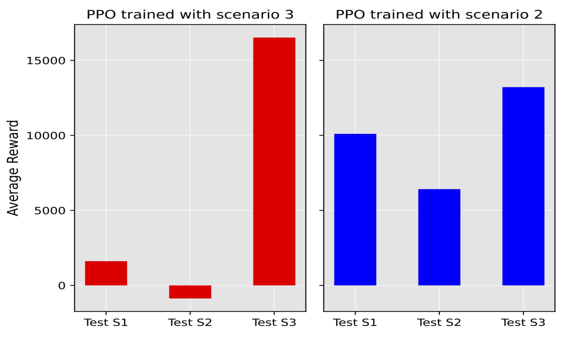


Figure 7: PPO Standard fitting.

The results suggest that utilizing RL for load management in DFaaS-Edge systems poses challenges, yet yields promising performance and underscores avenues for further development. One notable challenge observed is overfitting, underscored by the importance of meticulously crafting the training environment. Specifically, an agent exposed solely to a simplistic scenario risks overfitting and underperforming in unknown scenarios, as it might adopt highly specialized policies tailored exclusively to the considered setting. Another significant aspect concerns the challenges encountered in the second scenario: during training, the average reward is inferior compared to other scenarios, albeit demonstrating superior generalization during testing. Given the heightened noise

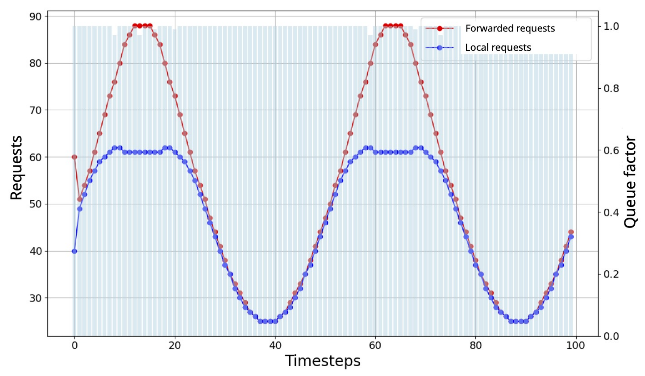


Figure 8: NEAT queue and forwarding factor in scenario 3.

in the load pattern relative to the third scenario, striking a balance between exploration and exploitation becomes pivotal to mitigate excessive policy updates that may prove suboptimal in the long run. Lastly, a notable observation is the agent's inclination to reject an excessive number of requests to prevent congestion. This tendency may stem from an imbalance in the components constituting the reward function, which may be adjusted in future works to mitigate this undesired behavior.

7 CONCLUSION

This paper presents an initial approach to workload distribution in decentralized FaaS systems at the edge using RL algorithms. The PPO, SAC, and NEAT methods were implemented and tested in a simulated DFaaS environment to determine their ability to handle requests optimally. After tuning the hyperparameters and setting up the scenarios, the results showed that PPO outperformed the other algorithms, yielding an average rejection rate below 4%.

In future works, a multi-agent system could be explored to improve collaboration and coordination between edge nodes. A more realistic simulated environment could be developed for training purposes, and new requests could be generated to extend the range of possible scenarios. Finally, new RL algorithms could be tested to find better alternatives for workload distribution.

ACKNOWLEDGMENTS

This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART")

REFERENCES

[1] Takuya Akiba et al. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In *ACM SIGKDD Proc.* 2623–2631.
[2] Paul Castro et al. 2019. The rise of serverless computing. *Commun. ACM*, 62, 12, (Nov. 2019), 44–54.
[3] Rong Chai et al. 2021. Dynamic Priority-Based Computation Scheduling and Offloading for Interdependent Tasks: Leveraging Parallel Transmission and Execution. *IEEE TVT*, 70, 10, 10970–10985.
[4] Michele Ciavotta et al. 2021. DFaaS: Decentralized Function-as-a-Service for Federated Edge Computing. In *IEEE CloudNet*, 1–4.
[5] 2023. FORTUNE Business Insights. Cloud Computing Market Size. Retrieved 2024-03-05 from https://www.fortunebusinessinsights.com/cloud-computing-market-102697.
[6] Alexander Fuerst and Prateek Sharma. 2022. Locality-aware Load-Balancing For Serverless Clusters. In *ACM HPDC Proc.* 227–239.
[7] Edgar Galva'n and Peter Mooney. 2021. Neuroevolution in deep neural networks: current trends and future challenges. *IEEE TAI*, 2, 6, 476–493.
[8] Laura Graesser and Wah Loon Keng. 2020. *Foundations of Deep Reinforcement Learning: Theory and Practice in Python. Addison-Wesley data and analytics series*, 379 pages.
[9] Tuomas Haarnoja et al. 2018. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. (2018). arXiv: 1801.01290 [cs.LG].
[10] HAProxy. 2023. HAProxy. The Reliable, High Performance TCP/HTTP Load Balancer. http://www.haproxy.org. Accessed: (04/03/2024). (2023).
[11] Diego Hortelano et al. 2023. A comprehensive survey on reinforcement-learning-based computation offloading techniques in Edge Computing Systems. *J. Netw. Comput. Appl.*, 216, 103669.
[12] Cheng-Ying Hsieh et al. 2023. Edge-Cloud Offloading: Knapsack Potential Game in 5G Multi-Access Edge Computing. *IEEE Wirel. Commun.*, 22, 11, 7158–7171.
[13] Michaela Iorga et al. 2018. Fog Computing Conceptual Model (NIST Special Publication 500-325). Retrieved 2024-03-07 from https://doi.org/10.6028/NIST.SP.500-325.
[14] Abednego Wamuhindo Kambale et al. 2024. Runtime Management of Artificial Intelligence Applications for Smart Eyewears. In *IEEE/ACM UCC* (UCC '23) Article 31.
[15] Binayak Kar et al. 2023. Offloading using traditional optimization and machine learning in federated cloud–edge–fog systems: a survey. *IEEE Commun. Surv. Tutor.*, 25, 2, 1199–1226.
[16] George Kousiouris and Dimosthenis Kyriazis. 2021. Functionalities, Challenges and Enablers for a Generalized FaaS based Architecture as the Realizer of Cloud/Edge Continuum Interplay. In *CLOSER*.
[17] Shumei Liu et al. 2023. Dependent Task Scheduling and Offloading for Minimizing Deadline Violation Ratio in Mobile Edge Computing Networks. *IEEE J. Sel. Areas Commun.*, 41, 2, 538–554.
[18] [SW] Alan McIntyre et al., neat-python. URL: https://neat-python.readthedocs.io, VCS: https://github.com/CodeReclaimers/neat-python.
[19] Volodymyr Mnih et al. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR*, abs/1312.5602. http://arxiv.org/abs/1312.5602.
[20] 1993. *Simplices. Elements of Algebraic Topology*. Perseus Books Pub. Chap. 1.
[21] Kai Wang Ng et al. 2011. *Dirichlet and Related Distributions: Theory, Methods and Applications. Wiley Series in Probability and Statistics*. John Wiley & Sons, Ltd.
[22] OpenFaaS. 2023. OpenFaaS. Serverless Functions, Made Simple. https://www.openfaas.com. Accessed: (04/03/2024). (2023).
[23] Adam Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, 8024–8035.
[24] Yiming Peng et al. 2018. NEAT for large-scale reinforcement learning through evolutionary feature learning and policy gradient search. In *ACM GECCO Proc.* 490–497.
[25] Nicholas J. Radcliffe. 1993. Genetic set recombination and its application to neural network topology optimisation. *Neural Computing & Applications*, 1, 67–90.
[26] [SW] Antonin Raffin et al., StableBaselines3 2021. URL: https://stable-baselines3.readthedocs.io/, VCS: https://github.com/DLR-RM/stable-baselines3.
[27] John Schulman et al. 2017. Proximal Policy Optimization Algorithms. (2017). arXiv: 1707.06347 [cs.LG].
[28] Kenneth O. Stanley and Risto Miikkulainen. 2002. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10, 2, 99–127.
[29] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT Press.
[30] Matthew E. Taylor et al. 2006. Comparing evolutionary and temporal difference methods in a reinforcement learning domain. In *ACM SIGEVO Proc.* (GECCO '06), 1321–1328.
[31] Yuan Tian et al. 2022. A prescriptive Dirichlet power allocation policy with deep reinforcement learning. *Reliab. Eng. Syst. Saf.*, 224, 108529.
[32] Mark Towers et al. 2023. Gymnasium. (Mar. 2023). Retrieved July 8, 2023 from https://zenodo.org/record/8127025.
[33] Xiangjun Zhang et al. 2023. RMDDQN-Learning: Computation Offloading Algorithm Based on Dynamic Adaptive Multi -Objective Reinforcement Learning in Internet of Vehicles. *IEEE TVT*, 72, 9, 11374–11388.