Universitat de les Illes Balears

MASTER'S THESIS

EXPLORING THE USE OF DEEP REINFORCEMENT LEARNING TO ALLOCATE TASKS IN CRITICAL ADAPTIVE DISTRIBUTED EMBEDDED SYSTEMS

Ramón Rotaeche Fernández de la Riva

Master's Degree in Intelligent Systems (MUSI)

Specialisation: Artificial Intelligence and Internet of Things

Centre for Postgraduate Studies

Academic Year 2020-21

EXPLORING THE USE OF DEEP REINFORCEMENT LEARNING TO ALLOCATE TASKS IN CRITICAL ADAPTIVE DISTRIBUTED EMBEDDED SYSTEMS

Ramón Rotaeche Fernández de la Riva

Master's Thesis

Centre for Postgraduate Studies

University of the Balearic Islands

Academic Year 2020-21

Key words:

Deep Reinforcement Learning, Distributed Embedded Systems, Combinatorial Optimization, Machine Learning

Thesis Supervisor's Name:

Alberto Ballesteros Varela, PhD

Julian Proenza Arenas, PhD

Exploring the use of Deep Reinforcement Learning to allocate tasks in Critical Adaptive Distributed Embedded Systems

Ramón Rotaeche Fernández de la Riva
Tutor: Alberto Ballesteros Varela and Julián Proenza Arenas
Trabajo de fin de Máster Universitario en Sistemas Inteligentes (MUSI)
Universitat de les Illes Balears
07122 Palma, Illes Balears, Espanya
ramonrotaeche@gmail.com

***Abstract*—A Critical Adaptive Distributed Embedded System (CADES) is a group of interconnected nodes that must carry out a set of tasks to achieve a common goal, while fulfilling several requirements associated to their *critical* (e.g. hard real-time requirements) and *adaptive* nature. In these systems, a key challenge is to solve, in a timely manner, the combinatorial optimization problem involved in finding the best way to allocate the tasks to the available nodes (i.e. *the tasks allocation*) taking into account aspects such as the computational costs of the tasks and the computational capacity of the nodes. This problem is not trivial and there is no known polynomial time algorithm to find the optimal solution. Several studies have proposed Deep Reinforcement Learning (DRL) approaches to solve combinatorial optimization problems and, in this work, we explore the application of such approaches to the tasks allocation problem in CADESs. We first discuss the potential advantages of using a DRL-based approach over several heuristic-based approaches to allocate tasks in CADESs, and we then demonstrate how a DRL-based approach can achieve similar results to the best performing heuristic in terms of optimality of the allocation, while requiring less time to generate such allocation.**

***Index Terms*—Deep Reinforcement Learning, Distributed Embedded Systems, Combinatorial Optimization, Machine Learning**

I. Introduction

A Distributed Embedded System (DES) is a combination of hardware and software, where the hardware is a set of interconnected *nodes*, and the software is typically implemented as a set of computational elements, called *tasks*, which are executed in the nodes in order to achieve some common goal. DESs play a key role in many engineering fields, such as avionics, energy management or telecommunications.

DESs are used in real-world applications, and some of them have *real-time* (RT) and *dependability* requirements. We refer to DESs that have demanding RT and dependability requirements (i.e. strict real-time response and very high dependability) as *critical* DESs. A system is said to have RT requirements if its correct operation depends, not only on its ability to provide a correct response, but also on its ability to provide such response before some deadline. On the other hand, a system is said to have dependability requirements [14] if it is required to have certain attributes that provide enough trustworthiness on the system's ability to provide a correct service. Two key dependability attributes are reliability and availability. A system has high reliability when it exhibits high probability of providing a correct service in a continuous manner. This is an essential requirement when the consequences of an incorrect service can be significantly negative (e.g. harm to humans or significant loss of data). In contrast, a system has high availability when it exhibits high probability of being prepared to provide a correct service. This is a requirement in systems where failures might be acceptable as long as the time and effort required to go back to a correct operation are minimal. For example, in a local WiFi network.

Nowadays there is strong interest in using critical DESs in changing *operational contexts*. By operational context, we mean all the relevant aspects involved in the operation of the system that are susceptible to change. More specifically, as seen in Fig. 1, we consider changes in the functional requirements (i.e. the fundamental functionalities the system must carry out) and in the non-functional requirements (e.g. the RT guarantees or the dependability guarantees). These requirements are referred to as *operational requirements*. Additionally, we also consider the *operational conditions*, that is, the circumstances under which the system has to operate. This includes both the state of the environment and the state of the system itself which could change, for example, due to faults in its components. For a critical DES to operate efficiently and effectively under a changing operational context, it needs to be *adaptive*. Adaptivity in the context of a critical adaptive distributed embedded system (CADES), implies that the system must be able to dynamically assign its computing and communication resources while in operation.

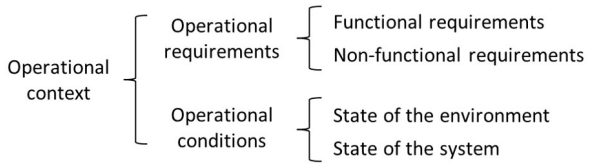

Figure 1. **Components of the operational context**

The Dynamic Fault Tolerance for Flexible Time-Triggered Ethernet (DFT4FTT) project [3] proposes a complete infrastructure that enables CADESs to meet their RT and dependability requirements while adapting to changing operational contexts. In DFT4FTT, a privileged node called *Node Manager* (NM) (Fig. 2) is responsible for continuously monitoring the operational context. If a change jeopardizes the correct operation of the system, then the NM is also responsible for searching and applying a new *system configuration* in a timely manner. A *system configuration* (or *configuration* for short) in DFT4FTT determines several attributes for each task and message. In particular, for each task and message, it defines the node to which it is assigned; schedule-related attributes such as the period, the deadline, or the priority level; and the level of replication. Replication consists of executing redundantly (multiple times) a given task or a given communication in order to increase the dependability of the system by making it tolerant to faults in some of the replicas.

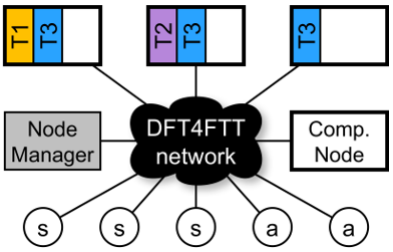

Figure 2. **DFT4FTT architecture**. A key component of the DFT4FTT network is the Node Manager (NM), which is in charge of monitoring the operational context, finding a new configuration if a change occurs, and applying such configuration.

The general goal that motivated this work was to explore the use of Machine Learning (ML) techniques to find new system configurations in the context of the DFT4FTT project. There are several reasons that make us consider ML, and in particular Deep Reinforcement Learning (DRL), as an interesting approach to solve this problem. For example, several studies have shown that DRL can outperform other approaches for solving similar problems. The reasons for exploring a DRL-based approach as well as the main theory behind ML and DRL are presented in more detail in section III.

Motivated by the general goal of exploring the use of ML to find new system configurations in DFT4FTT, and as discussed further in section II, we decide to focus on the tasks allocation problem (i.e. deciding which tasks are assigned to each node), which is one of the multiple parts of a system configuration. Thus, the specific objective of this project is to design and implement a ML-based approach capable of allocating tasks to nodes in a CADES that follows the DFT4FTT guidelines, and analyze the potential benefits of such approach both qualitatively and quantitatively. In order to achieve such objective we have executed the following tasks:

- Define the specific tasks allocation problem that needs to be solved (e.g. allocation criteria, assumptions made)
- Select the best ML-based approach to solve our problem
- Analyze the potential benefits of the selected ML-based qualitatively
- Define and implement the selected ML-based approach
- Execute the experiments required to quantitatively analyze the potential benefits of our approach

The results presented in this document show that a DRL-based approach is a promising option to find new system configurations in DFT4FTT, and the conclusions of this work will help guiding future research in the line of work of the DFT4FTT project. The remainder of this document is structured as follows. In the next section, we specify the problem to be solved, including the necessary simplifications. In section III we justify the use of a DRL-based approach and introduce the theory behind DRL and the specific techniques employed in this work. In section IV, we present our DRL-based solution. In section V, we show the experiments we have carried out together with their results. Additionally we compare these results with those obtained for heuristic-based solutions. In section VI we discuss related work. Finally, in section VII, we discuss the conclusions and future work.

The remaining sections assume some basic understanding of ML concepts and, in particular, of the design and training of Deep Neural Networks (DNNs).

II. Problem Statement

As with many engineering and mathematical problems, a reasonable approach to solve a complex issue is to break it down into smaller and simpler parts on which to gradually build a solution. In line with this principle, we have decided to narrow the scope of this work by simplifying the concepts of system configuration and operational context introduced in section I.

A. System configuration: simplifications considered

In this work we focus on a key aspect of a system configuration: the *tasks allocation*. Given a set of tasks that must be executed, a *tasks allocation* can be defined as the distribution of such tasks into the nodes of the system. More formally, it is a many-to-one binary relation between the set of tasks and the set of nodes available.

We leave aside other elements of the system configuration such as the tasks replication or the communications between nodes. Therefore, in the rest of this document we use the term “tasks allocation” rather than “configuration” to emphasize that we are only solving this aspect of the system configuration. Similarly, we will refer to “tasks reallocation“rather than “reconfiguration“ when talking about the specific act of finding a new tasks allocation after changes in the operational requirements.

It must be noted that the ability to reallocate tasks not only makes the system more adaptive, but also contributes to improve the dependability. This is because such reallocation ability allows the system to recover tasks, meaning that it can tolerate faults affecting tasks or nodes.

B. Operational context: simplifications considered

As discussed in section I, adaptive distributed embedded systems must be able to operate under unpredictable dynamic

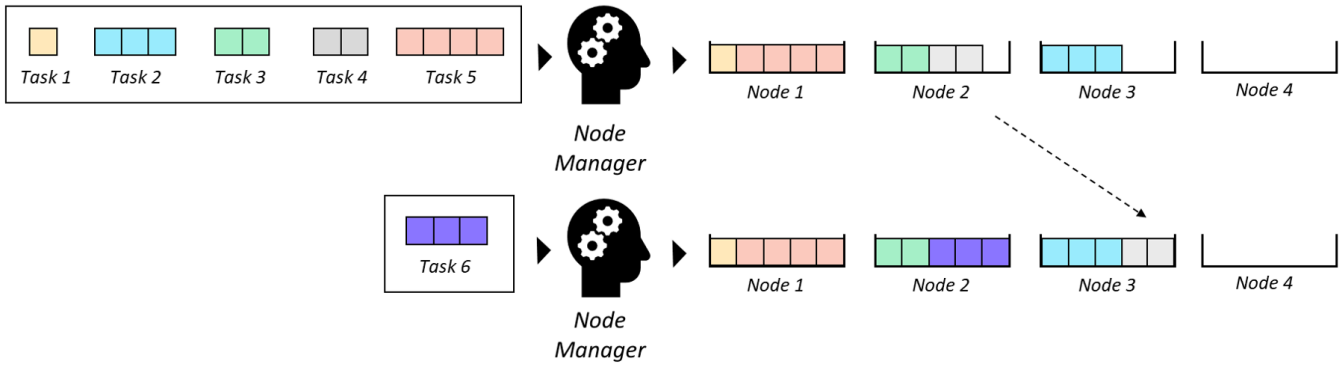

Figure 3. **Minimizing the number of active nodes - Illustrative example**. A reallocation in order to keep the number of active nodes at a minimum

operational contexts, which encompass both the operational requirements and the operational conditions.

The operational requirements considered in our problem are the set of tasks that must be allocated. We assign a *computational cost* to each task, which is a scalar number representing the quantity of resources that each tasks needs in order to be properly executed.

The operational conditions considered in our problem are the resources available to execute tasks in each node, which we refer to as the *computational capacity* of each node. A node's computational capacity determines the number of tasks it can execute at the same time. The tasks allocated to a given node cannot add up to a total computational cost higher than the node's capacity.

The Node Manager (NM) must allocate any given set of tasks to the available nodes. Each set of tasks may have a different size (i.e. different number of tasks), and each task of the set might have a different computational cost. In our system, the sets of tasks could change unpredictably and the NM should be able to reallocate the tasks as a response to this changes. This is in contrast to a non-adaptive system where the NM would only be programmed to deal with some specific sets of tasks defined a priori.

To simplify, we set the nodes' computational capacity to be constant for all the nodes and throughout the entire duration of the system operation.

We simplify the RT requirements by factoring them via the computational cost of the tasks and the computational capacity of the nodes, This means that if the NM is able to allocate all the tasks to the available nodes so that no node receives a subset of tasks with an aggregated computational cost higher than its computational capacity, we can consider that the tasks will meet their deadlines (see subsection D, *Assumptions made*, for a discussion on the assumptions required for this to be true). This is a simplification that allows us to avoid explicitly addressing the tasks deadlines, execution time, periods, RT-related factors, etc; which would add enormous complexity to the problem and hence we leave it for future research.

C. Tasks allocation criteria

When the set of tasks that must be allocated changes (i.e. when the operational requirements change) the NM must find a new tasks allocation. Such allocation could be chosen based on different criteria. We will model the problem where the NM must find the tasks allocation that, for a given set of tasks, uses the least number of nodes, which we will refer as the *number of active nodes*. If a new task appears, the set of tasks that need to be allocated changes, and a reallocation might be required to keep the number of nodes at a minimum. See the example in Fig 3: initially, the set of tasks to be allocated is *Task 1* to *Task 5* (represented in the top half of the picture), but if in a later step a new task (Task 6 in the picture) needs to be allocated, the new set of tasks that the node manager must allocate is *Task 1* to *Task 6*, which triggers a reallocation.

We chose to try to minimize the number of active nodes as our main goal for several reasons. The first one is that it results in us essentially trying to solve a version of the well known bin-packing problem [10], meaning that there are several well studied algorithms that we can compare with our DRL-based solution in terms of their performance. In addition, this allocation criterion would help to achieve the potential non-functional operational requirement of reducing the energy consumption, since a minimum number of nodes would have to be active in order to execute all tasks. Lastly, and this is more contentious, it could be argued that minimizing the number of active nodes contributes to meet the RT requirement (which, as discussed earlier, is a typical non-functional operational requirement in critical systems), because concentrating tasks in the same node reduces the network traffic (as less messages are required between nodes, which simplifies the scheduling of the complete distributed system). Of course, concentrating tasks in a minimum number of nodes has its own disadvantages (e.g. more challenging scheduling in each of the active nodes, higher severity in the event of a failure), and therefore it is not our intention to present this allocation criteria as a sufficient condition for a well-designed CADES.

It must be noted that despite we model the search for tasks allocation that minimizes the number of nodes as an optimization problem, our system does not need to find the *best* possible tasks allocation (i.e. the optimum). The system can work with just "good" tasks allocations. In fact, as discussed later, our proposed approach will find good solutions but is not guaranteed to find the optimal solution. This is completely

expected since, as discussed later, no algorithm has been found for this problem, other than an exhaustive search, capable of guaranteeing the finding of the optimal solution for any given set of tasks.

D. Assumptions made

We consider the problem in which all the tasks presented to the Node Manager (NM) must be allocated to nodes. Therefore, it is necessary to guarantee the existence of an allocation that includes *all* the tasks (i.e. no task is left without a node assigned). This could be enforced in the design of a real system by having a number of nodes large enough to guarantee a solution given the maximum possible number of tasks and the possible maximum cost. In a real scenario, this cannot be fully guaranteed because there is always the possibility of failing nodes, meaning there is a always a chance that there are not enough nodes to execute all tasks. In this work, we will assume that in such case, we would have some mechanism like the one in DFT4FTT [3] by which the system enters in *degraded mode*, taking measures such as prioritizing the most critical tasks or stop replicating some tasks.

An additional condition is required to meet the RT requirements, which is that the subset of tasks assigned to each node is *schedulable*. Schedulable means that, given each task's computation time and period, it is possible to find a schedule for the node that always meets the deadlines for said tasks.This requirement could be enforced (for nodes that use RT operating systems applying rate-monotonic scheduling) by capping each of the nodes' capacity made available in the allocation process to 69.3% of its *actual* computational capacity, since *Liu & Layland* [16] proved that for any set of n periodic tasks, a schedule exists if the resulting node's utilization is below 69.3%. The node's utilization is the sum of each task's utilization, which is defined as the computation time required to execute the given task in the given node, divided by the task's period. Therefore, in order to use this theorem to enforce our schedulability assumption (i.e. that the subset of tasks assigned to each node is schedulable) in practice, it would require to express the tasks' computational cost as the number of operations required each time the task is executed divided by the task's period, and to express the node's computational capacity as 0.693 multiplied by the number of operations that the node can execute in a time unit.

E. Tasks reallocation requirements

Lastly, two requirements related to the tasks reallocation that are common in CADES have been considered: memory requirements and latency. Given that in many systems the NM's software needs to be deployed in resource-constrained processors such as microcontroller units (MCU), it is important to develop an algorithm that can be stored in these devices' flash memory, which is up to 1MB in industrial settings [15], and that is able to generate a new tasks allocation as fast as possible to increase the likelihood that that not only the tasks meet their RT requirements but also that the reconfiguration itself can be done in a quick manner (although we will not try to establish any strict RT response requirements for the reconfiguration itself). In our work, we have factored these requirements by ensuring that our model's size is below 1MB and by having the inference latency as one of the dimensions benchmarked in our experiments.

F. Summarized problem statement

Recapitulating, the actual problem tackled in this work has been the design of a solution capable of allocating tasks to nodes in a way that ensures that no node receives more tasks than it can handle, while minimizing the total number of active nodes (i.e. nodes that receive at least one task). In addition, special attention will be paid to the memory requirements and time required to generate a solution.

III. Introduction to DRL and motivation for a DRL-based approach

The problem we are trying to solve (as described in the previous section) is an optimization problem, since we want to minimize the number of active nodes. More precisely, it is a *combinatorial optimization* problem, which is a type of optimization problem where the objective is to find the optimal object from a finite set of objects (in our case, the set of objects would be the set of all possible ways in which the tasks can be allocated, and an object would be a specific mapping between tasks and nodes).

As mentioned in the previous section, our problem is equivalent to the bin-packing problem [10], which is an NP-optimization problem, meaning that there is no known polynomial time algorithm to find the optimal solution [12]. This types of problems are typically be solved using solvers (e.g. [4]) or heuristics. In this work, we wanted to explore alternative approaches based on ML techniques, which, as detailed later in this section, might present a number of advantages.

In the rest of the section, we first introduce the concept of Deep Neural Network (DNN), which is a key component of the discussed Machine Learning approaches. Secondly, we introduce the concept of supervised ML and discuss why it is not fit for our purpose. We then introduce the concept of Reinforcement Learning and Deep Reinforcement Learning (DRL). Lastly, we elaborate on why DRL is the right approach for this problem and reflect on its suitability to a CADES.

A. Deep Neural Networks

As discussed in section II, in our problem the input is a set of tasks and the computational cost of each task, while the output must be some representation of how these tasks are assigned to the available nodes. We need to learn some way to map our inputs to the output. Given the high dimensionality of our input-output pairs, we considered Deep Learning (DL), that makes use of *Deep Neural Networks* (DNNs) [7] to parameterize the input-output mapping function.

A deep neural network, given an input x and an output y, is a mapping function $y' = f(x, \theta)$, that tries to approximate the unknown function f^* which maps any possible set of input-output pairs $y = f^*(x)$. The symbol θ represents the

parameters that define this mapping function (i.e the operations that map x to y) and they can be “learnt" using different ML algorithms. There are many types of DNNs, which differ in the computations they perform to compute the output based on the input. However, all DNNs have some things in common that is precisely what makes them a DNN.

First of all, the input and outputs are always represented as multidimensional arrays (a.k.a. tensors), with no specific restriction to the number of dimensions. For example, the output can be a simple scalar (which can be seen as a 1x1 array), or a colored image, which is represented as a three-dimensional array.

Secondly, all the computations of a DNN can be represented as a composition of functions, each function is referred to as a *layer* that takes the array output by the previous layer, performs a mathematical operation, and returns a new array. The array taken by the first layer would be the input, and the array returned by the last array would be the output. This layered representation is the reason we call it a “network", and they are called "deep" because they typically chain a large number of layers. See Fig 4.

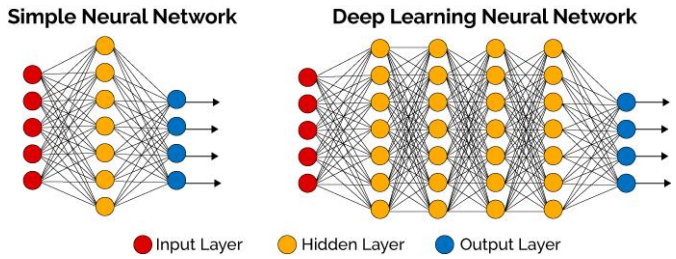

Figure 4. **Schematic view of a Neural Network**. Deep Neural Networks (DNNs) are just Neural Networks with multiple hidden layers[7]

Lastly, each of these layers typically have several *neurons* which take each element from the input array, multiply it by a scalar and then apply an *activation function*. The scalars that multiply the elements of the input array are precisely the learnt parameters θ of a network. The activation function is typically a non-linear transformation (e.g. ReLU [7], soft-max [7])

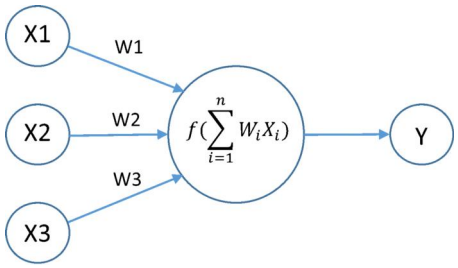

Figure 5. **Schematic view of a neuron**. Elements from the array returned by the previous layer are multiplied by a scalar (the parameters θ) and then a non-linear activation function is applied.

B. Supervised Machine Learning

The first family of ML techniques that we considered was supervised ML, which allows to infer a function (a.k.a learn a function) that is able to map input data to an output (a.k.a estimation or prediction) even if the model has never ‘seen’ that input data before. That is why it is said that supervised ML models learn to ‘generalize’. In order to do this, supervised ML techniques require a training dataset with examples of input-output pairs. However, supervised ML is quite limited for our purposes, because it requires examples of the optimal tasks allocation (desired output) for each set of tasks in the training dataset. There is no known polynomial time algorithm to find the optimal solution for our problem [12], so generating optimal solutions is complex and time consuming, specially with large sets of tasks.

C. Reinforcement Learning

Once supervised learning was discarded for the reasons described in the previous section, further research led us to conclude that the right approach for this problem was Reinforcement Learning (RL). Reinforcement Learning (RL) is the sub-field of machine learning that studies methods for a decision maker, the *agent*, to learn, only by interacting with the *environment* (everything outside the agent), what *actions* to take so as to maximize a numerical signal, the *reward* [20], accumulated over time.

The RL problem is formalized using a decision process (Fig. 6), where at each time step *t*, the agent receives a representation of the environment’s *state* S_t and on that basis selects an action A_t. As a consequence of its action, the agent receives the numerical reward R_t and the new state S_{t+1}.

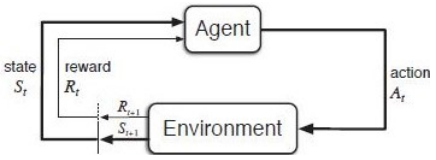

Figure 6. **The agent–environment interaction in a RL decision process**[20]

The mechanism that the agent uses to decide which action to take is called *policy*. A policy is a mapping from states to actions [20], which we denote by $\pi(A_t \mid S_t)$. In some RL approaches like the one used in our work, the policy is a *parameterized policy*, meaning that such mapping between states and actions is a parameterized function. The objective is to learn the parameters (i.e. to learn the policy) that yields the maximum reward. The parameters, which we denote by θ, are modified based on the reward obtained in the interactions with the environment, using techniques like the *policy gradient method* [20], which is the one we use and which we explain in detail in the next section.

D. Deep Reinforcement Learning

One way to parameterize the policy that is followed by the agent is with a DNN, since a DNN is effectively a parameterized function that takes an input (in this case, the state) and produces an output (in this case, the action). Using a DNN, is particularly useful when the states space is very large (or infinite), like in our case where the number of different sets of tasks that the agent can receive is very large (as the number

of total tasks can vary as well as the cost of each task). Given that we use a DNN to parameterize the policy, our approach falls into the field of Deep Reinforcement Learning (DRL), which is the term used to refer to the set of RL techniques that make use of DNNs. Note that our approach is only one of many ways in which DNNs can be used in RL, so the method we will detail later on is not a comprehensive view of what DRL is.

E. Rationale for a DRL approach

One of the main reasons for DRL being a better approach to our problem is that, as opposed to supervised ML, the model can learn to take better actions (in our case the actions would be the tasks allocation) without actual examples of what the right action is. The only signal it needs is the reward associated to each action that is undertaken. For example, in our case and based on the problem statement from previous section, the reward would be higher the lower it is the number of active nodes.

One can see that our problem fits perfectly in the RL framework: the NM (the *agent*) has to select among a set of possible tasks allocations (the *actions*), based on the tasks cost and nodes capacity (the *state*), in order to minimize the number of active nodes (the *reward*). In fact, although the study of modern RL falls under the ML umbrella, its "modern fathers" A. Barto and Richard S. Sutton [20] were highly influenced (as declared by themselves [20]) by previous ideas in the field of *adaptive* optimal control (e.g. [13]), a field concerned with controlling unpredictable dynamical systems, and which one could see as more closely related to CADESs.

Lastly, it is worth mentioning that when we first decided to explore the use of DRL to tackle this problem, although we had some intuitions on its potential advantages, it was driven primarily by the desire to undertake such an academic exercise. However, as we progressed in our research, we realized that a DRL-based approach, when compared to solvers and heuristic based algorithms, could have several advantages that make it a real alternative for CADESs. Namely:

- DRL methods have proved to be near as good or even better than many popular heuristics in solving different combinatorial optimization tasks [5], [9].
- The same algorithm can be used to teach the agent to maximize any *reward* function. In contrast, solvers and heuristics used in combinatorial optimization are generally specific to the problem statement. This is especially relevant when using high-dimensional states (i.e. states characterized by a large number of variables) and/or complex reward functions. In those cases, finding a heuristics-based solution might require significant ad-hoc work.
- Once a DNN is trained, the inference latency (i.e. the time required for the DNN to generate a solution) is relatively low, with the potential to be lower than many heuristics that might require exploring the search space, sorting the inputs, etc. This suggests that DRL agents might be a suitable solution for a CADES with real-time requirements.
- Recent developments in Tiny ML [15], [6] propose frameworks to deploy complex DNN models on resource constrained processors such as microcontroller units (MCU), with good results in terms of inference accuracy and latency. This suggests that DRL agents could be used in resource-hungry CADESs.

IV. Problem formulation and approach

As discussed in section II, in this work we aim to design of a solution capable of allocating tasks to nodes in a way that ensures that no node receives more tasks than it can handle, while minimizing the total number of active nodes (i.e. nodes that receive at least one task).

Our tasks allocation problem maps to the DRL framework discussed in the previous section as follows:

1) the *state* is the set of tasks that must be allocated, each of them with a cost
2) the *environment* is the set of nodes available and its capacity
3) the *action* is a specific mapping of tasks to nodes
4) the *reward* must be some scalar that is inversely proportional to the number of nodes used (since we want to minimize this number)
5) the *policy* is a DNN that maps any given state to an action (i.e. any given set of tasks to a specific allocation of tasks to nodes)

Before jumping into the details in the upcoming subsections, let's discuss the items listed above, which will help to outline the design decisions that are required to formulate an approach. These design decisions are then discussed one by one in the upcoming subsections.

If we want our DNN to take a given set of tasks as an input, and generate as the output a specific allocation of those tasks to nodes, the first thing we need is a way to numerically *represent* both a set of tasks and an allocation of tasks to nodes. This is what we call the *input-output representation approach*, and is discussed in section A.

The second thing we need to define is how the reward is computed, because as mentioned above, the only real restriction we have is that it needs to be scalar that is inversely proportional to the number of nodes used. This, the *reward signal approach*, is what we discuss in section B.

Third, we need to choose a specific RL technique that can be used to "teach" our policy to allocate tasks in a way that maximizes the reward. For the sake of clarity, note that "teaching our policy to allocate tasks in a way that maximizes the reward" is equivalent to saying "finding the DNN parameters that map inputs to outputs in a way that maximizes the reward". The *RL approach* is discussed in section C.

Lastly, the DNN has to take the input (the set of tasks) and generate the output (the allocation of tasks to nodes). As mentioned, the numeric format for the input and the output is discussed in section A. However, we also need to define the operations that take place to compute the output based on the input, which depends on the architecture of the DNN. This, the DNN's architecture, is discussed in section D.

A. Input - output representations approach

The input (i.e. the state) to our parameterized policy (i.e. to our DNN) is the set of tasks that must be allocated. We decide to represent it as a sequence S of l tasks characterized by their cost c_i, $i \in [1, L]$.

We decide to represent a tasks allocation (i.e. a specific mapping of tasks to nodes) as follows. Our representation (i.e. the output of the DNN) will be a sequence A that represents the order in which the tasks in S are allocated to nodes following a given heuristic H. To represent this allocation order, we define A as a sequence of L integers $a_i \in [1, L]$, $i \in [1, L]$, where a_i represents the position of the task in the input sequence S. This representation is better understood with an example, like the one shown in (Fig. 7).

With regards to the heuristic H that we use once we have defined the allocation order A, we will experiment with two:

- A Next-Fit (NF) heuristic: the first available node receives tasks until task a_i does not fit, at which point a_i is allocated to the next node, which keeps receiving tasks until the same happens, and so on.
- A First-Fit (FF) heuristic: as with NF, the first node is "opened" and tasks are sequentially allocated until a task does not fit. However, when a new node is "opened" the previous one is not "closed". Nodes are kept "open" unless they are completely full. On each allocation step all "open" nodes are checked until a node where the task fits is found. If no node is found, then a new node is "opened".

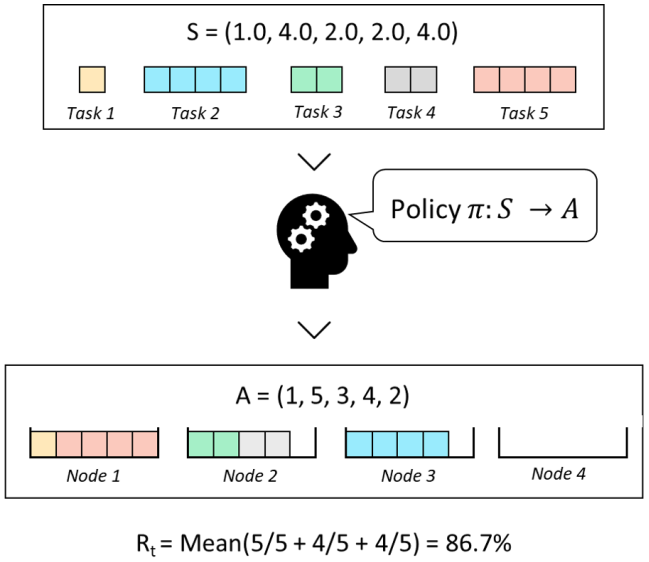

Figure 7. **Input-output representation example** for a set of 5 tasks and node capacity C = 5, for the case where the NF allocation heuristic is followed. The set of available tasks is a sequence containing the cost of each task. The output of the DNN, the sequence A is the order in which the tasks are allocated following the NF rule. This is a simple example, but maximizing the occupancy ratio is an NP-optimization problem which becomes challenging to solve for large sets of tasks

Note that what we are doing is using the allocation order sequence A generated by the DNN to implicitly dictate how tasks are grouped together. Alternative representations for the output were considered, such as a sequence containing the index of the node to where each task is allocated. However, that leads to a large equivalence class of solutions, and, as pointed by [5], restricting as much as possible the equivalence class for the outputs leads to better results. Moreover, the chosen representation, when combined with the selected allocation heuristics, guarantees that all generated solutions comply with the nodes' maximum capacity constraint. If the former was not guaranteed, then the reward function would also need to check for whether there is any node that exceeds its capacity, and return a negative or zero reward.

B. Reward signal approach

We have selected the *average node occupancy ratio* (O) as the reward to maximize. O, as its name suggests, is calculated by averaging the node's occupancy ratio (sum of allocated tasks' cost divided by node capacity) of all the active nodes (i.e. nodes that receive at least one task). For example, the value of O in fig. 7 is 0.867 which is the mean of 1.0, 0.8 and 0.8.

Note that we could have chosen a more direct metric to maximize, like the inverse of the number of active nodes. However, our metric is more independent of the total number of tasks in the set. This facilitates learning a policy that can be applied to sets of tasks of different sizes.

C. RL method used

It is not the objective of this work to present and explain all of the techniques and algorithm that fall under the RL umbrella. We will only explain the method used in our solution, which is a *DRL-based actor-critic policy gradient method*. The first reason why we chose this method over the rest was because it can handle very large (even infinite) states spaces, which is a requirement in our case (there are many possible sets of tasks). In addition, this method had shown fast convergence and good results for other types of combinatorial optimization problems [5].

In a DRL-based actor-critic policy gradient method, the policy is parameterized using a DNN. As mentioned in previous sections, the DNN maps each state to an action and we denote it by $\pi(A \mid S, \theta)$. The symbol θ, as it is common practice in ML and as explained in section 2, represents the parameters of the DNN, and stresses the fact that the mapping of states to actions depends on those parameters.

Before detailing the specific algorithm and formulas, let's explain the *training* process at a more conceptual level. Remember that we want to find the parameters θ for the DNN that maximize the reward. We start with a DNN with random values for θ. Policy gradient methods rely on running a large number of simulated interactions with the environment. A simulation starts by generating a random initial sequence of tasks S_t (the state), we then use our existing DNN $\pi(A \mid S, \theta)$ to generate a tasks allocation A (the action) and then we calculate the associated reward O. Lastly, the parameters θ of the DNN are then updated using a formula discussed in next paragraph. Each simulation and parameters update step is known as a *training step*.

Intuitively, if A has led to a *better reward than the average reward of current policy* $\pi(A \mid S, \theta)$, we want to *update the parameters* θ *of the DNN in the "direction" that leads to more outputs like* A. In the formula that we present later in this section, the concept of "updating parameters in the direction that leads to more outputs like A", is represented by $\nabla_\theta p(\pi(A \mid S_t, \theta_t) = A_t)$, which is the gradient with respect to the parameters θ of the probability of having chosen A_t. The concept of "better reward than the average reward that we were getting" is represented by the term $(O_t - \hat{v}(S_t, \theta'_{t+1}))$, where O_t is the obtained reward and $\hat{v}(S_t, \theta'_{t+1})$ is a function that estimates the expected reward that will be obtained with the current policy and parameters $\hat{v}(S, \theta')$ given the sampled state S_t'.

The function $\hat{v}(S, \theta')$ is known as the *critic*. The critic is a *state-value* function, meaning that it takes a given state S and approximates the expected reward that will be obtained following policy π_θ. We use another DNN with parameters θ' to parameterize the critic. The critic parameters θ' are also continuously updated during training. Fig. 8 gives an overview of the training process.

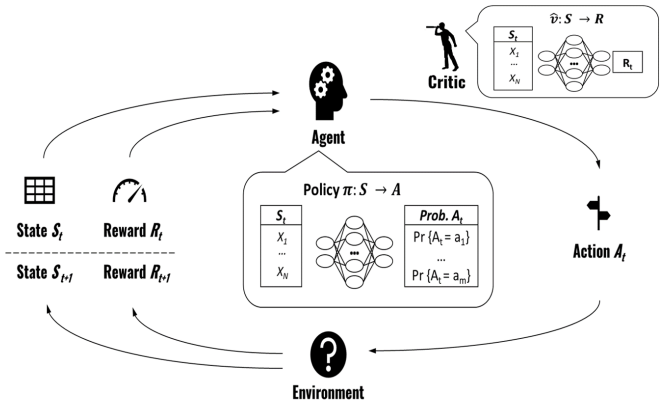

Figure 8. **DRL-based actor-critic policy gradient**. We use a DRL-based actor-critic policy gradient method. This means that two DNNs are trained. The first DNN parameterizes the policy used by the agent (a.k.a actor) to map states to probabilities of taking each action. The second DNN parameterizes the critic, which maps states to the reward that it is expected to receive from the environment following the current policy

The specific algorithm followed in the training process that we just described is the so-called *actor-critic reinforce stochastic gradient ascent algorithm*, based on the policy gradient theorem [20]. The algorithm, on each training step, updates the parameters θ and θ' of the policy and the critic DNNs respectively as follows:

At time step t:

Randomly generate state S_t*, and then sample action a from* $\pi(A \mid S_t, \theta_t)$

Calculate reward $O_t = O(S_t, a)$

Update critic parameters:

$$\theta'_{t+1} = \theta'_t - \alpha' \nabla_{\theta'_t} MSE(O_t, \hat{v}(S_t, \theta'_t)) \quad (1)$$

Update policy parameters:

$$\theta_{t+1} = \theta_t + \alpha(O_t - \hat{v}(S_t, \theta'_{t+1})) \nabla_\theta p(\pi(A \mid S_t, \theta_t) = A_t) \quad (2)$$

Note that equation (1) is just a standard training step for a supervised ML regression problem where we want our critic DNN $\hat{v}(S, \theta')$ to learn to estimate $O(A, S)$, and hence we use as a loss function the Mean Squared Error (MSE) of the difference between the obtained reward and the estimated one.

α' and α are the learning rates, which are simply two scalars representing the update "step size" for equation (1) and (2) respectively.

$p(\pi(A \mid S_t, \theta_t) = A_t)$ is the probability of having chosen A_t, given state S_t and following policy $\pi(A \mid S_t, \theta_t)$. This probability can be calculated because the DNN that we use (more details on its architecture are provided in the next subsection) does not directly output a specific action (i.e. a specific mapping of tasks to nodes), but rather, it outputs a *probability for each possible action*. Therefore, we can know what was the probability of choosing A_t. A natural question is how we choose A_t from all the possible actions. During training, we choose action A_t by *sampling* it from the discrete probability distribution for each possible action given by our DNN $\pi(A \mid S, \theta)$. Note that this is different to supervised ML, where we would select the action (or the class in supervised ML) with the highest probability. This means that during training we might select an action which does not have the highest probability according to policy π, but this is actually essential to ensure the *exploration* of the actions space, a fundamental principle in RL. During inference, however, once training has been completed, we make the policy *greedy* on the action values, meaning that in inference we always select the action with the highest probability.

D. Architecture of the policy and critic DNNs

We use a *pointer network* [21] to parameterize our policy $\pi(A \mid S, \theta)$. A pointer network is a DNN architecture for solving variable length sequence-to-sequence problems whose output can be interpreted as a sequence "pointing" at positions in the input sequence. This is ideal for the input-output representation approach defined in subsection A.

The pointer-network follows a encoder-decoder Recurrent Neural Network (RNN) architecture [18]. This architecture is widely used in problems where the inputs and output are sequences (like our problem, but also in areas like Natural Language Processing since text is also sequential data), because it is designed to process each element of the input sequence S in order in what is known as the *encoding* (allowing the DNN to take the order of the elements into account) and then recurrently generate the output sequence A in what is known as the *decoding*. The decoding takes place in recurrent *decoding steps* where at each step i, the element i of sequence A, a_i, is a function of the input sequence S as well as of the previous outputs $a_1, ..., a_{i-1}$. We have based our specific implementation on the architecture proposed by [5].

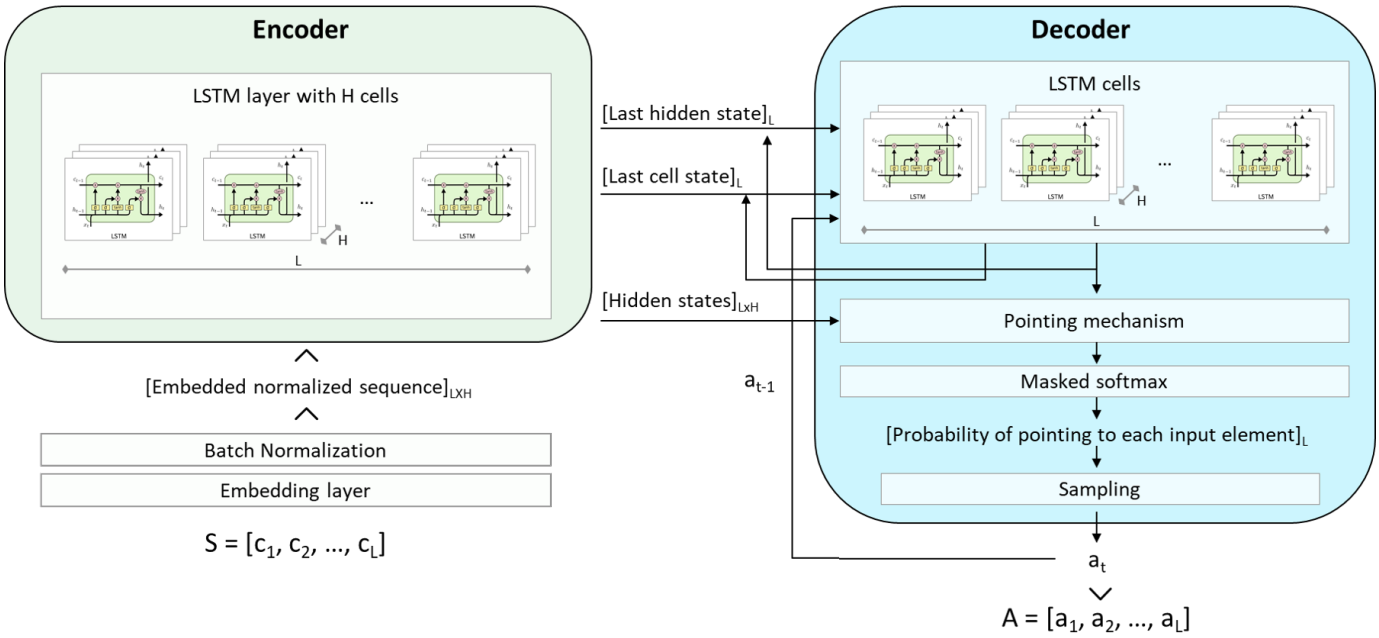

Figure 9. **Neural network architecture**. The encoder reads the input sequence $S = [c_1, ..., c_L]$ of task costs and ultimately produces an allocation order sequence A, which can be defined as numbers "pointing" at positions in the input sequence S

See Fig. 9 for a detailed overview of the architecture of our policy DNN. The encoder reads the input sequence $S = [c_1, ..., c_L]$ of task costs, one at a time, and transforms it into a sequence of latent memory states of dimension LxH. Where H is the hidden dimension of our DNN. Each element of the input sequence is first transformed into a H-dimensional embedding, obtained via a linear transformation shared across all input steps whose parameters are also learned. The decoder network also maintains its latent memory states and uses a *pointing mechanism* to produce a probability distribution over the next task that should be "pointed" at. Once the next task is selected (by sampling from the distribution), it is passed as the input to the next decoder step, together with the last memory states.

The pointing mechanism is based on the so-called Bahdanau's attention mechanism first proposed in [2]. It takes the decoder hidden state as the query vector, and the hidden states from the encoder as the reference vectors. The output of the pointing mechanism is masked first (to avoid pointing to the same input element twice) and then a soft-max activation is applied, so that the output can be interpreted as a probability distribution representing the degree to which the model is pointing to each element of the input sequence. The allocation order A (see subsection A) is generated by sampling a_i, at each decoding step i, from the probability distribution of a_i, whose discrete density function is the output of the DNN (which has a softmax activation in the last layer). Appropriate masking is applied before the softmax activation to ensure that the same position cannot be "pointed" twice.

The DNN used to approximate the critic function $\hat{v}(S, \theta')$ is made of an encoder with the exact same architecture as the DNN used for the policy, followed by a similar attention mechanism but using the encoder outputs as the query and reference vectors. Lastly, it has two standard feed-forward layers, the last one with a single neuron since $\hat{v}(S, \theta')$ is a regression model that attempts to estimate a scalar (i.e. the expected reward given state S).

E. Additional remark for readers familiar with other RL problems

One thing to note is that we have defined an action as the sequence A representing the allocation order. This means that in our RL framework, when the agent receives a set of tasks that must be allocated, it allocates all of them in *one* action. Using the RL terminology: the *episode* lasts only one time step, and the recurrent nature of the tasks allocation decision is already taken care by the RNN architecture of the policy.

An alternative approach would have been to define an action as the allocation of a single task to a node, and the RL episode would last L time steps, where L is the length of the input tasks sequence S. This alternative approach is equivalent to our approach if in the alternative one the reward is set to zero for all timesteps except for the last one, which would make sense because the average occupancy ratio is not known until the end (and using a partial average occupancy ratio on each timestep would deviate the agent from learning to optimize the final average occupancy).

Our one-timestep approach is more convenient to implement. The only difference that needs to be taken into account is in equation (2) from section $IV.A$ above. In that equation, the term $p(\pi(A \mid S, \theta) = a)$ represents the probability of having selected action a. In our approach a is actually a sequence A of L integers $a_i \in [1, L]$, $i \in [1, L]$ (where a_i represents the position of the task in the input sequence). Therefore, $p(\pi(A \mid S, \theta) = a)$ in equation (2) should be the probability

of generating the sequence A following policy π_θ, which can be calculated using the chain rule as:

$$p(\pi(A \mid S, \theta) = \{a_1, ..., a_L\}) = \prod_{i=1}^{L} p(\pi(a_i) \mid a_1, ..., a_{i-1}, S, \theta)\} \quad (3)$$

$p(\pi(a_i) \mid a_1, ..., a_{i-1}, S, \theta)$ is the probability of sampling a_i in the decoding step i of the pointer network, which is given by the output of the masked soft-max layer (note that $p(\pi(a_L) \mid a_1, ..., a_{L-1}, S, \theta) = 1$ always because in the last timestep there is only remaining task that can be pointed at).

V. Experiments

A. Experiments objective

The primary objective of our experiments is to evaluate how well the policy learns to generate allocations with high average occupancy O.

Problem label	# of tasks	Min. task cost	Max. task cost	Nodes capacity
1	24	1	6	8
2	40	3	8	10
3	50	4	14	15
4	Variable (min 3 - max 50)	4	14	15

Figure 10. **Problem conditions in our experiments**. Tasks costs are sampled from the uniform distribution over the interval [Min. task. cost, Max. task. cost]

In addition we want to gain some basic understanding of the inference latency and of the model size, in line with our hypotheses exposed in section III.E, related to the suitability of a DRL approach for time-sensitive and resource constrained environment like CADESs.

B. Experiments approach

We have considered four different problem conditions (Fig. 10). In problems 1 to 3, the sets of tasks that the agent learns to allocate all have always the same size (24, 40 and 50 respectively). We have chosen these three different problem conditions pseudo-arbitrarily. We say “pseudo” because we have paid some attention to choosing a set of conditions that did not easily lead to very high occupancy ratios which could not pose a big enough “challenge” to our agent. Problem 4 tests the ability of the agent to learn to allocate sets of tasks of variable length, with a minimum of 3 and a maximum of 50 tasks. This means that the same parameters θ of the policy DNN are applied to sequences of length 3, 4, 5, .. up to 50. In practice, all the sets of tasks are input as an array of length 50, but they might have zeroes at the end, meaning that there are no tasks. The DNN has two masking layers (one for the inputs and one for the outputs) that ensures these tasks are not taken into account.

As discussed in section II, we assume that there is always at least one valid tasks allocation (i.e. there is always one valid allocation where no task is left unassigned). In our implementation we enforce this by assuming that we always have the number of nodes required to fulfill the most optimal allocation that our NM is able to generate. Therefore, the number of nodes available is not a fixed value which is part of the problem conditions, but rather we assume it is high enough to accommodate the generated tasks allocations.

We will train our parameterized policy to see how well it learns to allocate tasks, by analyzing the average occupancy ratio O achieved after 10,000 *training steps*. One training step consists of one iteration of the actor-critic reinforce stochastic gradient ascent algorithm described in section IV.C.

As it is common in ML, specially in stochastic gradient ascent algorithms like ours, we will use *training batches*. This means that in each training step, we don’t just use a *single* randomly generated state (i.e a set of tasks), but rather, we use a *batch of randomly generated states*, which in our case is of size 128 (i.e. the batch consists of 128 randomly generated sets of tasks). In batch training, the parameters of the DNNs are updated using the mean of the individual updates calculated for each of the 128 samples (following equations (1) and (2) in section IV.C). This approach, i.e. updating the parameters only once with the mean of the batch rather than 128 times with the value of each single element of the batch, has been shown to achieve faster and closer to the optimal convergence [7].

For comparison purposes, we will also evaluate the avg. occupancy ratio obtained for the batch with three popular heuristics [10] used to solve the bin-packing combinatorial optimization problem (Fig. 11): the Next-Fit (NF), the First-Fit (FF), and the First-Fit Decreasing (FFD) heuristic. The NF and FF heuristics are the same as the ones described in section IV.A, when discussing the Input-Output representation approach and the heuristic H assumed for the allocation order A generated by the DNN. Nevertheless, for ease of read, we describe the NF and FF heuristics again below, together with the additional heuristic used in the benchmark: the FFD. Thus, the benchmark heuristics are:

- *Next-Fit* (NF): Tasks in the input set are considered in an arbitrary order (i.e. in the order in which they are randomly generated). The first node is “open" and tasks are sequentially allocated until a task does not fit. At that point the node is "closed" and the next node is “opened". A “closed" node does not receive any additional task during the remaining of the allocation.
- *First-Fit* (FF): As with NF, tasks in the input set are considered in an arbitrary order and the first node is “opened" and tasks are sequentially allocated until a task does not fit. However, when a new node is "opened" the previous one is not “closed". Nodes are kept “open" unless they are completely full. On each allocation step all “open" nodes are checked until one where the task fits is found. If no node is found, then a new one is “opened".
- *First-Fit-Decreasing* (FFD): Similar to the FF heuristic but the input tasks are first sorted in non-increasing order of their cost.

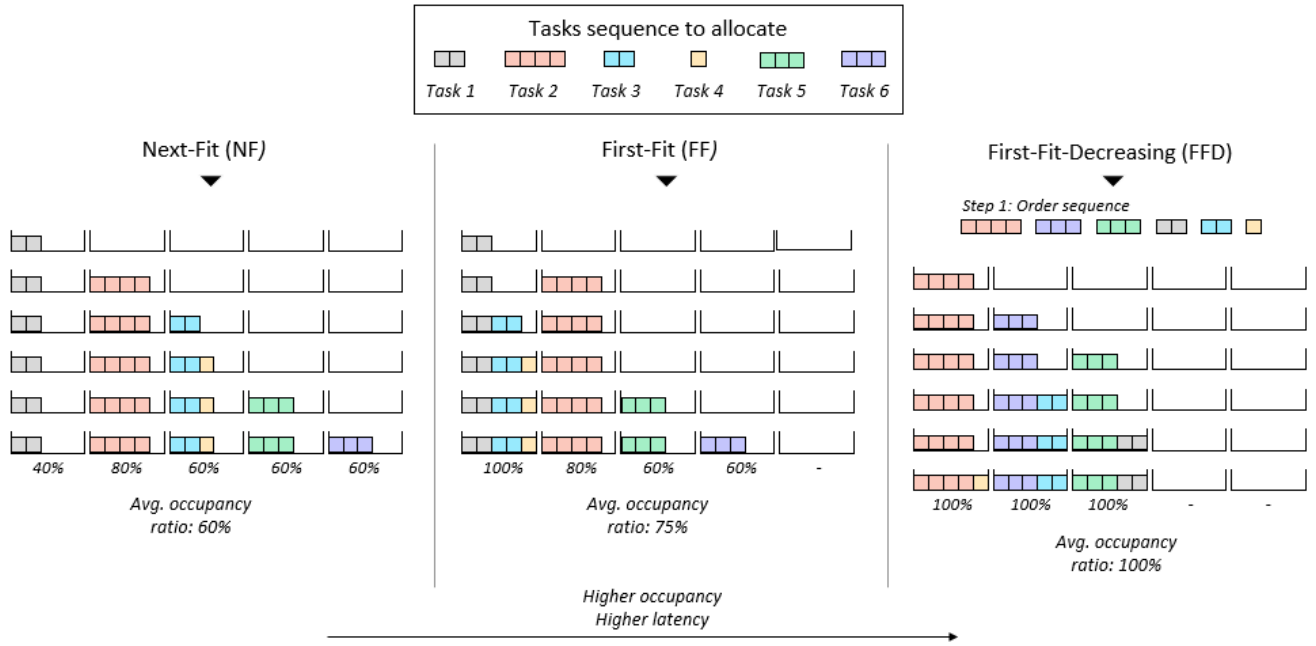

Figure 11. **Benchmark heuristics**. Visual example of how a given sequence of tasks would be allocated following each of the three benchmark heuristics

We have selected the NF and FF heuristics because they are two well-studied *online* heuristics used to solve this problem. *Online* means that they do not have to process the full input set prior to the start of the allocation. Online heuristics are faster, have lower time complexity and lower memory requirements than offline heuristics such as FFD or our own DRL-based approach (which, as opposed to online methods, require to process the entire set of tasks prior to generating a solution), and that is precisely why we included two online heuristics in our benchmark. If such heuristics were to lead to a similar or better average occupancy ratio than our DRL-based approach, it would be hard to find reasons in favour of the DRL-based method.

We also include an offline heuristic, FFD, in our benchmark because we also wanted to compare against a solution that it is proven (and observed in our experiments) that yields results closer to the optimal than online heuristics.

To calculate the average occupancy ratio obtained with each of the heuristics, we take the mean of 1280 random samples of sets of tasks (100 batches of 128 samples).

As explained earlier, in our DRL approach, we take the allocation order A output by the DNN and test both the case where tasks are allocated following a NF heuristic, and following a FF heuristic. When looking at the avg. occupancy ratio that we obtain *just* using those heuristics, those are the baseline values for each respective case, since those are the avg. occupancy ratios that the policy would score if it does not manage to learn at all and simply generates random allocation orders A.

We have not tried to improve the results with *hyper-parameter tuning*, i.e. changing different combinations of parameters in the network architecture and training algorithm to see if it drives better results, or with more sophisticated decoding strategies (e.g. beam search or those proposed by [5]), which could potentially yield better average occupancy ratio O.

In order to have it as a reference, we tried to calculate the optimal average occupancy for the four problems analyzed. We used Google's tools for optimization [1]. However, it was not possible due to the amount of computation time required. We needed to calculate the optimal for a large sample size (as discussed above, to benchmark the results we are using 1280 samples, 100 batches of 128 samples), which was unfeasible since, already for Problem 1 (which is the one with the shortest input length), it took the solver $> 10h$ to find the solution for some of the samples. There are some exact algorithms like [12] that claim to be very fast for most of the problem instances (although still no guarantees of polynomial time). However, implementing such algorithms was beyond the scope of this work. Something that can be said regarding the optimal average occupancy is that it is likely not far from the ratio obtained with the FFD heuristic: the FFD heuristic guarantees a solution with no more than $11/9$ of the optimal number of nodes [10] , and [12] empirically showed that the FFD achieved the optimal solution in 94.7% of the problem instances tested.

For the inference latency comparison, we will compare the time it takes to allocate the tasks following the FFD heuristic (the one that achieves better occupancy ratios) with the time it takes to generate an allocation with our agent, using the same hardware.

To understand the model size, we will just look at the storage space that the agent's DNN uses in the hard drive, which is directly related to the number of parameters of the DNN and the format in which they are stored, which in our case is 32-bit integers.

C. Experiments implementation

For each of the experiments, we have trained 3 models, each with a random weight initialization, during 10,000 training steps and selected the model with the best results. An initial learning rate of 0.001 was used for both the agent and the critic network, with a 0.9 decay rate every 1000 steps. This simply means that the learning rate (i.e. the scalars α' and α in equations (1) and (2) in section IV.C) is reduced every 1000 steps by a factor of 0.9. This is shown to improve the convergence, because it helps to escape spurious local minima at the beginning of the training while avoiding oscillation around local minima at the end of the training [11].

The actor and critic DNNs have been implemented in Python using the popular Tensorflow framework [8]. We have implemented both the DRL-based approach and the benchmark heuristics as Tensorflow's [8] functions in graph mode, and run them on a laptop with an Intel(R) Core(TM) i7-7600U CPU, no GPU, and 32GB of RAM.

D. Experiments results

As shown in Fig. 12 our agent matches the best heuristic (the FFD) when being trained to generate allocation orders that then follow a FF rule. Perhaps more interesting, when trained to generate allocation orders that then follow a NF rule (which is faster than the FF and the FFD), it achieves a higher avg. occupancy ratio than the NF and the FF heuristics, and gets close to the FFD performance. This is is consistent across all experiments.

Problem label	*Average occupancy ratio %*				
	DRL Agent		Heuristics		
	DNN + NF	DNN + FF	NF	FF	FFD
Problem 1	92.3	93.8	77.5	89.2	93.8
Problem 2	89.1	89.4	73.9	85.8	89.4
Problem 3	89.3	89.8	74.7	86.5	89.8
Problem 4	86.2	86.7	74.3	84.4	86.7

Figure 12. **Optimization performance experiments results**. Avg. occupancy ratio (%) comparison between our trained DRL-agents and the selected heuristics. Values are the average across 100 randomly generated batches of 128 samples each.

In Fig. 13, it can be seen how the average occupancy ratio that the agent achieves improves over time as the DNNs are trained (both the agent's DNN and the critic DNN).

For Problem 4, where the agent achieves an average occupancy ratio of 86.2% and 86.7% (depending on the allocation heuristic H used), we checked that results were consistent for all possible input lengths. As it can be seen in Fig. 14, the combination of the DNN plus the NF heuristic is consistently better than the NF and FF heuristics and consistently ends up close to the FFD for all algorithms. As expected, the average occupancy for all approaches tends to be higher the longer the sequence of tasks, because there are more possible combinations. The same consistency across all lengths is observed for the DNN plus the FF heuristic.

With regards to the inference latency. As mentioned earlier, given an input set of tasks, we have compared the time it takes to our agent (when using the NF allocation rule) and to the FFD algorithm to generate a solution. The FFD heuristic requires a pre-sorting of the input tasks and a FF allocation, which impacts the time required to generate a solution.

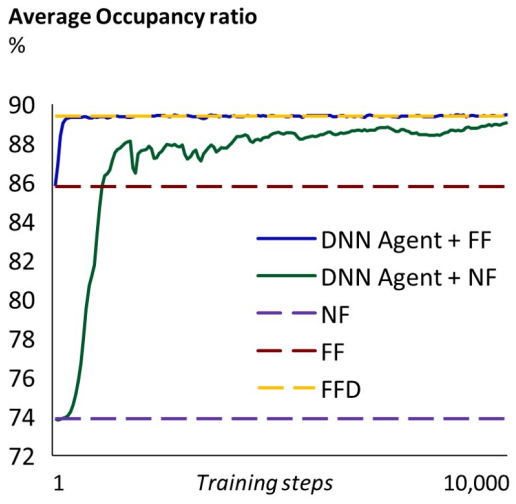

Figure 13. **Training history**. Problem 2: batch average occupancy ratio (%) after each training step, compared to the average occupation ratio obtained using the NF, FF and FFD heuristics

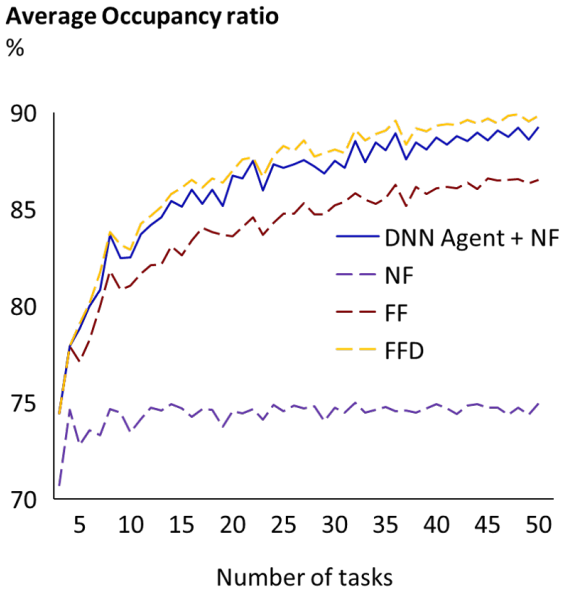

Figure 14. **Problem 4 results by size of the input set of tasks**. Average occupancy ratio obtained in Problem 4 when applying the trained DNN plus a NF heuristic, and when applying the three benchmark heuristics. A batch of 128 random samples was generated for each possible length, the values shown are the mean of the average occupancy ratio for each of those 128 sequences

For the Problem 2 conditions, on average, it takes *18 ms* to generate a solution using our RL-based agent, while it takes *68 ms* with the FFD algorithm. Note that the time measured for the RL agent includes the time required to allocate the output sequence following a NF heuristic.

This does not pretend to be a rigorous comparison of

inference latency as among other things, it depends on the implementation and the hardware used. For example, the use of GPUs, given their parallelization capacity, would speed up the DNN inference. Nevertheless, the results align with our hypothesis that DRL has a better time-reward trade off than heuristic based approaches. Specially in memory constrained environments where time-optimized implementations of sorting and searching operations might not be feasible.

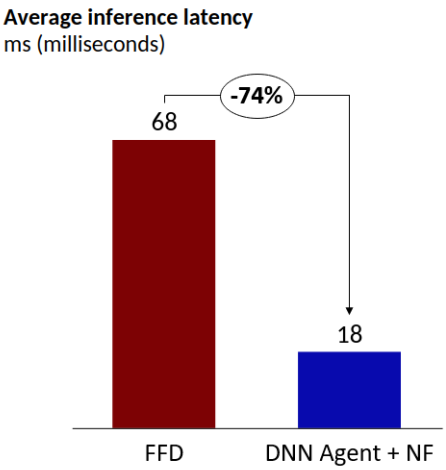

Figure 15. **Inference latency comparison**. Problem 2: average time required to generate a solution with the FFD vs with the DRL + NF approach

Regarding the model size, our model required 264 KB of storage memory. Our DNN's hidden dimension (i.e. number of neurons in the hidden layers) is 64. This is lower than typically used sizes (e.g. 128), but in the spirit of developing something that can easily be deployed into resource hungry CADESs, we wanted to test the performance of a lighter DNN. The pointer network architecture and the hidden dimension size of 64 resulted in the agent's DNN having 66,754 parameters (this was for problem 3, where the input length was 50. The other problems had less parameters since the maximum length was smaller). These 66,754 parameters are ultimately stored as 32-bit integers requiring a storage memory of 264 KB. which is more than enough to fit a standard micro-controller unit (MCU), which might have up to 1 MB of storage capacity [15].

VI. Related work

In this work we solve a combinatorial optimization problem using DNNs. The general term for this type of approach is *neural combinatorial optimization*. A significant contribution to this area was made by [21] with the pointer network architecture mentioned earlier. The authors proved that such architecture worked well for several combinatorial optimization tasks in a supervised ML setting (where examples of the optimal solutions were needed for the network to be trained).

A framework to tackle combinatorial optimization problems using DNNs and RL was proposed by [5]. Our DNN architecture (except for the input and output layers which are specific to our problem's input-output representation) and the selection of the specific RL algorithm to use (the actor-critic reinforce stochastic gradient ascent algorithm) is based on their work. However, they apply it to different combinatorial optimization problems with different problem conditions. We had to design all the aspects of this work that are specific to the tasks allocation problem (or equivalently, to the bin packing problem). For example, the input-output representation approach, the reward signal or the benchmark heuristics.

On the grounds of [5], several studies have tackled resource allocation problems [9], [17], [22], [19] with DRL techniques. Perhaps, the most related to ours are [19] and [9]. In [19], the authors aim to allocate services to hosts in a way that minimizes power consumption, resulting in a reward function similar but not identical to ours. The DNN architecture, problem constraints and constraints-enforcing strategies are also different. In [9] they solve a 3D packing problem by minimizing the surface required to pack a set of items, which requires to take 3 decisions for each problem instance. They use DRL for taking one of the decisions and heuristics for the other two.

To the best of our knowledge, our work is the first to use DRL to train a DNN to directly generate solutions to the 1D bin packing problem [10]. A problem which, despite its simple formulation, is NP-hard and has multiple applications like the one discussed here for CADESs. The degree of optimization achieved combined with the low inference latency suggest that this approach might be a better option for a CADES than traditional heuristics.

VII. Conclusions and further work

Our solution for the tasks allocation example has illustrated the potential of using DRL-based methods to find system configurations in CADESs. This is in line with the general aim of this project (as presented in section I) of exploring the use of Machine Learning (ML) techniques to find new system configurations in the context of the DFT4FTT architecture.

Moreover, we have fulfilled the specific objective of this project, which was stated as follows: design and implement a ML-based approach capable of allocating tasks to nodes in a CADES that follows the DFT4FTT guidelines, and analyze the potential benefits of such approach both qualitatively and quantitatively. In section III, based on our research, we have outlined, in a qualitative manner, the potential benefits of using a DRL-based approach in a CADES. With our solution design, implementation and experiments presented in sections IV and V we have shown that:

- Our DRL agent, combined with an intermediate performing heuristic (the FF), can achieve similar average occupancy ratios to the best performing heuristic (the FFD).
- Our DRL agent can be used in combination with the fastest heuristic (the NF), yielding better average occupancy ratios than two popular heuristics (the NF and FF) and getting close to the best performing one (FFD).
- Our latency comparison supports our hypothesis that the agent's inference latency might be significantly lower than the best performing heuristic's latency (which requires pre-sorting the inputs).

Further work in this area could be targeted at modelling more complex operational contexts closer to real-life applications. On the one hand, this could be achieved by including a wider range of variables in the state information consumed by the DRL-agent. Examples of these additional variables are: the time required by each task to complete its execution, the number of nodes available (which could change over time due to node failures or incorporations), or a varying computational capacity for the nodes (which could vary over time due to, for example, some nodes being partially busy executing a task that cannot be reallocated). On the other hand, more complex operational contexts could also be factored in the reward function that defines the allocation criteria. For instance, the cost of reallocating a task could be factored in the reward function, which would give priority to tasks allocations that do not require moving a large number of tasks from one node to another.

We are convinced that the full potential of DRL-based configurations search can be realized when modelling more complex operational contexts such as the ones just described. The more complex the problem is, the harder it is to find heuristics or search strategies that deliver good results in a timely manner, and that is precisely where a versatile approach such as using a DRL-trained policy can have a greater impact.

VIII. Publications related to this work

R. Rotaeche, A. Ballesteros and J. Proenza, *Exploring the use of Deep Reinforcement Learning to allocate tasks in Critical Adaptive Distributed Embedded Systems*. Published in the WIP (work in progress) papers section of the 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2021, pp. 01-04, doi: 10.1109/ETFA45728.2021.9613409. Presented to the audience of this online conference by the first author.

References

[1] Google OR-Tools for optimization - The Bin-Packing Problem.

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015.

[3] A. Ballesteros, J. Proenza, M. Barranco, L. Almeida, and P. Palmer. First Description of a Self-Reconfigurable Infrastructure for Critical Adaptive Embedded Systems. Technical report, 2019.

[4] C. W. Barrett. SMT solvers: Theory and practice. 2008.

[5] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *5th Int. Conf. on Learning Representations, ICLR 2017*, 2019.

[6] R. David et al. TensorFlow Lite Micro: Embedded Machine Learning on TinyML Systems. 2020.

[7] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[8] Google Research. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *Network: Comp. in Neural Sys.*, 2015.

[9] H. Hu, X. Zhang, X. Yan, L. Wang, and Y. Xu. Solving a new 3D bin packing problem with deep reinforcement learning method. 2017.

[10] D. S. Johnson. Near-Optimal Bin Packing Algorithms. *Thesis*, 1973.

[11] B. Kleinberg, Y. Li, and Y. Yuan. An alternative view: When does SGD escape local minima? In *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[12] R. Korf. A new algorithm for optimal bin packing. pages 731–726, 01 2002.

[13] P. R. Kumar. A survey of some results in stochastic adaptive control. *SIAM Journal of Control and Optimization*, 23:329–380, 1985.

[14] J.-C. Laprie, A. Avizienis, and B. Randell. Fundamental Concepts of Dependability. 1145:7–12, 2001.

[15] J. Lin, W. M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han. MCUNet: Tiny Deep Learning on IoT Devices. (NeurIPS), 2020.

[16] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46 – 61, 1973.

[17] S. Sheng, P. Chen, Z. Chen, L. Wu, and Y. Yao. Deep reinforcement learning-based task scheduling in IOT edge computing. *Sensors*, 2021.

[18] A. Sherstinsky. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *CoRR*, abs/1808.03314, 2018.

[19] R. Solozabal, J. Ceberio, A. Sanchoyerto, L. Zabala, B. Blanco, and F. Liberal. Virtual Network Function Placement Optimization with Deep Reinforcement Learning. *IEEE J-SAC*, 2020.

[20] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*, volume 16. 2005.

[21] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. *Advances in Neural Information Processing Systems*, 2015.

[22] Z. Xu, Y. Wang, J. Tang, J. Wang, and M. C. Gursoy. A deep reinforcement learning based framework for power-efficient resource allocation in cloud RANs. In *IEEE Int. Conf. on Comms. (ICC)*, 2017.