TÉCNICO LISBOA

Fog Computing Task Offloading Optimization based on Deep Reinforcement Learning

Luís Pires de Matos Morgado Ferreira

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Prof. António Manuel Raminhos Cordeiro Grilo

Examination Committee

Chairperson: Prof. Teresa Maria Sá Ferreira Vazão Vasques
Supervisor: Prof. António Manuel Raminhos Cordeiro Grilo
Member of the Committee: Prof. Paolo Romano

March 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Abstract

Industry and society are constantly changing, evolving into an ever more technology oriented system. The ubiquity of Internet of Things (IoT) comes to connect the surroundings through sensors and actuators to the Internet. Cloud Computing (CC) comes as a computing paradigm that can also leverage IoT. However, it has its shortcomings regarding communication bandwidth and latency. Fog Computing (FC) comes as an extension of CC in order to tackle the aforementioned problems, shifting partial computation closer to the edge and managing communications to the Cloud. To make this technology viable, it is first necessary to manage the resources present in the Fog. A proposed concept by academia is a choreography styled resource management, where each Fog node makes its own decisions based on requests from the end devices in a group cooperation to improve the overall performance. Therefore, it is necessary to implement an algorithm that makes the decisions with awareness of other nodes. Studies have been made to introduce Deep Reinforcement Learning (DRL) as a viable decision maker. To create groundwork for standardized further studies, a simulator based on an open source DRL tool is developed. Then, a Proximal Policy Optimization (PPO) with an Advantage Actor-Critic (A2C) architecture algorithm is implemented and tested in a set of cases in the developed tool. In the developed scenarios, the choreography PPO algorithm demonstrated the capability to surpass the non intelligent baselines algorithms when optimizing based on a common reward. However, some limitations are found and should be addressed in future works.

Keywords

Fog Computing, Resource Management, Choreography, Reinforcement Learning, Deep Reinforcement Learning, Proximal Policy Optimization.

Resumo

A indústria e a sociedade estão em mudança e tendem a evoluir para um sistema cada vez mais orientado para a tecnologia. A ubiquidade do IoT é de grande importância para o futuro da tecnologia ao conectar o ambiente envolvente à rede. O CC, que é um paradigma computacional que desloca a necessidade de recursos do utilizador para a Cloud, vem como um complemento ao IoT. No entanto, o CC tem as suas limitações tanto na banda de rede como na latência. FC vem como uma resposta a estes problemas, procurando fazer a ponte entre os clientes e a Cloud com processamento parcial ou total dos dados e gestão de ligações. Para tornar esta tecnologia viável é necessário gerir os recursos do Fog. Uma forma de gestão de recursos proposta pelos académicos é a coreografia, onde cada nó no Fog faz as suas próprias decisões de gestão com o objetivo de cooperação entre nós e melhoramento do desempenho geral. Para criar uma base de trabalho para estudos padronizados, um simulador baseado numa ferramenta de código aberto para DRL é desenvolvida. Depois, um algoritmo PPO com arquitetura A2C é implementado e testado para o conjunto de casos de teste na ferramenta desenvolvida. Nas simulações, o algoritmo PPO de coreografia demostra a capacidade de superar algoritmos não inteligentes quando otimiza uma recompensa de desempenho comum a todos os nós. No entanto, algumas limitações são identificadas por forma a serem endereçadas em futuros estudos.

Palavras Chave

Computação na Névoa, Gestão de recursos, Coreografia, Aprendizagem por Reforço, Aprendizagem por Reforço Profunda, Otimização de Política Proximal .

Contents

List of Figures

Acronyms

A2C	Advantage Actor-Critic
A3C	Asynchronous Advantage Actor-Critic
CC	Cloud Computing
CNN	Convolutional Neural Network
DNN	Deep Neural Network
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
FC	Fog Computing
IoT	Internet of Things
IoV	Internet of Vehicles
MDP	Markov Decision Process
NN	Neural Networks
OOP	Object Oriented Programming
PPO	Proximal Policy Optimization
POMDP	Partially Observable Markov Decision Process
QoS	Quality of Service
RA	Reference Architecture
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SDN	Software Defined Network
TD	Temporal-Difference
UE	User Equipment

1

Introduction

Contents

1.1 Context

Cloud Computing (CC) is a computing paradigm where the resources utilized by a user are located on the cloud. This seemingly infinite computing power allows the user to make highly complex computational tasks with low effort. Internet of Things (IoT) takes great advantage in the use of CC, since it means smaller connected devices are still able to perform intricate operations as long as there is connection. However, CC has some shortcomings. First, some tasks might have a strict deadline that cannot be held because of a large unpredictable latency in the connection to the cloud. Second, with the growth in number of IoT devices and all the data they generate, the bandwidth required in communication becomes too large for a centralized process.

In order to tackle the constraints found in CC, many computing paradigms have been proposed. A study encompassing the most notorious computing paradigms is done in [YFN+18]. Among them, Mist Computing, Edge Computing, and Fog Computing (FC) take relevance. Mist Computing refers to a computing paradigm at the end devices, i.e. at IoT devices level. Edge Computing proposes a computing location near the end devices, as far as one hop, in small data centers at the edge. Although its definition depends on the literature, here FC is defined as a concept that seeks to complement all the computing paradigms placed between end devices and the cloud, to work as the network link with computing and storage capabilities. Due to its versatility and wide use-case range, FC, proposed by both industry and academia, is widely accepted as a new computing paradigm [Con17]. The FC proposal architecture was designed to support the real-time and bandwidth constraints of the ever-growing use of IoT applications.

It can be said that FC is a mid level architecture meant to bridge end-users with the cloud. It takes some toll from the network links to the cloud, helping to manage the resources allocated, possibly improving Quality of Service (QoS) in real-time applications and improving overall efficiency. Rather than replacing the current CC paradigm, it exists to complement it.

To achieve and deploy an FC network, it is necessary to solve the problem of resource management, in either a centralized or distributed system approach, in order to achieve optimal delay in task execution within physical constraints. This management is focused in computing, storage, and communication resources. It will translate into a problem of task offloading, i.e. allocate atomic tasks to another computing device rather than the current device. A problem of caching, i.e. defining local temporary storage by frequency of request in applications to reduce content delivery time. Plus, a problem of bandwidth management. All problems to be approached with multiple environment associated constraints.

Industrial IoT, FC, and Edge computing have a promising future with the association of the two largest international consortia in industrial IoT (OpenFog Consortium and Industrial Internet Consortium) [Con19].

1.2 Motivation

As one of the problems presented in FC management, computing task offloading is the main focus of this work. Each of the aforementioned atomic computing tasks can be seen as a module in a single application. The more an application is divided into smaller modules, the greater the opportunity for distributed computing. However, it is necessary to manage it and associate the respective modules with a computing device, i.e. placement of a task in either an Edge, Fog, or Cloud node.

The traditional optimization techniques, often reaching optimal placement policies, lack in scalability in terms of computing power and require previous knowledge. On the other hand, the machine learning methods allow for a more flexible and model free approach, which can manage to converge into an optimal policy as well. This way it is possible to translate from an offline to an online optimization problem.

Reinforcement Learning (RL) comes as a reasonable response to the problem at hand. For example, Baek et al. propose the use of a RL technique in order to manage the task offloading [BKG+19]. The algorithm constraint can be the size of the Q-table. Depending on the scale of the system, a Deep Reinforcement Learning (DRL) may come as a suitable replacement for the previous RL technique, and that is the motivation behind this work.

Further study into DRL for FC choreography showed no consistent standardization in terms of simulators and baseline comparisons. A subsequent work from the authors, where DRL is used for choreography [BK20], was taken as a baseline and as a starting point to create standardized research on DRL for FC choreography.

1.3 Objectives

In this work, the main goal is to create an optimized FC choreography algorithm based on the previous work done in the area, specifically by Baek et al. [BK20]. Improved DRL techniques are used to make a scalable and efficient choreography in this ubiquitous and heterogeneous environment.

To do so, it is firstly necessary to analyze the FC environment set up, to analyze the state-of-the-art in RL, and study the related works in FC choreography featuring RL to identify opportunities for contribution. Afterwards, an open-source environment based on open-source DRL tools is created. Then an algorithm is implemented, where the improvement in training convergence speed is thought to provide a more useful algorithm for FC choreography.

1.4 Outline

The outline of this report is as follows: Chapter 2 starts by revising the previous work done on RL, DRL, and FC architecture and resource management; Chapter 3 describes the first steps of the work, the tools, basic algorithms and the simulated FC environment; Chapter 4 explains the implementation of the used DRL algorithms; Chapter 5 contains the results in various test cases; Chapter 6 concludes the work and proposes further research.

2

State-of-the-art

Contents

The objective of this work is to create an optimized FC choreography algorithm based on the previous work done in the area, specifically by Baek et al. [BK20]. Improved DRL techniques are used to make a scalable and efficient choreography in this ubiquitous and heterogeneous environment.

Therefore, the previous work review starts by a review of FC concepts, then an introduction to the RL techniques, passing onto DRL, and ending with a revision of the state-of-the-art in both FC resource management and direct applications of it.

2.1 Fog Computing

FC came to tackle the problems of real-time response and bandwidth limitation in CC with the growth of IoT. It was defined by the OpenFog Consortium [Con17], now part of the Industrial Internet Consortium, as:

"A horizontal, system-level architecture that distributes computing, storage, control and networking functions closer to the users along a cloud-to-thing continuum."

According to the OpenFog Consortium, FC comes into play as an extension of the CC continuum, in order to permit the environment from IoT to Cloud to S.C.A.L.E.:

- Added **S**ecurity;
- Improved **C**ognition;
- Greater **A**gility;
- Reduced **L**atency;
- Overall optimized **E**fficiency.

The challenge of FC is to manage the tasks with multi-objective optimization in a heterogeneous, agile (mobility, creation and deletion of nodes), and constrained environment. The management of FC is focused on reducing task execution delay with the overall process constrained by the physical capability of the processing nodes, bandwidth limits, and schedule necessities in tasks.

2.1.1 Entities of the Fog

The physical layout of a Fog architecture can be seen as a continuum between end devices and cloud.

As illustrated in Figure 2.1 on the right side, the Fog is made to manage the payoff between computational needs and task execution delay. To do so, it forms a link between the end devices and the cloud, making use of the edge devices, cloudlets and other intermediate devices for additional computation and storage.

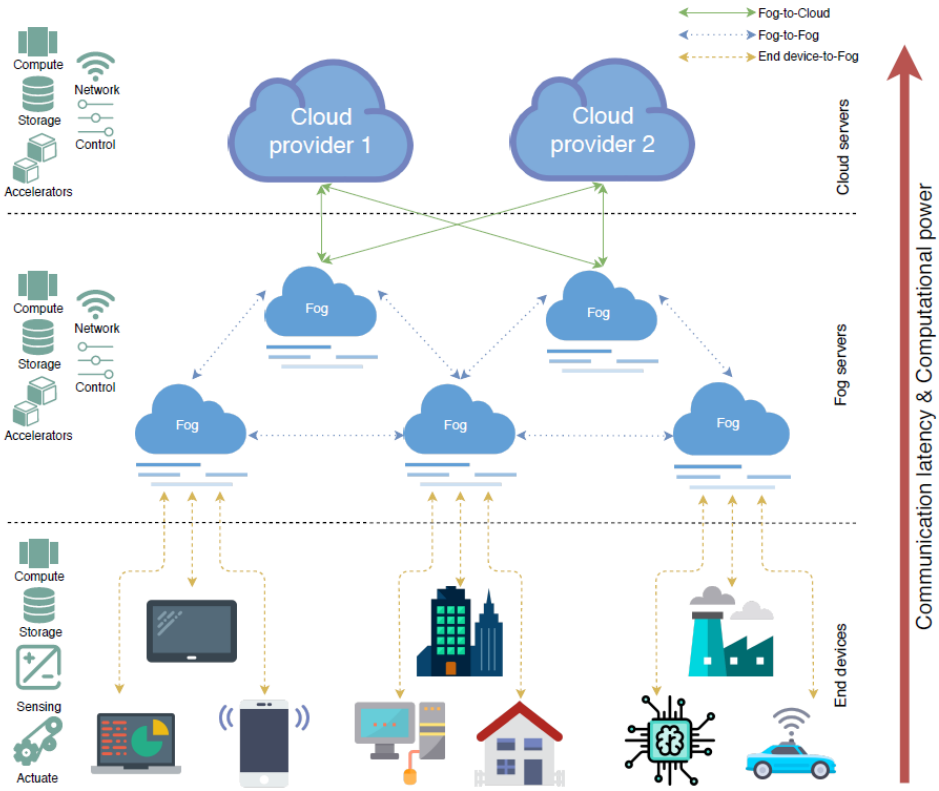

Figure 2.1: Fog physical layout, taken from [Vie19].

The listing of the main entities that come into play can be reduced to the following: the cloud, with virtually infinite storage and computing power; fog nodes, organized either physically or by a Software Defined Network (SDN), constituted by edge routers, cloudlets, among others; and finally the end devices, which can also be referred as User Equipment (UE), composed by sensors, actuators, and other user related devices.

2.1.2 Resource management - Orchestration vs Choreography

The FC resource management is still divided between offline/centralized and online/decentralized approaches [YFN+18]. Orchestration, the centralized approach, considers that there is an entity that holds the whole knowledge of the physical system, including information about the tasks to be deployed in the Fog, doing all decision making and optimization with an offline method before task allocation. Choreography, on the other hand, is when no entity holds the entirety of information about the system, but instead nodes cooperate with each other in a decentralized fashion to manage the tasks deployed in the Fog, adapting to needs in an online fashion. Another possibility is a hybrid approach, where nodes can make their own decisions but may share some information, for example in [BK20].

2.1.3 OpenFog Reference Architecture

The OpenFog Consortium, created in a coalition between industry and academia, proposed the Reference Architecture (RA) to enable the future of IoT. It is their purpose to help innovation with an open architectural framework, guiding business leaders, software developers, silicon architects, and system designers to create and maintain a Fog enabled world [Con17]. The RA stands upon eight pillars: Security, Scalability, Openness, Autonomy, Programmability, Agility, Hierarchy, and RAS (Reliability, Availability, and Serviceability). Standing on those pillars, the OpenFog RA can be seen by multiple technical viewpoints, offering views on different structural aspects, from different professional perspectives. Further study of the OpenFog RA is possible in [Con17] but out of the scope of this work.

2.1.4 Use Cases

As mentioned before, FC comes to tackle some problems found in using only CC. The use cases of FC are when the bottle neck of the problem lies near the client rather than the cloud. Three relevant examples of use cases are:

i **Vehicular**: Vehicular FC is one of the most recurrent examples, when smart cars generate terabytes of information that simply cannot be sent to a cloud based solution without some extent of treatment due to bandwidth and real-time restrictions. Multiple approaches can be considered, such as vehicle to vehicle information sharing, vehicle to infrastructure, among others. All of this in order to improve efficiency and safety on the road.

ii **Visual Security and Surveillance**: Cameras nowadays produce a vast amount of data, reaching terabytes daily when in higher definition, making it unreasonable to transport it to the cloud due to bandwidth and cost constraints. FC comes as a paradigm that allows pre-processing in a node closer to the sensor, reducing the impact of the data traveling onto the cloud, managing and discarding when no anomaly is detected, or just adding a privacy layer in a latency-sensitive scenario.

iii **Smart Cities**: this is the perfect example for an FC network to operate. It improves the economical efficiency of almost every aspect, contributes to extra layers of security since smart cities will be dealing with sensitive data, and helps connectivity when deployed with 5G technologies.

2.1.5 Multi-armed bandit in the Fog

In an FC environment there is a limited number of resources to be allocated to a limited number of tasks by a controller with the objective of minimizing costs with some constraints. The tasks are often periodical, if produced by sensors and alike, and if the controller does not have the whole information

of the system, the cost function is dependent on the action but not solely dependent on it. Translating the cost to a reward, this is the multi-armed bandit problem. The most prominent method to solve a multi-armed bandit problem is through RL.

2.2 Reinforcement Learning

RL can succinctly be defined as a method to solve a closed-loop problem, where the agent, in order to reach a goal, adjusts its future actions without direct instructions, based on a numeric reward from interacting with the environment on its previous actions, that can influence a distant future [SB18]. Therefore, an agent must be able to sense the state, change it with actions, and have a defined goal. The combination of these three requirements is called an RL method. An illustrative explanation can be seen in Figure 2.2.

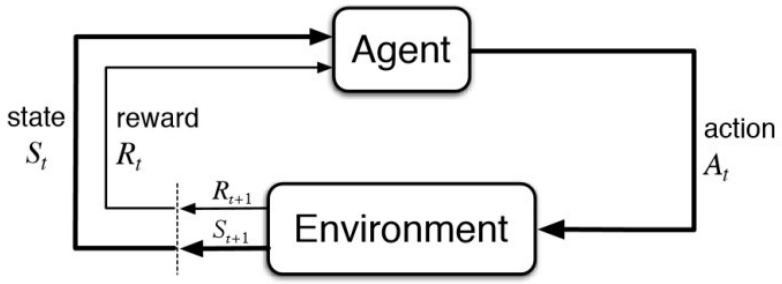

Figure 2.2: Reinforcement learning diagram, taken from [Bha18].

Unlike other classical machine learning methods, such as unsupervised and supervised learning, a key feature of RL is the duality of exploration and exploitation. The agent has to balance between exploiting previously successful actions or exploring a possibly better path. However, RL's simplicity remains clear when compared to other machine learning techniques in terms of mathematical complexity.

Besides the agent and the environment, there are four elements that we must consider in RL: a policy, a reward signal, a value function, and an environment model.

A policy defines an action for the agent to do at a given state. It can be either a look-up table or another way of representing a state-action relation. A reward signal is a representation of the goal and the agent's sole purpose is to maximize the reward in the long run. Unlike the instant reward that is defined at each action, a value function translates a long term gain from the current state. The last element is the model of the environment. It allows the agent to make inferences based on the model or simulations on how the system will behave.

A model free algorithm is also not uncommon in RL, using the environment as a test case. Another element that might not be used is the value function. For example, in policy gradient methods only the

reward is used. However, the value function can often be used to improve the gradient estimates.

2.2.1 Markov Decision Processes

Sutton and Barto briefly define a Markov Decision Process (MDP) as an RL task that satisfies the Markov Property [SB18]. With finite states and actions, the problem can be called a finite MDP.

The Markov Property is stated as follows: the state signal retains past relevant information for the next action to be based solely on the current state [SB18]. For RL, even when having a non-Markov state signal, it is better to approximate to a Markov state, in which decisions and values are better assumed as a function of the current state alone. The dynamics can be defined solely by the current state's probability distribution, formally defined as:

$$p(s', r|s, a) = \Pr\{R_{t+1} = r, S_{t+1} = s'|S_t, A_t\}, \tag{2.1}$$

for all rewards, states and actions. Where R_t, S_t, and A_t represent the reward, the state and the action, respectively, at the time t. Given the dynamics in the Equation 2.1, the rest of the computations about the environment are derived.

2.2.2 Value based methods

Value based methods use a proxy to decision making. There is a value function to be learned, and based on that value function, an action can be chosen.

A well established value based method is Q-learning. The main advantage of Q-learning is being a Temporal-Difference (TD) method, i.e. it does not need a model of the environment. Comparing to Monte Carlo methods, Q-learning implements an online increment method, improving at each time step. It can be formally defined as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \tag{2.2}$$

This allows state-action pair values to be updated at each step, independently of the policy being followed. However, the chosen path is dependent on the policy. The latter usually seeks to maximize the value function while taking into account the explore and exploit paradigm, for example, using an ϵ-greedy action selection [SB18].

2.2.3 Policy gradient methods

Policy gradient methods, on the other hand, improve the actions taken on the environment rather than a proxy value. They use a set of parameters θ (e.g. coefficients from a polynomial or weights from an

Neural Networks (NN)) that update iteratively based on an estimation of the expected episode return $\mathbb{E}[R_t]$. Often this iteration is done by a gradient ascent on the expected return, i.e. to maximize it. This has an advantage over value based methods since it is an online policy update, able to handle continuous states and actions, and deals better with state uncertainty.

However, the used estimator has a major impact on the bias-variance duality of the algorithm. When training with a naive policy gradient, using the instant reward as an estimator, the variance between actions in a state might be low, but the bias to the real expected return is huge. On the other hand, when training with a Monte Carlo target, i.e. the expected return, the bias is by definition zero, but the variance might be huge, since an action change in the beginning of the episode might have totally different outcomes in the end. Here, the scenarios could lead to an under-fit to the episode, when training with a high bias, or an over-fit to the episode, when training with a high variance.

This problem is often solved with a separate function for the estimator, i.e. a value function estimator. The joint policy-value methods are called actor-critic methods.

Since a policy is created with a parameters base, the emergence of NN created a natural transition to DRL, with the θ as the weights and biases of a network.

2.2.4 Actor-critic methods

Actor-critic methods are TD methods with separate structures for value estimation and policy. They come as a natural extension of policy gradient methods, where the value function is estimated in bootstrap to reduce the variance [Wil92].

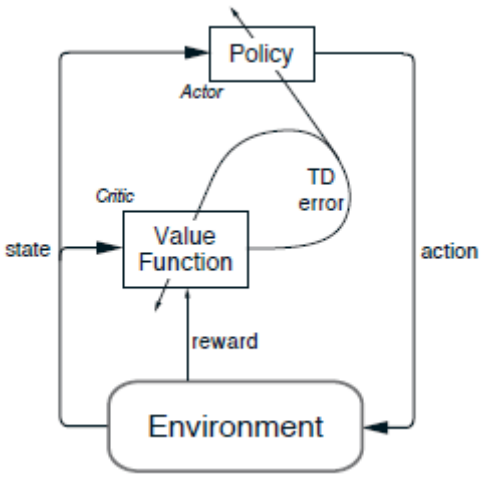

Figure 2.3: Actor-Critic methods representation in [SB18]

As illustrated in Figure 2.3, the actor, which is a policy mapped by a set of parameters θ, chooses the actions for the environment, the critic, which is a value function, uses a TD method to update both

itself and the actor based on the environment state and reward returns.

The critic might be an estimator for the value function $V(s)$, the quality of an action $Q(s, a)$ or even the advantage of an action $A(s, a)$.

However, when using a TD error as update, there is the problem of bias-variance duality in the choice of the target. Similar to the scenarios discussed in Section 2.2.3, if using a small trajectory for the update there is a high bias to the expected reward, and there could be an under-fit of the policy to the trajectory. On the other hand, with a long trajectory the bias to the expected reward is smaller but the variance is larger, so there could be an over-fit of the policy to the trajectory. The obvious choice is the neutral ground, using a n-step TD target bootstrap to prevent both over-fit or under-fit in the training steps.

2.3 Deep Reinforcement Learning

Goodfellow et al. succinctly define Deep Learning as a way of solving a complex representation problem expressed in terms of other simpler representations [GBC17]. An essential part of Deep Learning is the multi-layered perceptron, a map between a set of input and output values stored as weights, in neuron inspired nodes, able to generalize procedures in a relatively small memory space.

DRL seeks to bring the best of the two worlds together: the adaptability of the agent in the environment of RL, and the abstraction from raw data combined with information compression of deep learning. Some algorithms were proposed in order to combine these two concepts. As in RL, they are divided into value based methods, policy based methods and actor-critic.

The main problem of DRL is to find the convergence of the non-linear approximator in order to maximize the total reward. This is done by placing the objective as a target in the loss of an NN, with methods that will be presented in the following sections.

2.3.1 Deep Q-Network

Deep Q-Network (DQN) was proposed by Mnih et al. to be a generic agent that learns through sensory input, until reaching an optimal playing style based solely on pixels and game score [MKS+15].

Previous solutions, presented to enable the use of a non-linear approximator in Q-learning, have a fault that DQN seeks to correct: a need for an extensive training including previous episodes. This way, DQN achieves a reasonable efficiency in training a Convolutional Neural Network (CNN).

The problem can be formalized as a finite MDP. The agent has a legal set of actions to take which modify the state. The state is defined as a sequence of screen observations and taken actions, since the current screen would not translate any temporal dependence on its own.

Most value based RL algorithms estimate the action-value function by using the Bellman equation as

an iterative update:

$$Q_{i+1}(s,a) = \mathbb{E}_{s'}\Big[r + \gamma \max_{a'} Q_i(s',a')\Big|s,a\Big]. \tag{2.3}$$

However this is often impractical, since the game cannot be played "backwards". In the DQN algorithm, the authors utilize an approximation for the optimal value. Since the Q-function is written by the weights of the NN, the weights of the previous network update are used for the required approximation:

$$y_i = r + \gamma \max_{a'} Q(s',a';\theta_i^-). \tag{2.4}$$

The training is made using a loss function that changes at each iteration i,

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\Big[(y_i - Q(s,a;\theta_i))^2\Big] + \mathbb{E}_{s,a,r}\Big[V_{s'}[y]\Big], \tag{2.5}$$

where the targets depend on the networks' weights. This is characteristic of RL, where the targets are updated at each step based on a reward. Differentiating in order to the weights, the following gradient was used for the training with RMSprop [Hin12]:

$$\nabla L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\Big[\Big(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\Big)\nabla Q(s,a;\theta_i)\Big]. \tag{2.6}$$

The proposed algorithm modifies online Q-learning to make it suitable for training NN. First, experience replay was used, with the minibatches randomly selected to avoid the high correlation between consecutive samples, add data efficiency, and to learn off-policy. Secondly, the authors found a way to delay the time between an update to Q and an update to the target y, improving stability and making divergence and oscillation less likely, by using a fixed Q' (cloned from Q every n steps) to generate the targets for every n update steps in the Q-function.

During experiments, in 29 out of 49 games, the scores were above human-level, peaking with video pinball where the screen represents all the knowledge about the game. A performance below human-level was obtained in games where the rewards are more exploratory based, for example in Montezuma's revenge.

Overall, the novel algorithm inspired in a human-like thought, managed to greatly outperform many of the previously proposed algorithms, with two main advantages: very little prior knowledge and a model free base.

Later improvements on this value based algorithm were made with [HGS15], where the authors proposed a different network to estimate the target, called double Q-learning. And also in [WSH$^+$16], where a dueling network output would separate the estimation of the advantage and the value state functions, $A(s,a)$ and $V(s)$, in order for the algorithm to learn the $V(s)$ faster and perform better estimates on the subsequent Q-values.

2.3.2 Advantage Actor-Critic

Proposed in [MBM+16], an Asynchronous Advantage Actor-Critic (A3C) method was presented, where the advantage function is used as the value estimate for the critic in an asynchronous way to allow multiple parallel runs. Later, adapted by the authors, a synchronous approach Advantage Actor-Critic (A2C) was also presented referencing that the asynchronous characteristic of A3C did not give any performance improvement besides distribution of the computational power required, as later demonstrated by its authors.

An A2C uses the advantage as means to update a policy, with the objective to increase the log-likelihood of an action with a positive advantage. It is done so with the minimization of the loss,

$$L(\theta) = \mathbb{E}_t \left[-log\pi(a_t|s_t;\theta_t)(R_t - b_t(s_t)) \right], \tag{2.7}$$

where the expected return R_t is an estimate of the quality function $Q^\pi(a_t, s_t)$ and the baseline $b_t(s_t)$ is calculated by the critic as the value function $V^\pi(s_t)$. Making it so that the update is based on the advantage $A(s,a) = Q(s,a) - V(s)$.

The authors run the A2C for the same test set as in DQN, demonstrating that it finds a faster convergence for the algorithm and often offers an increased reward.

2.3.3 Proximal Policy Optimization

The Proximal Policy Optimization (PPO) algorithm is an improvement on policy updates, to ensure that the step is done in order to improve the policy allowing multiple steps of optimization with one trajectory [SWD+17].

It is done so with a surrogate objective $r_t(\theta)$ rather than the policy itself,

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \tag{2.8}$$

where θ_{old} refers to the policy parameters that created the trajectory, resulting in $r(\theta_{old}) = 1$.

The proposed loss function looks to clip the update to a pessimistic update, in order to avoid an excessively large policy update, allowing for a lower bound approach to a function maximum. It is translated as,

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right], \tag{2.9}$$

where $\hat{\mathbb{E}}_t[...]$ denotes the empirical average over a finite batch of samples, $\hat{A}_t$ an estimation of the advantage at time step t, and the probability ratio $r_t(\theta)$ is clipped with a PPO hyperparameter ϵ.

In experiments, with an A2C architecture, PPO over-performed other algorithms, currently being a lower computational demand state-of-the-art for DRL.

2.4 Related Work

Studies of machine learning or traditional optimization techniques have been made in hypothetical scenarios and in simulation environments. The traditional optimization techniques, often reaching an optimal offloading policy, require previous knowledge and lack in scalability in terms of computing power, since they must compute an optimized strategy taking into account the entire system. On the other hand, the machine learning methods allow for a more flexible and model free approach, that can manage to converge into an optimal policy as well. However, the latter can find limitations in memory and, mostly, training time. It is an intricate balance to obtain the best policy in a reasonable time.

2.4.1 Reinforcement learning in Fog Computing

Traditional RL techniques, as Q-learning, are a valid approach to FC resource management. In the work [BKG+19], a Q-learning algorithm is presented showing a reasonable working resource management. However, the traditional RL lacks in scaling, since, in order to reach a convergence criteria, the states are required to be often visited and updated. With a large enough complexity, the number of states to visit is just too much to do with within a reasonable time as referenced later in a second work of this authors [BK20].

2.4.2 Deep reinforcement learning in Fog Computing

From RL to DRL, the state-of-the-art in FC resource management comes to offer a more scalable solution. However, the main problem is training. It is necessary to find a objective for the non-linear approximator that the NN is, i.e. the loss of a prediction, to maximize the system reward that then translates into a better system performance.

In the work presented in [BK20], a variant of a DQN is presented as the optimal resource manager. This type of algorithm uses the Bellman equation as the loss function. To solve the non-linear approximator convergence problem experience replay and a target network are used as proposed in [MKS+15].

Furthermore, the proposed scheduling and offloading algorithm is in a choreography fashion but with a common reward, i.e., each node has its own Q-value learner and makes actions on its own in order to maximize a common calculated reward. This would allow a low load on the communication between nodes necessary for the control panel, leaving most of the channels to actual offload work. However, this creates a problem of a cooperative game where each action might affect the stability of the others decisions. To solve the convergence problem in a maximized common reward, as proposed in [KK02], the authors stated that it is necessary to:

- Decay the learning rate α over time such that $\sum^t \alpha = \infty$ and $\sum^t \alpha^2 < \infty$.

- Each node visits every action infinitely often.
- The probability $\mathbb{P}_i^t(x)$ that node i selects action x is non zero for every action.
- The exploration strategy is exploitative such that in the limit to infinity the probability of a non-optimal action tends to zero.

With this and an architecture of CNN, Recurrent Neural Network (RNN) and dense layers, the authors managed to do a non-linear approximator to manage actions at each node.

However, after calculations, there is the hypothesis that the high performance of the proposed algorithm, compared with the baselines, is due to a weak performance of the latter. With a more appropriate choice baseline algorithms, even without the use of machine learning, the comparison could yield different results. Nevertheless, the learning capability of such algorithms is still a reason to press further and improve the work presented before.

2.4.3 Applications

Some applications are already being studied using the state-of-the-art for FC resource management, namely the use cases presented in Section 2.1.4.

i **Vehicular FC** is one of the hot topics. Internet of Vehicles (IoV) promises a massive reduction in accidents and an increase in overall traffic efficiency: reducing the time of transportation, energy consumption, among other benefits.

FC comes as a reasonable paradigm in order to deal with the massive data creation by the smart vehicles, allowing comprehensive data offloading and faster time response. Investigation seeks to improve DRL resource management techniques in the IoV scenario, in training [LL19], in a dense traffic environment [XLH+19], and even in energy-efficient solution [DWG+19].

ii **Healthcare** is one of the areas that technology might make the greatest impact in our lives. With the increase of sensors in quality and reduced size, combined with optimization and expert knowledge, healthcare can be personalized to increase life quality and expectancy. However, since sensor data is worthless if unused, Kraemer et al. discuss different ways that FC can complement existing IoT related with the healthcare industry [KBTP17]. Proposed themes for contributions are partitioned into different use case examples, but can be grouped in data collection, data analysis, critical analysis, critical control, and context management. For example, a type of data analysis would be speech analysis for Parkinson disease, where the forwarded data would not be raw but rather the result of local analysis [MDM+16]. A critical analysis could be electrocardiogram (ECG) monitoring, giving real-time notifications of the patient status, possibly enriching it with location [GJR+15].

There are many practical examples where FC would greatly contribute for the improvement of healthcare, and it is an area to be invested after proper optimization of FC resource management.

3

Methods, tools and Model Definition

Contents

The initial course of action was to implement the described FC environment and the baselines, and from then work through RL and DRL algorithms as resource management controllers. To implement the desired FC environment, the used tools were chosen in accordance with the future DRL implementation. This chapter explains the logical steps that were taken.

3.1 Methods and tools

In this work, focused on the algorithms rather than the environment, established deep learning tools have to be considered. The most used is Tensorflow 2 [AAB+15]. With open-source and extensive peer review, it is rated as one of the best libraries since the joining of Tensorflow and Keras. Pytorch [PGM+19] also has to be considered, known as a more pythonic approach to machine learning, and performs as well as Tensorflow. However, if working in a more Object Oriented Programming (OOP), Deep Learning for Java (DL4J) [Tea16] is also an option created by Java developers in response to the Python machine learning popularity rise, since it also uses tensors and has the option to train with GPU.

It is also necessary to define a tool for simulation of the resource management algorithm. There are a few options: OMNeT++ [VH08], SimuLTE [VSN14], ns-3 [Con], iFogSim [GDGB16], among others. OMNeT++ is a modular network simulator, and SimuLTE was built on top, providing tools for complex system-level performance evaluation of LTE networks. Another separate software is ns-3, a simulator for network discrete events, built in C++ with Python APIs. The last one, FC specific, is named iFogSim, and it was built using the established CloudSim [CRB+11] framework for handling events between Fog entities.

However, all these tools lack an inherent RL approach, i.e., an online optimization approach. Due to the lack of a standardized tool for RL resource management in an FC environment, another simulation tool has to be created. Rising as an example for the DRL community, OpenAI provides tools for standardized RL environments, so the environment will be set as a subclass of an OpenAI Gym [BCP+16].

Given this information, the tool decided for the algorithm is Tensorflow 2 which is in accordance with the programming language used in an OpenAI Gym.

In terms of algorithms, the logical steps would be to first re-implement given baselines and work through to the RL and DRL approach to FC resource management.

3.2 Simulation environment

An FC environment has heterogeneous characteristics: each device and connection may be different in its specifications. Entities can be divided into three layers: Cloud devices, Fog devices, and end devices. Each of these devices might be grouped or not, from an SDN point of view, and considered a node, i.e.

Cloud node, Fog node, and End node. Each node is somehow connected to the network through a network interface, translated to physical form in either optical cable, coaxial cable, or cellular network, among other possibilities, including multi-hop routing segments.

The work in [BK20] will be used as reference for the FC environment modelling.

The FC environment is set in an SDN point of view, where the data plane processes and forwards task while the control plane is in charge of decision-making. Furthermore, it is split in logical slices, which provides an extra level of abstraction allowing priority allocation or concise physical resource division per layer, among other possible operations in a sliced environment. This further complexifies the problem to an joint offload and scheduling optimization.

This problem is defined as a problem of choreography, where each node makes its own decisions. However, as nodes share their local rewards, a joint reward is used to improve the system as a whole, which includes a hint of orchestration.

For the whole Section 3.2, notation will follow the Table 3.1.

3.2.1 Environment description

Each node is characterized by its resources $U_i = (U_i^c, U_i^m)$, CPU and memory respectively, transmission resources bandwidth BW_i and transmission power P_i. From the control layer point of view, the node buffer is divided in K slices, each of this slice partitions of the buffer is seen as a buffer itself with a maximum queue size of $\bar{b}_{i,k}$.

Different task types are assigned at different slices. In each slice a task is characterized by its packet size T_k, a delay constraint D_k^{max}, and resource requirements for processing the task $L_k = (L_k^c, L_k^m)$, CPU processing density and necessary memory respectively.

It is assumed that each allocation unit, η_i^c and η_i^m respectively for CPU and memory, is indivisible. Furthermore, it is assumed that a CPU allocation unit is a single computing core with frequency η_i^c. Another assumption is that each task of size T_k and computing density L_k^c is sequential, i.e. it can not be split to compute in parallel cores. With those assumptions, each task in slice k is assigned one core and the necessary memory when being processed:

$$L_k = \left(\eta_i^c, \left\lceil \frac{L_k^m}{\eta_i^m} \right\rceil\right) \tag{3.1}$$

With the assigned resources stated in Equation 3.1, a task from slice k processing in node i has a processing delay calculated with the following expression:

$$D_{i,k}^p = \frac{T_k \cdot L_k^c}{\eta_i^c}. \tag{3.2}$$

The communication is modelled in a full duplex fashion, with each node splitting its assigned band-

Table 3.1: Environment description notation

Notation	Unit	Description
I		The number of Fog nodes
K		The number of logical slices in the Fog
(β_1, β_2)		The path loss constant and exponent
N_0	dBm/Hz	Thermal noise density
P_i	dBm	Transmission power of fog node i
BW_i	MHz	Transmission bandwidth attributed to fog node i
U_i^c	GHz	The total amount of CPU resources at fog node i
U_i^m	MB	The total amount of memory resources at fog node i
η_i^c	GHz	The allocation unit of CPU resources at fog node i
η_i^m	MB	The allocation unit of memory resources at fog node i
r_i^c		The number of available units of CPU resources at fog node i
r_i^m		The number of available units of memory resources at fog node i
T_k	bit	Packet size of a task in slice k
D_k^{max}	ms	Maximum delay budget of a task in slice k
L_k^c	cycle/bit	CPU processing density demanded for a task in slice k
L_k^m	MB	Memory size demanded demanded for a task in slice k
$a_{i,k}$		A Boolean variable that indicates whether or not a task from slice k arrived at fog node i during a certain time step
$\bar{b}_{i,k}$		The maximum number of tasks from slice k in the buffer of fog node i
$b_{i,k}$		The number of tasks from slice k in the buffer of fog node i limited by $\bar{b}_{i,k}$
$b_{i,k}^e$		The number of tasks from slice k in the buffer of node i which have resources allocated to its execution limited by $b_{i,k}$
$f_{i,k}$		The offloading decision made by fog node i to where the task of slice k will be processed
$w_{i,k}$		The resource allocation decision done by fog node i that indicates how many tasks of slice k will have resources allocated to them
$\mu_{i,k}$		The service rate of the node i for slice k, calculated as the number of tasks leaving the buffer $b_{i,k}$ per time step
$\lambda_{i,k}$		The arrival rate at node i for slice k

width for the number of concurrent tasks being offloaded to another node. Another assumption is that it is a far field approach. To calculate the bit rate between node i and node j, offloading n tasks, the Shannon-Hartley theorem is used [Sha49]:

$$v_{i,j,n} = BW_{i,n} \cdot log_2 \left(1 + \frac{\beta_1 {d_{i,j}}^{-\beta_2} \cdot P_i}{BW_{i,n} \cdot N_0}\right), \tag{3.3}$$

where $d_{i,j}$ is the distance between the two nodes, β_1 and β_2 are the path loss constant and exponent respectively, $BW_{i,n}$ is the bandwidth of node i evenly divided by the n concurrent tasks, P_i is the transmission power and N_0 is the thermal noise density.

With Equation 3.3, a transmission delay from node i to node j, with $i \neq j$, can be calculated as:

$$D^t_{i,j,k,n} = \frac{T_k}{v_{i,j,n}}, \tag{3.4}$$

with T_k as the packet size of a task in slice k.

Another delay that has to be taken into account for computation of the total delay is the waiting time in the queue, given by the following expression:

$$D^q_{i,k} = \frac{b_{i,k}}{\mu_{i,k}}, \tag{3.5}$$

with $\mu_{i,k}$ as the service rate of node i for the slice k.

The total delay of a task from slice k processing in the Fog can be computed, including transmission delay ($D^t_{i,j,k,n} \neq 0$) if there is an offloading ($i \neq j$), as the sum of all delays as follows:

$$D_{i,j,k,n} = D^p_{i,k} + D^q_{i,k} + D^t_{i,j,k,n}. \tag{3.6}$$

Limited by their delay constraints D^{max}_k, when a task reaches its maximum delay budget it is discarded, since it has lost its data value or necessity to be processed and considered unsuccessful. This is only done when a CPU unit and the necessary memory units can be assigned to the task in order to verify the delay constraint. Another limitation imposed is that, when a task arrives from the end device, the decision to offload or not the task is done in that time step and the task cannot be offloaded to a third node.

The time step is considered small enough for the event of a task arriving at a node i from a slice k to be unique. So, at each time step t, the task arrives following a Bernoulli distribution with the parameter $\lambda_{i,k}$.

3.2.2 Discrete Event Simulator

To simulate the environment described in Section 3.2.1, a Discrete Event Simulator is used. A unique event queue sorts posted events and runs each event that modifies the representation of the physical system changing its state. Eight events were implemented as a subclass of a generic sortable event of the event queue:

- **Set Arrivals** - is an event that triggers a Task arrival from an end device based on a Bernoulli distribution with the parameter $\lambda_{i,k}$ for each node i and slice k. This event is called once in the beginning of the simulation and recursively posts itself in the queue for the next time step, it simulates the end devices requests.
- **Task Arrival (i, k, task)** - is an event that receives a task from slice k on node i. This event is called either from an Offload task event or a Set arrivals event.
- **Start Processing (i, k, w)** - is an event that sets $w_{i,k}$ tasks from slice k to process in node i, occupying necessary resources L_k as described in Equation 3.1; it posts a Discard task event if any attempted task exceeded its delay budget D_k^{max} or the Stop processing event if the task completes its processing within time constraint $D_{i,j,k,n} < D_k^{max}$. This event is posted from the control plane, made by the decision making algorithm.
- **Stop Processing (i, k, task)** - is an event that stops the processing of a specific task from the slice k in the node i, freeing all the taken resources L_k. This event is posted by the Start processing event.
- **Discard Task (i, k, task)** - is an event that discards a specific task from a slice k that attempted processing or is in the middle of processing in the node i, that has exceeded its delay budged D_k^{max}, returning all assigned resources L_k to the node pool of available resources. This event is called by the Start processing event.
- **Offload Task (i, j, k, n)** - is an event that offloads the task from a slice k that arrived in this time step in the node i to a node j, where $i \neq j$. It will only offload a recent arrival, since the decision of where a task will be processed, either locally or remotely, is done when the task arrives. To offload a task, this event splits the currently available bandwidth with the concurrent offloads n from other slices, calculates an arriving time and triggers a Task arrival event for each offloaded task. It also triggers a Start transmitting event for each offloaded task that will use the portion of the bandwidth to send the task. This event is called from the control plane, made by the decision making algorithm.
- **Start Transmitting (i)** - is an event that starts a transmission of a specific task in node i, occupying the given bandwidth $BW_{i,n}$, and posting the Stop transmitting event when the transmission ends. This event is posted by the Offload task event.
- **Stop Transmitting (i)** - is an event that stops a transmission in node i, freeing the previously

occupied bandwidth $BW_{i,n}$ for a specific task. This event is posted by the Start transmitting event.

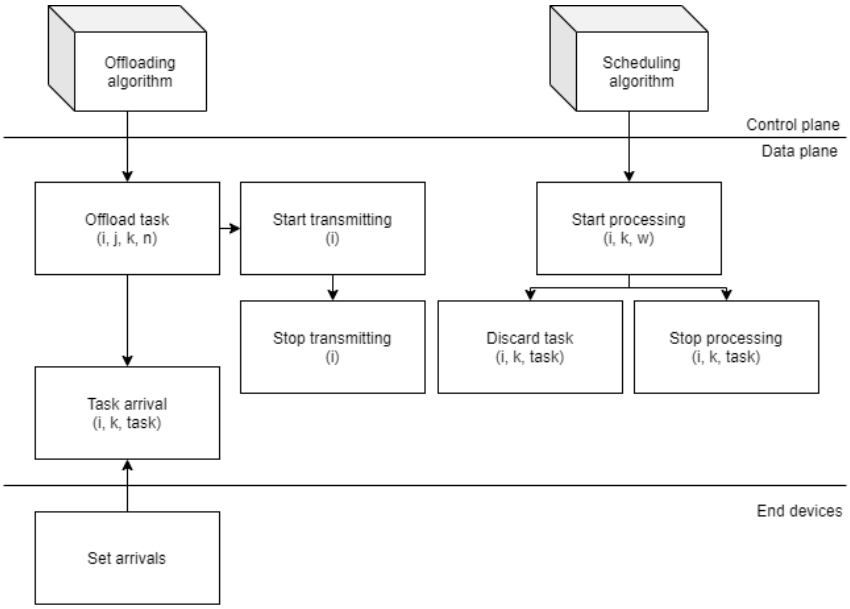

Figure 3.1: Flow structure of the discrete events

An illustration of the events flow is provided in Figure 3.1.

3.2.3 OpenAI Gym - Fog_env

In order to make a FC environment with an RL approach, the OpenAI Gym base class *Env*, provided by OpenAI under the MIT Licence, is used as a super class of the implemented FC environment Fog_env. To do that it is first necessary to translate the environment to an MDP, i.e. define a state, an action, a probability of transition and a reward function. However, each node only knows its own information, so the MDP is in fact the sum of the Partially Observable Markov Decision Process (POMDP) from each node. The local observation of the node in a time step is described, similar to the one in [BK20], as follows:

$$O_i(t) = (A_i(t), B_i(t), B_i^e(t), R_i(t)) \,, \tag{3.7}$$

where $A_i(t)$ is the vector of $a_{i,k}(t)$, the boolean that indicates a task arrival from an end device, $B_i(t)$ the vector of $b_{i,k}(t)$, the number of tasks in the buffer for this time step t in each slice k, $B_i^e(t)$ the vector of $b_{i,k}^e(t)$, the number of tasks under processing in the buffer for this time step t in each slice k, and $R_i(t) = (r_i^c(t), r_i^m(t))$ are the available resource units from the resource pool in the Fog node i.

The action a node can take based on his observation is:

$$X_i(t) = (F_i(t), W_i(t)) \,, \tag{3.8}$$

where $F_i(t)$ is the vector of $f_{i,k}(t)$, the decision to where the task that arrived at the time step t from an end device in node i from slice k will be processed, and $W_i(t)$ is the vector of $w_{i,k}(t)$, the number of tasks from slice k to be assigned resources in node i. The values of $f_{i,k}(t)$ are either $f_{i,k}(t) = 0$ if no task arrived, or $f_{i,k}(t) = j$ given a task from an end device. If the decision is to offload then $i \neq j$ else the task will be processed locally if $i = j$.

However, there are limitations in the action chosen by the node at a specific time step. These limitations, which can be implemented by the environment or by any action clipping before applying them to the environment, are the following:

$$f_{i,k}(t) = 0, if a_{i,k} = 0, \forall k \in K_i, \forall i \in I, \tag{3.9}$$

$$w_{i,k}(t) \leq b_{i,k}(t) - b^e_{i,k}(t), \forall k \in K_i, \forall i \in I, \tag{3.10}$$

$$\sum_{k \in K_i} w_{i,k}(t) \leq r^c_i, \forall i \in I, \tag{3.11}$$

$$\sum_{k \in K_i} w_{i,k}(t) \cdot \left\lceil \frac{L^m_k}{\eta^m_i} \right\rceil \leq r^m_i, \forall \in I, \tag{3.12}$$

where Equation 3.9 states that the node cannot make a decision on where to process a task that did not arrive, Equation 3.10 limits the number of tasks to assign resources to the number of idle tasks and Equations 3.11 and 3.12 limit the number of requested tasks depending on the free memory.

Lastly the reward of the POMDP is decided as an average of probable completion reward for each task arriving at the different slices, as modelled in [BK20],

$$\psi_i(O_i(t), X_i(t)) = \frac{1}{K} \cdot \sum_{k \in K_i} a_{i,k}(t) \cdot \left((-1)^{\mathbb{1}(D^{max}_k \leq D_{i,k})} - \xi_k \cdot \mathbb{1}(b_{f_{i,k},k}(t + D^t_{i,k}(t)) \geq \bar{b}_{f_{i,k},k}) \right), \tag{3.13}$$

where the first term estimates whether the task will be completed successfully or it will exceed its time budget D^{max}_k, in which $D_{i,k}$ indicates the total delay estimate made by Equation 3.6, and the second term with the constant weight ξ_k, a tunable parameter, negatively impacts the reward if the task is expected to overflow the node $f_{i,k}$ assigned to process the task. The buffer size evaluation $b_{f_{i,k},k}(t + D^t_{i,k}(t)$ is done after the transmission $D^t_{i,k}(t)$ occurred for the offline optimization algorithms. An estimation $\hat{b}_{f_{i,k},k}(t + D^t_{i,k}(t))$ had to be used for online optimization algorithms at the moment of arrival t:

$$\hat{b}_{f_{i,k},k}(t + D^t_{i,k}(t)) = min(max(t + D^t_{i,k}(t) \cdot (\lambda_{i,k} - \mu_{i,k}), 0), \bar{b}_{f_{i,k},k}). \tag{3.14}$$

Given the POMDP for each node, in the Fog_env Gym the observation space is defined as a vector of all node partial observations $O(t) = (O_i(t), i \in I)$ and accordingly the action space is defined as $X(t) = (X_i(t), i \in I)$. Then the time step is also implemented in the Fog_env: it takes an action within the action space decided by the control plane as input, runs all the consequent events of an action until

$t+1$, and then returns the environment observation $O(t+1)$ and total reward $\psi = \sum_{i \in I} \psi_i(O_i(t), X_i(t))$.

It is noted that having multiple agents with a shared reward is a way to make a cooperative game, reaching an optimal Nash Equilibrium if the common reward is maximized. In this way there is no need to share large messages with the state, which might occupy the bandwidth, but rather a small reward signal that allows the cooperation to occur.

3.3 Baseline algorithms

For the comparative baselines decision to construct the action vector as described in Equation 3.8 of Section 3.2.3, the decision was split between an offloading algorithm and a scheduling algorithm.

The offloading algorithms are the following:

- Nearest Node: is a threshold offloading with nearest node selection strategy, as used in [BK20] for the baselines, where each node will only offload tasks if it is above a defined threshold Th, and will offload it to the nearest node possible, minimizing both transmission time and energy consumption to incur in less costs.
- No Offload: will not offload any task, assuming each node would be better on its own and could handle tasks assigned to itself.
- Random Offload: is a threshold offloading with random node selection, will select a random node to offload tasks after the buffer capacity is above a defined threshold Th. This baseline works to demonstrate, together with Nearest Node, the impact of choosing or not the offload target in each test case.

For the scheduling algorithm, two different baselines were used, with improvements relative to the ones described in [BK20]:

- Round Robin: where the algorithm, based on the available resource pool R_i, schedules a tasks per slice in turns by increments of the action value, $w_{i,k}$, providing a reasonably spread action vector $X_i(t)$ to the POMDP interface.
- Priority Queuing: where the algorithm, based on the available resource pool R_i, schedules tasks starting in a higher priority slice, and only moving to lower priority if all higher priority buffers have their tasks under processing, by increments of the action value $w_{i,k}$ for each slice. Here, the priority is here determined by the delay budget.

To illustrate the two mentioned scheduling algorithms, in Figure 3.2 a generic time step can be observed. Where in the same state the resource assignment path is done in a different way until all available resources are assigned.

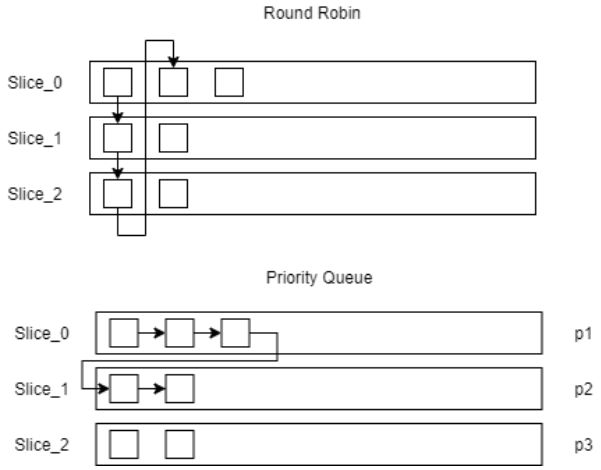

Figure 3.2: Illustration of Round Robin and Priority Queuing scheduling

The purpose is that the baseline algorithms also interact with the environment through the MDP, or more specifically for each node the POMDP. This way, in each time step, the algorithm in each node will receive a system observation $O_i(t)$ and build the action vector to apply in the next step of the environment $X_i(t)$.

3.4 Q-learning resource management

A Q-learning algorithm for FC resource management was presented by Baek et al. [BKG+19]. However, there are some issues to be associated with a Q-learning algorithm.

First, as mentioned in [BK20], the amount of action-state values is too large for a conventional Q-learning process to visit them enough times to converge within a decent time. Secondly, the algorithm presented in [BKG+19] that used an ϵ-greedy policy, had ϵ values between $[0.9, 0.7[$, which would result in a random action most of the times. Lastly, the action space in the proposed MDP was defined to force an offload, not allowing the non-offloading decision.

A Q-learning algorithm implementation was done for the resource management problem in a simplified version of the environment described in Section 3.2, as it was in [BKG+19], using only one slice and five Fog nodes. It was then promptly discarded due to the previously mentioned issues. The conclusion was that the algorithm was unsatisfactory and only performed better against very weak baselines, i.e., very simple non-machine learning algorithms outperformed the algorithm in most situations. However, a theoretical convergence for an optimal offloading with an appropriate reward function would be possible if given enough time to explore and train.

4

Deep Reinforcement Learning Orchestrators

Contents

The introduction of DRL, compared to RL, is expected to bring a scalable solution in terms of number of states that can be encompassed in the same amount of memory space, and a better generalization of actions for unvisited states, i.e., similar states will result in similar actions whenever there is an exploitation policy. The main objective would be to build an choreography framework with a fast converging algorithm, like PPO, to prove the usefulness of DRL in FC choreography.

4.1 Deep Reinforcement Learning function approximation

The DRL approaches to FC choreography, mean the RL algorithm is approximated with a Deep Neural Network (DNN). To achieve the best performance in the presented MDP in Section 3.2.3, the function approximation for the RL algorithms is as proposed in [BK20], where the one dimensional CNN provides the feature extraction from a sequence time steps and the RNN allows a state memory to make necessary correlations between inputs, seeing that the FC environment has a time correlation between states.

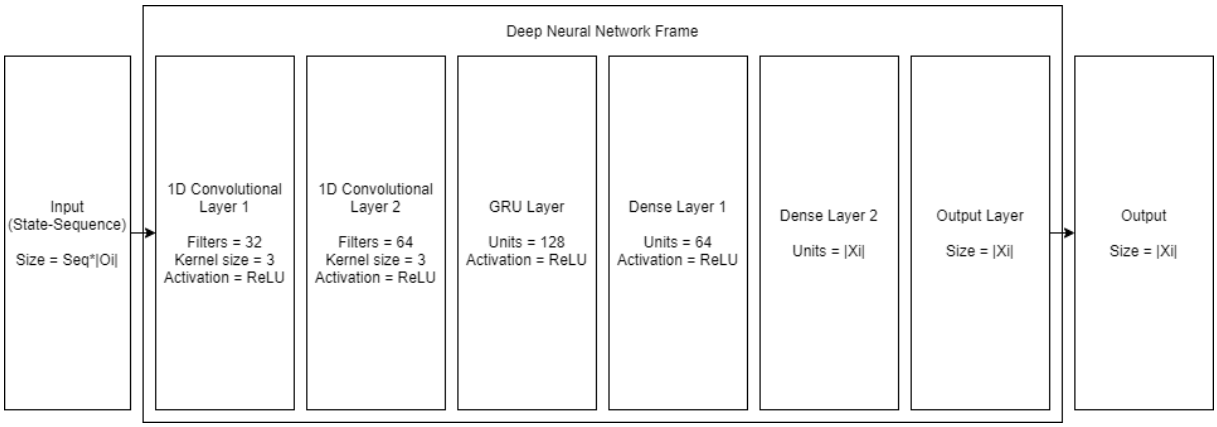

Figure 4.1: Structure of the DNN frame utilized as the non-linear approximator for the DRL algorithms

Presented in Figure 4.1, the DNN used for approximating the DRL algorithm parameters takes as input a time sequence of observations, collected during the execution of the algorithm, and returns a vector with the size of the action space. This DNN can be trained to return a vector that approximates the $Q^w(s, a)$ state-action values, in a DQN type of algorithm, or the policy $\pi_w(s)$ values, in a policy or actor-critic algorithm.

One of the challenges in DRL is to avoid divergence with this non-linear approximation, and each algorithm solves that with its methods as will be presented throughout this chapter. Another challenge is the multi-agent game theory, where it is necessary to reach a Nash Equilibrium and to maximize the common reward.

An issue to take in mind is to adapt this architecture for the defined MDP.

4.2 Proximal Policy Optimization with Advantage Actor-Critic architecture

In this section, the synchronous version of the A3C presented in [MBM+16], A2C, is implemented with a PPO loss actor step update as proposed in [SWD+17] for the MDP presented in section 3.2.3. Since it is a multi-agent scenario with a common reward, the critic is a common estimator as proposed in [LWT+20] to allow for a better convergence. In short, a PPO with an A2C architecture DRL algorithm for a multi-agent scenario.

The loss function for the actors is as mentioned in equation 2.9, section 2.3.3, with the ϵ clipping parameter set to the value of 0.2. The advantage estimator used was the Generalized Advantage Estimator as proposed in [SML+18], and used in [SWD+17] for the PPO algorithm.

The training is done with parallel short trajectories to collect information, and then apply the gradients in minibatch to the network, as popularized in [MBM+16] and [SWD+17] specially for training of RNN. A short description of the training process is presented in algorithm 4.1.

Algorithm 4.1: PPO with A2C architecture adaptation for multi-agent scenario

Create N agents and a common critic;
for *i = 1,2, ...* **do**
  for *env = 1,2, ..., E* **do**
    Run policy $\pi_{\theta old}$ in environment env for T timesteps;
    Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$;
  end
  Optimize actors surrogate L wrt θ, with L epochs and minibatch size $M \leq ET$;
  $\theta_{old} \leftarrow \theta$;
  Optimize critic surrogate L wrt w, with minibatch size M;
  $w_{old} \leftarrow w$
end

Seen that this method gives an online update of the algorithm, the reward calculation as referred in Equation 3.13, Section 3.2.3, uses an estimation of the buffer size for the second term, rather than the complete information only obtained at time $t + D^t_{i,k}(t)$.

Another issue related with the online update is, since the states depend on the actions taken, successive misplaced actions might bring the environment to a pitfall. To avoid this, it is either necessary to define a failure parameter or to restart the environment periodically during training. The second option was chosen for the presented results. Which in turn allows for the algorithm to explore more courses of actions and not to over-fit the DNN to a stable reward case.

5

Results

Contents

Throughout this chapter the results for the DRL agents presented in Chapter 4 are obtained with test cases on the environment created in Section 3.2.3. The simple baselines presented in Section 3.3 are used to characterize the specific requirements of the test cases and to offer a fair comparison with the new proposed orchestrator presented in Section 4.2.

5.1 Simulations specifications

For the particular FC environment, as described in Section 3.2.3 Chapter 3, the used base values in the tests are defined in table 5.1.

Table 5.1: Simulation configuration.

Variable	Value(s)	Variable	Value(s)
Area	100 $\times$ 100 m^2	$\lambda_{i,k}$	[0.6, 0.7, 0.8, 0.9]
I	5	(β_1, β_2)	(1, -4)
K	[1, 2, 3]	N_0	-174 dBm
P_i	20 dBm	T_k	5 Kbit
BW_i	1 MHz	CPU	[5GHz, 6GHz, 7GHz, 8GHz, 9GHz, 10GHz]
η_i^c	1 GHz	Memory	[2400MB, 4000MB, 8000MB]
η_i^m	400 MB	Th	0.8
$\bar{b}_{i,k}$	10	ξ_k	[0.2, 2.0]

Each of the I nodes is located within the plane of the defined Area, with assigned bandwidth BW_i, transmission power P_i, a CPU speed sampled from [5GHz, 6GHz, 7GHz, 8GHz, 9GHz, 10GHz], and a memory size sampled from [2400MB, 4000MB, 8000MB]. The allocation units defined in a node are η_i^c (1 GHz) and η_i^m (400 MB) for CPU and memory resources respectively. The surrounding environment has the path loss constant and exponent (β_1, β_2) and thermal noise density N_0.

With the number of slices K defined either for an only offload problem ($K = 1$) or offload and scheduling ($K = 2$ or $K = 3$), the task packet size is constant in every slice with T_k = 5 Kbit and the arrival rate $\lambda_{i,k}$ is set by default to 0.6, i.e. every time step there is a 60% chance of a task arriving at a node slice. To test increased load impact the arrival rate was increased to the values of 0.7, 0.8 and 0.9.

The baseline offloading threshold Th is 0.8, meaning only when 80% of the slice buffer is occupied will the tasks begin to be offloaded. Whereas the tunable overflow penalty ξ_k is either set to 2.0 in test cases with no time constraint or 0.2 when there is time constraint.

Different test cases have to be created to verify the algorithm performances. The cases have two levels of heterogeneity: in terms of resources and in terms of delay constraints. To formulate the test case resource demands, it is necessary to take into account that the number of tasks that can be

Table 5.2: PPO training parameters.

Parameter	Value
T	64
E	3
L	3
M	32

concurrently processed by a node is limited. Similar to Equations 3.11 and 3.12, the total number of concurrent tasks processing in each node, $W_i = \sum_{k \in K_i} b^e_{i,k}$ is constrained as follows:

$$\sum_{k \in K_i} b^e_{i,k} \leq \frac{U^c_i}{\eta^c_i}, \forall i \in I, \tag{5.1}$$

$$\sum_{k \in K_i} b^e_{i,k} \cdot \left\lceil \frac{L^m_k}{\eta^m_i} \right\rceil \leq \frac{U^m_i}{\eta^m_i}, \forall \in I. \tag{5.2}$$

Also, for a node to be able, on average, to handle tasks given to it, it is necessary that the number of tasks arriving is the same as the number of tasks being completed. However, the Fog is a resource constrained environment, where often the unbalanced load has to be balanced by a controller that does not necessarily provide a guaranteed 100% success rate.

The time constraint D^{max}_k provides the second level of heterogeneity. It can exemplify the good management of the resources, since a wrong distribution of the resources that hypothetically could complete all tasks might incur a task in missing its time constraint. Since the processors are limited to one unit per task, another constraint is posed on the delay,

$$\frac{L^c_k \cdot T_k}{\eta^c_i} \leq D^{max}_k, \tag{5.3}$$

meaning a task has, at least, to be able to be processed in the node where it is placed whenever there is an empty queue.

The test cases formulation are presented with the respective results, with the soft constraints provided by the Equations 5.1 and 5.2, and a hard constraint provided by Equation 5.3.

For the PPO algorithm, the tunable training parameters default values are presented in Table 5.2.

5.2 Performance metrics

To evaluate the performance of an algorithm in the test cases two metrics were used, and another one can be derived by the former two metrics. The first metric is the success rate, indicating the percentage of tasks assigned to the FC environment that was successfully completed and returned. The second

metric is the overflow rate, which, in turn, indicates the number of tasks that tried to be offloaded to the FC environment and were refused due to a full buffer. The implied metric is the number of tasks that fail to meet their delay total constraint D_k^{max}, that were nor successful nor refused.

To evaluate the DRL algorithm convergence, a moving average of the reward over training iterations is used, since the purpose of the algorithm is to maximize the long term reward. However, it is to be noticed that, to maximize the evaluation metrics, it is necessary to formulate a reward that takes them into account, i.e., maps the objectives of the algorithm into a numeric function $R(s, a, s') \rightarrow \mathbb{R}$. A maximized reward will not guarantee maximized metrics.

5.3 Simple test cases

First, the algorithms were tested in a simple set of test cases with only one abstract slice (K=1), reducing the action space to two values per node, one to offload and another to indicate the number of tasks to schedule in the only existing slice. With a regular setup of 5 nodes and resources enough to concurrently process all tasks in the buffer, it translates to 60 different actions on each node.

For these simple test cases, the scheduler of the baseline algorithms was set to schedule all tasks in the buffer, according to their arrival, whenever resources are available. With that in mind, the algorithms that will be compared are:

- Nearest Node (baseline);
- Random Offload (baseline);
- No Offload (baseline);
- PPO, with an A2C architecture, DRL algorithm with schedule and offload decision making.

The purpose of this section is to demonstrate the offload strategy learning capability of the proposed algorithm. In spite of the single slice, it also shows the learning capability of the algorithm to choose actions that involve scheduling, since the algorithm is not constrained in a lower bound, i.e., it can opt not to schedule a task.

5.3.1 Test environments scenarios

For the simple test cases, two specific test environments were created to demonstrate the performance of the algorithm. The developed scenarios were designed with the CPU clock speed and memory sizes of the nodes sampled from the values given in Table 5.1, and within the specified space constraint in the same table.

On the first scenario, *Env1* as illustrated in blue in Figure 5.1, the node configurations were set with a smooth spacial distribution but uneven resources, causing offloading to be necessary.

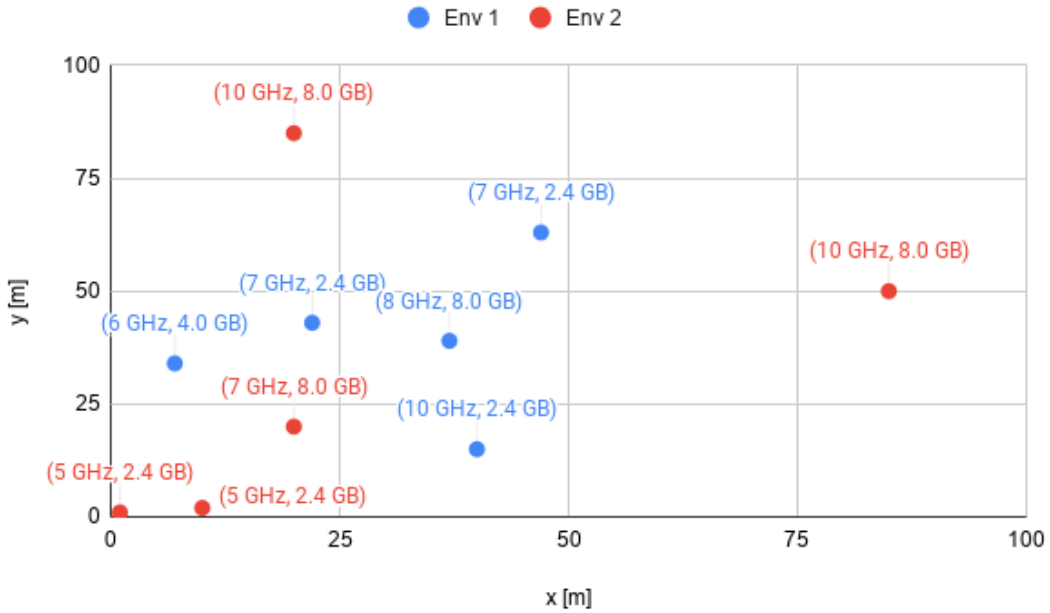

Figure 5.1: Test environments. In blue (*Env1*) a favourable for Nearest Node offload strategy and in red (*Env2*) a case with no exact optimal strategy.

On the second scenario, *Env2* as illustrated in red in Figure 5.1, the nodes were spread unevenly on the plane and computing autonomy was given to two nodes. Here, the unsuccessful tasks are mainly due to the limited resources in the nodes closer to the origin.

5.3.2 Offload case 1

The first offload case was set to experiment if the algorithm could offload to a node with enough resources to take care of a task before buffer overflow, without actually constraining the task deadline.

$$\text{Case 1:} \quad D_k^{max} = 1000 \text{ ms}, \quad L_k^c = 1200 \text{ cycles/bit}, \quad L_k^m = 800 \text{ MB}.$$

In a first trial run, using the scenario *Env1* presented in Figure 5.1, the results are presented in Table 5.3.

Table 5.3: Performance on the first trial of Offload Test Case 1.

Algorithm	Nearest Node	Random Offload	No Offload	PPO algorithm
Success rate (%)	99.8	94.0	88.5	99.5
Overflow rate (%)	0.2	6.0	11.5	0.5

The resources constraints of the first trial with this specific case lie on the memory capability to have concurrent tasks, so the Nearest Node offload strategy has inherent advantage in the pairing happening

between the memory resource rich nodes with memory resource constrained nodes in the first scenario.

The No Offload strategy caused the performance of a test episode to have a success rate of 88.5%, whereas the buffers overflowed in the other 11.5%, discarding those tasks. If a Random Offload strategy is selected instead, since there is a low punishment to the offload delay with a soft delay constraint, the success rate increased to 94.0%. On the other hand, the Nearest Node offload strategy has the highest success rate of 99.8%, given that the nearest nodes happen to pair with the resource disparities.

The PPO algorithm also managed to optimize the offloading strategy, showing a success rate of 99.5%. It is also assumed that it managed to schedule tasks successfully, choosing the right action from the scheduling action space, which is limited by the number of resources (Equations 3.11, 3.12).

Figure 5.2: PPO first trial training reward on Offload Test Case 1.

As it can be observed in Figure 5.2, the system average reward improved along the training iterations. The system reward is a sum of the node rewards, which are defined in Equation 3.13 in Section 3.2.3, where the maximum reward for each node is 1, so the maximized average reward, given an arrival rate of 0.6, is 3, which is the value for which the reward converges. The overflow penalty ξ_k used in this test case was 2.0.

However, it is to be noticed that, with such a high delay constraint defined on Offload Test Case 1, the reward function defined in Equation 3.13 will be highly unlikely to miss its delay constraint, so the reward function is mostly limited to the second term of overflow. The PPO algorithm rather than opting for an offload strategy within the delay constraint, will minimize the overflow penalty, and therefore maximize the success rate in this case where almost no task will be discarded for not meeting its delay constraint. With a simple reward like this, convergence to an optimal case is simpler to achieve. In a more complex case, convergence is not guaranteed to go to the maximum value, since it can fall on a local maximum or even diverge.

In a second trial run, using the scenario *Env2* presented in Figure 5.1, the results are presented in Table 5.4.

Table 5.4: Performance on the second trial of Offload Test Case 1.

Algorithm	Nearest Node	Random Offload	No Offload	PPO algorithm
Success rate (%)	93.4	93.3	93.5	99.9
Overflow rate (%)	6.6	6.7	6.5	0.1

Given more resources for each node, scheduling whenever there is a task and resources available with No Offload strategy results in a total success rate of 93.5%, with the buffers overflowing with the other 6.5% of the tasks. With the Random Offload strategy, which offloads tasks when the threshold is reached, didn't improve the success rate, keeping it at 93.3%. Whereas the Nearest Node offloading strategy fell into a trap where the two most resource limited nodes are the nearest to each other, failing to improve the success rate, which is kept at a value of 93.4%. Using the offload strategy trained with the PPO algorithm the nodes learn to offload for a node with more resources increasing total success rate of a test run to 99.9%.

5.3.3 Offload case 2

A second simple case was created, similar to Offload Test Case 1, but now with a delay constraint on the number of milliseconds a task can take to complete.

$$\text{Case 2:} \quad D_k^{max} = 15 \text{ ms}, \quad L_k^c = 1200 \text{ cycles/bit}, \quad L_k^m = 800 \text{ MB}.$$

The delay constraint will ensure that the reward function now has to take into account not only the resources of the target node, but also the communication time, in order to maintain within the stipulated maximum delay. Therefore, the reward function shape can be complex enough for the DRL algorithm to fail a convergence to the optimal value.

Consider the first test environment *Env1*, illustrated in figure 5.1. In this scenario, the Offload Test Case 2 is also constrained by the memory resources, as in Offload Test Case 1, but the new time constraint limits the offload strategy to nearby nodes. The results for the second case run in the first scenario are in the Table 5.5

Table 5.5: Performance on the first trial of Offload Test Case 2.

Algorithm	Nearest Node	Random Offload	No Offload	PPO algorithm
Success rate (%)	95.5	79.6	67.3	86.9
Overflow rate (%)	0.1	1.8	4.0	0.9

The Nearest Node offload strategy will always opt for the best strategy possible in the first scenario, due to the distance pairing of the resource constrained nodes with resource rich nodes. Adopting that

strategy, given a delay constraint, the success rate of the nodes in completing the tasks was 95.5% and a negligible overflow rate, reduced from the Offload Test Case 1 due to the newly imposed time constraint. If the No Offload strategy is used, the resource constrained nodes deteriorate the success rate to 67.3% and 4.0% of overflow rate, while the Random Offload strategy sets the bar to 79.6% in success rate and 1.8% of overflow rate.

Figure 5.3: PPO first trial training reward on Offload Test Case 2.

The DRL algorithm manages to find a reward convergence during training, as illustrated in Figure 5.3, but it is not the best achievable solution with a success rate of only 86.9% and an overflow rate of 0.9%. It demonstrates itself better than a Random Offload strategy, but cannot find the simple solution necessary to maximize the indicators as the Nearest Node can.

Running the Offload Test Case 2 for the second test environment *Env2* in Figure 5.1, with more resources than in the first environment, increased overall indicators. The results are shown in Table 5.6

Table 5.6: Performance on the second trial of Offload Test Case 2.

Algorithm	Nearest Node	Random Offload	No Offload	PPO algorithm
Success rate (%)	81.2	78.3	80.2	83.9
Overflow rate (%)	1.4	2.7	1.9	1.9

A No Offload strategy managed a success rate of 80.2% and an overflow rate of 1.9%. A Random Offload strategy only degraded the resource richer nodes, decreasing the success rate to 78.3% and increasing the overflow rate to 2.7%. With the Nearest Node offload strategy in this scenario, it fell into the trap of offloading between resource constrained nodes, as mentioned for Offload Test Case 1, but managing to leverage the low communication delays that increased the success rate to 81.2% and decreasing the overflow to 1.4%.

On the other hand, the PPO algorithm, although converging to a sub-optimal reward, demonstrated

a stable performance with a success rate of 83.9% and an overflow rate of 1.9%. This hints towards a more flexible approach to an FC environment. While specific cases can outperform the algorithm in favorable situations, the PPO algorithm, with a well formulated MDP, can keep a decent and more flexible performance.

5.3.4 Increased load

The Offload Test Case 1 was run with the second scenario *Env2*, which has less resource constraints as presented in Figure 5.1, with a changing the arrival rate adopting the values of 0.7, 0.8 and 0.9. This was done to test the impact of an increased load for the algorithm training. The results are presented in Table 5.7.

Table 5.7: Success rate on the second scenario with Offload Test Case 1 over increasing arrival rate.

Arrival rate	Nearest Node	Random Offload	No Offload	PPO algorithm
0.7	88.0%	89.2%	88.0%	93.5%
0.8	84.5%	85.1%	84.5%	89.0%
0.9	82.3%	82.2%	82.3%	86.5%

As observed in Table 5.7, even with increasing resource constraints, the PPO algorithm can manage to outperform the baselines.

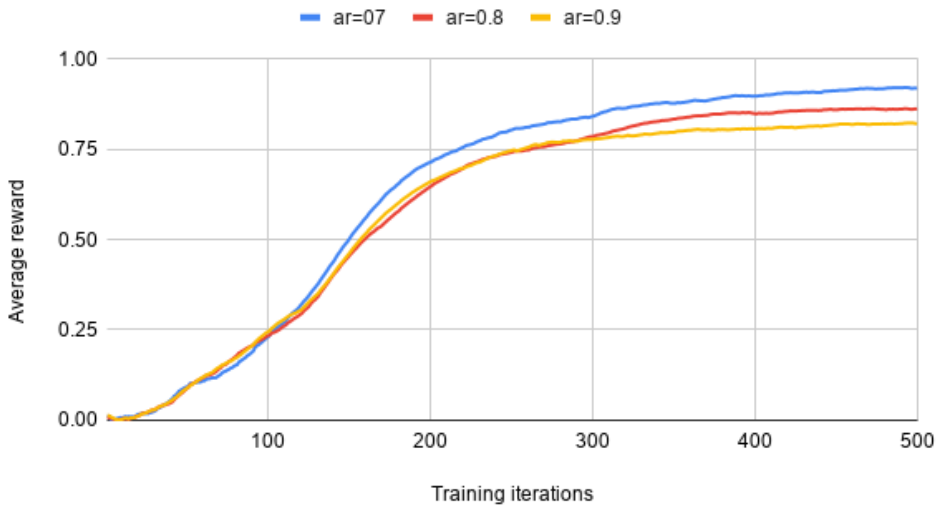

Figure 5.4: Normalized average reward over training iterations for an increasing arrival rate in Offload Test Case 1 running in scenario 2.

To prove convergence to the strategy adopted by the DRL algorithm, the moving average reward curves over training are illustrated in Figure 5.4. The graphs are normalized so that the minimum value

reached is 0 and the achievable theoretical maximum is 1. It can be observed that, with the increase of the arrival rate, the convergence average reward value decreases. Shows that the reward shape is dependent on the resource demand.

5.3.5 Reward function impact

To test the influence of the MDP reward formulation, tests were made with different formulations for the same case. The used case is the Offload Test Case 2 presented in Section 5.3.3, using the test environment *Env1* illustrated in Figure 5.1. It is done so, to demonstrate that a poor PPO performance can be due to its MDP formulation.

Based on the idea that the global objective is to maximize the success rate, shorter delays would mean a better capacity to process tasks and therefore maximize the success rate. With that rationale, a new reward, based on Equation 3.13, is presented:

$$\psi_i(O_i(t), X_i(t)) = \frac{1}{K} \cdot \sum_{k \in K_i} a_{i,k}(t) \cdot \left(\beta \cdot (-1)^{\mathbb{1}(D_k^{max} \leq D_{i,k})} - \xi_k \cdot \mathbb{1}(b_{f_{i,k},k}(t + D_{i,k}^t(t)) \geq \bar{b}_{f_{i,k},k}) \right), \quad (5.4)$$

where β, the new term inserted, would scale the reward within the delay constraints to minimize delay. That is done with the following equation:

$$\beta = \begin{cases} \left(1 - \dfrac{D_{i,k}^q + D_{i,j,k,n}^t}{D_k^{max}} \right) & \text{if } D_k^{max} \geq D_{i,k}, \\ 1 & \text{otherwise.} \end{cases} \quad (5.5)$$

The Equation 5.5 takes into account that the delay to process a task $D_{i,k}^p$ is constant in this specific environment, due to core attribution imposed limit referenced in Chapter 3, so unnecessary to minimize. It is also to be noticed that the equation always rewards whenever a tasks is completed within its delay constraint, i.e., it doesn't have a zero value.

Another formulation for the reward was done to only penalize the system. The purpose is to decrease the probability of the worst actions enough, for them not to become a favourable exploring path. With that intent, the reward formulation is a simple penalty whenever a task is not completed, either by being refused in an overflow situation or by failing to meet its deadline the penalty is calculated as follows:

$$\psi_i(t) = (-1) \cdot (f + o), \quad (5.6)$$

where f is the number of tasks that missed its deadline and o the number of tasks that overflow in time step t.

A third reward formulation was build with the objective of maximizing the success rate and minimizing

Table 5.8: Performance on the first trial of Offload Test Case 2 for different reward functions.

Algorithm	Nearest Node	PPO (Eq. 3.13)	PPO (Eq. 5.4)	PPO (Eq.5.6)	PPO (Eq. 5.7)
Success r. (%)	95.5	86.9	90.2	93.9	99.5
Overflow r. (%)	0.1	0.9	0.8	0.2	0.1

the overflow. It rewards whenever a task is completed and penalizes whenever a task is discarded due to a full buffer:

$$\psi_i(t) = (+1) \cdot s + (-1) \cdot o, \tag{5.7}$$

where s is the number of successfully completed tasks and o the number of tasks that overflowed in time step t.

Similar to Table 5.5, the new results with different reward formulations are presented in Table 5.8. However, the only baseline shown is the Nearest Node strategy since it previously had the best results.

As it can be observed in Table 5.8, the MDP reward formulation can have a great impact on the algorithm performance. This is a key point in applying the DRL algorithms to FC choreography, where the objective function has to be modelled in order to provide ease of convergence and maximization of performance.

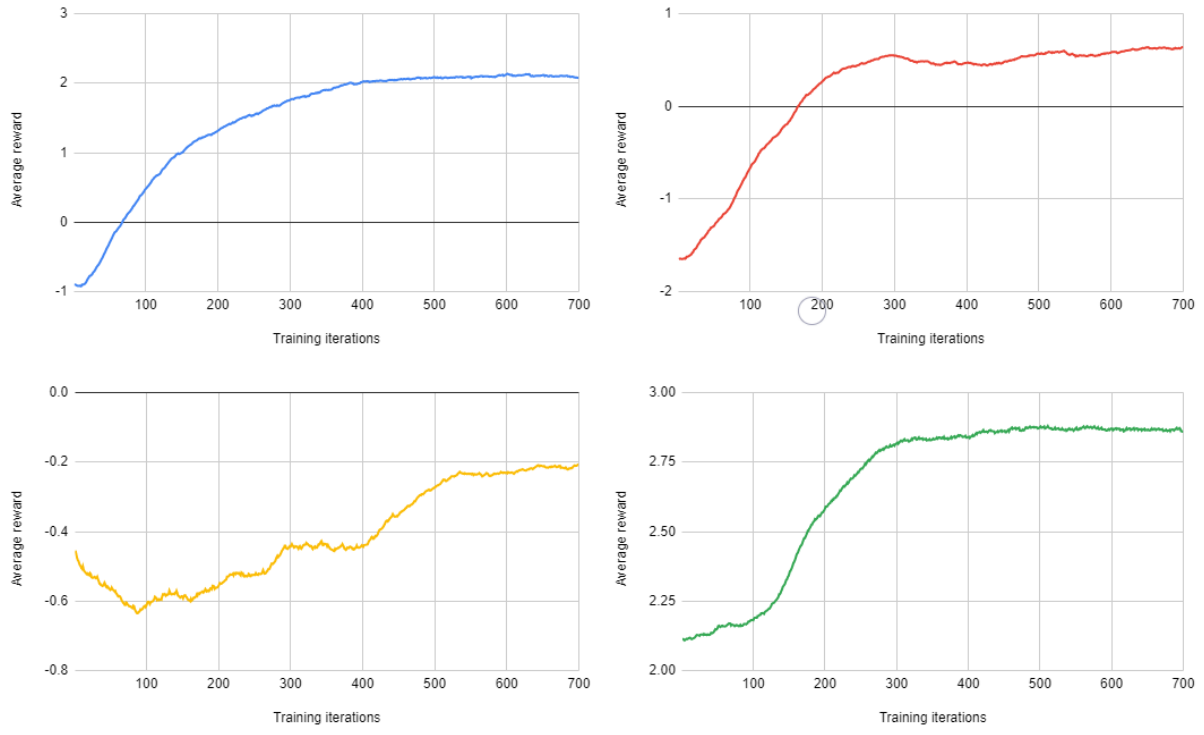

Figure 5.5: Moving average reward over training iterations. In blue trained with the reward on Equation 3.13, in red trained with the reward on Equation 5.4, in yellow trained with the reward on Equation 5.6, and in green trained with the reward on Equation 5.7.

As illustrated in Figure 5.5, different rewards can be more or less robust. The designed reward in Equation 3.13, represented in blue, reaches convergence smoothly. The modification to the original reward done in Equation 5.4, represented in red, increased the performance while also keeping a smooth convergence. In yellow, the sparse reward of Equation 5.6, although improving the results for this specific test case, has difficulties in converging since it leans heavily towards exploration. The final reward function, in Equation 5.7, managed to converge and provide exceptional performance for this case, however, the PPO algorithms training hyper-parameters, namely the trajectory length, had to be tuned since it proved likely to converge to a sub-optimal value if not given enough exploration options.

5.4 Complex test case

The addition of a second slice (K=2) in the FC environment increases the complexity of the test cases exponentially, increasing the action space to four different values and allowing task heterogeneity throughout the slices. With just 5 nodes and enough resources to process all tasks in each buffer, the action space size for each node is 2500.

This addition requests the baselines to use the scheduling algorithms, as presented in Section 3.3. Along with the offload strategy, the baselines are named as follows:

- Nearest Node Priority Queue (NNPQ);
- Nearest Node Round Robin (NNRR);
- Random Offload Priority Queue (ROPQ);
- Random Offload Round Robin (RORR);
- No Offload Priority Queue (NOPQ);
- No Offload Round Robin (NORR).

With two different slices, a resource heterogeneity is offered in the first test case:

Case 3 Slice 1: D_k^{max} = 1000 ms, L_k^c = 400 cycles/bit, L_k^m = 1200 MB;
Case 3 Slice 2: D_k^{max} = 1000 ms, L_k^c = 1200 cycles/bit, L_k^m = 400 MB.

A test run is made on the *Env2*, with the results presented in Table 5.9.

Table 5.9: Complex Test Case 3 performance results comparison with baselines.

Algorithm	PPO algorithm	NNPQ	NNRR	ROPQ	RORR	NOPQ	NORR
Success r. (%)	93.1	87.1	92.3	87.9	93.8	87.2	92.4
Overflow r. (%)	6.9	12.9	7.7	12.1	6.2	12.8	7.6

An explanation for the bad performance overall in Priority Queue scheduling is that, with no delay constraints, the priority attributed to a higher computational demanding slice might greatly increase the

overflow rate. On the other hand, the PPO algorithm managed to match the best baseline for this specific case. Here, the reward used was the one described in Equation 3.13 since it provided the best results without the need of hyper-parameter tuning. This hints towards a possible approach to choreography in a complex environment based on DRL algorithms.

A second test case with two slices is presented. In this case, both resource and delay constraint heterogeneity are offered:

Case 4 Slice 1: D_k^{max} = 15 ms, L_k^c = 400 cycles/bit, L_k^m = 1200 MB;
Case 4 Slice 2: D_k^{max} = 50 ms, L_k^c = 1200 cycles/bit, L_k^m = 400 MB.

A test run is made on the *Env1*, with the results presented in Table 5.10.

Table 5.10: Complex Test Case 4 performance results comparison with baselines.

Algorithm	PPO algorithm	NNPQ	NNRR	ROPQ	RORR	NOPQ	NORR
Success r. (%)	91.9	97.1	96.1	89.6	86.7	84.1	79.7
Overflow r. (%)	2.9	0.5	0.6	3.9	4.1	7.6	6.8

In this specific case, the Priority Queue scheduling has an inherent advantage, since prioritization coincides with a less computing demanding task, which results in an overall better success rate. Since the *Env1* favours the Nearest Node strategy, the best baseline is the Nearest Node Priority Queue with 97.1% success rate.

The PPO algorithm is trained with the reward presented in Equation 5.7. It fails to achieve the best optimization, scoring a 91.9% success rate in this test case, but demonstrates learning ability. However, with further testing the algorithm was shown not to be robust enough. Hyper-parameter tuning was necessary to achieve a reasonable performance, and it was not always the best. Its performance limitations are due to the non-linear approximation used, the nature of the PPO algorithm itself, the multi-agent scenario, and the MDP formulation itself.

5.5 Limitations in DRL for Fog Computing

The first limitation comes from the DNN approximation used in DRL. It is possible that the network proves to correctly approximate a sub-set of states, i.e. has a low loss for the sub-set, and therefore does not perform any major changes in the network. This cripples the exploration policy and the algorithm might reach convergence with a sub-optimal performance.

A second limitation comes from the PPO algorithm itself. As any on-policy algorithm, the bias-variance duality is an issue that influences the convergence value. Whereas short trajectories allow for a smaller variance in the reward, and therefore the algorithm can adapt its actions easier to converge to a

maximum, the bias to the total expected reward might be big enough for the convergence to be for a local maximum. On the other hand, a long trajectory would have a smaller bias to the total expected reward, but a a high variance, i.e. small change in the actions could have big impacts in the final reward, causing the reward convergence to be harder to find. For the presented test cases, the PPO algorithm managed this trade-off with a trajectory length of 64 time steps, performing an n-step TD target bootstrap.

A third limitation is the multi-agent scenario, which creates a non-stationary environment with the changing policies of other agents. In this case, a cooperation game is required and, in order to ease convergence in training, a common critic is implemented as proposed in [LWT+20]. However, the construction of the critic comes as an extra variable for the convergence.

A fourth source of performance limitations comes from the MDP itself. Firstly, the information in the state is not exhaustive, for example, it doesn't include task characteristics, making it so that the algorithm has to extrapolate the information by itself. However, this is not the main problem of the MDP, since the model free approach of this algorithm can easily adapt. The main limitation of the MDP is the reward function definition. For an easier convergence, it has to be designed so that the state-action relation with the reward is coupled. A sparse reward might increase the training time to unreasonable duration or provide poor performance, as discussed in [MKS+15] for Montezuma's Revenge. For a good performance, the reward has to translate the metrics of the optimization objective, and possibly have a concave shape to improve convergence for a global maximum.

Lastly, the reward designed in Equation 3.13, has a term that has to be estimated in an on-policy method. Only an off-policy method could store the reward for later completion before inserting it in a replay buffer - in order to tightly correlate the state-action-reward. Another note is that the total reward of this system is heavily dependent on the resources and placement of the nodes, as it should be. But this will translate into a more complex function to converge, dependent on the test case, so original training parameters are not guaranteed to converge for all cases.

6

Conclusions

Contents

The introduction of DRL to FC resource management could prove critical in the future of FC choreography. As a machine learning alternative to conventional supervised learning, RL has the advantage of creating its own training set with environment interactions. DRL proves to be a scalable and efficient approximation to conventional RL, and its flexibility makes it a good choice to the highly heterogeneous environment that is FC.

6.1 Achievements

In a specific sub-set of DRL, the actor-critic algorithms, it was shown that the PPO algorithm can adapt to different situations. It managed to obtain the optimal offload strategy when it was known and constant for all nodes, for example in the *Env1* favourable to Nearest Node, and when it was unknown and different for each node, as seen in *Env2*. It also managed to schedule tasks for completion based on a combined offload-schedule action. This demonstrated a flexible algorithm that can achieve a good performance without the need of human interaction, which, in turn, implies that there is no the need to choose a different algorithm depending on the scenario.

However, there are limitations found to the demonstrated performance as referenced in Section 5.5.

First, the DNN approximation used for the PPO algorithm. It is possible that the network proves to correctly approximate a sub-set of states, i.e., has a low loss for the sub-set, and therefore does not perform any major changes in the network. It was discovered when the loss function dropped values substantially without an improvement on the reward. This cripples the exploration policy and the algorithm might reach convergence with a sub-optimal performance.

Secondly, in the PPO algorithm itself, where the bias-variance duality influence the reward value to which the algorithm converges. Therefore, to improve performance there is the necessity to tune hyper-parameters, especially the trajectory length. The results displayed were obtained with the tuned training hyper-parameters of Table 5.2, different scenarios or MDP formulations might require different parameters.

Thirdly, the application of the PPO algorithm in this multi-agent scenario implies a cooperation game, where convergence to a common maximum might be hard to achieve. For this, other techniques should be explored to improve or replace the common reward and common critic for training.

Lastly, the MDP formulation, as seen in Section 5.3.5, can deeply impact the performance of the PPO algorithm. A design that manages to both represent the real system, in the state and the action, and to represent the performance objective, in the reward, has to be accurately created depending on the situation.

Nevertheless, DRL should be considered as path to explore in FC resource management, since, after training to an optimized strategy, it can map complex operations in a relatively short time and memory

space.

6.2 Future steps

To enable a deliverable DRL algorithm for FC resource management it is still necessary to investigate further. Future steps of research include:

- Improve and expand the built FC simulation environment modelling to be closer to reality, and allowing a broad but standardized research on the subject.
- Re-formulate the current MDP in order to hold a better perception of the system, increase the performance and make training easier.
- Develop an understanding test suite of the FC problems. For example, create specific test cases as an IoV scenario in an intersection, a Surveillance scenario, among others, to then create a broad test suite where the each DRL algorithm is tested and compared with common performance metrics.
- Implement other DRL algorithms for choreography to compare in this environment, to provide a wider baseline for research, and keep up with the DRL research for multi-agent scenarios.
- Explore the impact of the DNN architectures for FC choreography algorithms.
- Explore the orchestration side of resource management with the inclusion of DRL.

Bibliography

[AAB+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[Bha18] Shweta Bhatt. 5 things you need to know about reinforcement learning, 2018. [Online; accessed May 19, 2020].

[BK20] J. Baek and G. Kaddoum. Heterogeneous task offloading and resource allocations via deep recurrent reinforcement learning in partial observable multi-fog networks. *IEEE Internet of Things Journal*, pages 1–1, 2020.

[BKG+19] J. Baek, G. Kaddoum, S. Garg, K. Kaur, and V. Gravel. Managing fog networks using reinforcement learning based load balancing algorithm. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–7, 2019.

[Con] NS-3 Consortium. ns-3. [Online] Available on https://www.nsnam.org/.

[Con17] OpenFog Consortium. Openfog reference architecture for fog computing, 2017. [Online] Available on https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf.

[Con19] Industrial Internet Consortium. Joining forces, 2019. [Online] Available on https://www.iiconsortium.org/pdf/2019-Q1-Quarterly-Report.pdf.

[CRB+11] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Softw. Pract. Exper.*, 41(1):23–50, January 2011.

[DWG+19] Peiran Dong, Xiaojie Wang, Liang Guo, Joel Rodrigues, Xiangjie Kong, Jun Huang, and Ricky Kwok. Deep reinforcement learning for intelligent internet of vehicles: An energy-efficient computational offloading scheme. *IEEE Transactions on Cognitive Communications and Networking*, PP:1–1, 07 2019.

[GBC17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. The MIT Press, 2017.

[GDGB16] Harshit Gupta, Amir Dastjerdi, Soumya Ghosh, and Rajkumar Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in internet of things, edge and fog computing environments. *Software: Practice and Experience*, 06 2016.

[GJR+15] T. N. Gia, M. Jiang, A. Rahmani, T. Westerlund, P. Liljeberg, and H. Tenhunen. Fog computing in healthcare internet of things: A case study on ecg feature extraction. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 356–363, 2015.

[HGS15] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[Hin12] Geoffrey Hinton. Neural networks for machine learning online course - lecture 6a - overview of mini-batch gradient descent, 2012.

[KBTP17] F. A. Kraemer, A. E. Braten, N. Tamkittikhun, and D. Palma. Fog computing in healthcare–a review and discussion. *IEEE Access*, 5:9206–9222, 2017.

[KK02] Spiros Kapetanakis and Daniel Kudenko. Reinforcement learning of coordination in cooperative multi-agent systems. In *Eighteenth National Conference on Artificial Intelligence*, page 326–331, USA, 2002. American Association for Artificial Intelligence.

[LL19] S. Lee and S. Lee. Poster abstract: Deep reinforcement learning-based resource allocation in vehicular fog computing. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 1029–1030, 2019.

[LWT+20] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments, 2020.

[MBM+16] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[MDM+16] A. Monteiro, H. Dubey, L. Mahler, Q. Yang, and K. Mankodiya. Fit: A fog computing device for speech tele-treatments. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–3, 2016.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei Rusu, Joel Veness, Marc Bellemare, Alex Graves, Martin Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–33, 02 2015.

[PGM+19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[SB18] Richard S. Sutton and Andrew Barto. *Reinforcement learning: an introduction*. The MIT Press, 2018.

[Sha49] C. E. Shannon. Communication in the presence of noise. *Proc. Institute of Radio Engineers*, 37(1):10–21, 1949.

[SML+18] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

[Tea16] Eclipse Deeplearning4j Development Team. DL4J: Deep Learning for Java, 2016. Available online on https://github.com/eclipse/deeplearning4j.

[VH08] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, Brussels, BEL, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Vie19] José C. Vieira. Fog and cloud computing optimization in mobile iot environments. Master's thesis, Instituto Superior Técnico, 2019.

[VSN14] Antonio Virdis, Giovanni Stea, and Giovanni Nardini. Simulte — a modular system-level simulator for lte/lte-a networks based on omnet++. In *Proceedings of the 4th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, SIMULTECH 2014, page 59–70, Setubal, PRT, 2014. SCITEPRESS - Science and Technology Publications, Lda.

[Wil92] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[WSH+16] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1995–2003, New York, New York, USA, 20–22 Jun 2016. PMLR.

[XLH+19] Kai Xiong, Supeng Leng, Jie Hu, Xiaosha Chen, and Kuanli Yang. Smart network slicing for vehicular fog-rans. *IEEE Transactions on Vehicular Technology*, PP:1–1, 02 2019.

[YFN+18] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 08 2018.