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

Ramón Rotaeche, Alberto Ballesteros and Julián Proenza
*DMI - Universitat Illes Balears, Palma, Spain*
{ramon.rotaeche1, a.ballesteros, julian.proenza}@uib.es

***Abstract*—Critical Adaptive Distributed Embedded Systems (CADES) must carry out a set of funcionalities while fulfilling their associated real-time and dependability requirements. Moreover, they must be able to reconfigure themselves in a bounded time as the operational context changes. Finding a proper configuration can be non-trivial and time-consuming. Several studies have proposed Deep Reinforcement Learning (DRL) approaches to solve combinatorial optimization problems. In this paper, we explore the application of such approaches to CADES by solving a simple tasks allocation problem using DRL and comparing the results with three popular heuristics. The results show that DRL beats two of them and gets very close to the third, while requiring significantly less time to generate a solution.**

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

## I. Introduction

Distributed Embedded Systems (DES) are constituted by a set of interconnected *nodes* executing *tasks* in order to achieve some common goal. Critical Adaptive Distributed Embedded Systems (CADES) are DES that must be reliable and hard real-time, and that must automatically reconfigure themselves to adequate its operation as the *operational context* changes. The operational context encompasses both: (1) *the operational requirements*, i.e. what the system must do (e.g. tasks that must be executed, real-time guarantees); and (2) the operational conditions, i.e. the environment and the system itself (which can change due to faults). This means that, at each point in time, CADES should be able to find, in a short and bounded time, the *system configuration* that works best for the current operational context.

The ongoing Dynamic Fault Tolerance for Flexible Time-Triggered Ethernet (DFT4FTT) project [1], aims at providing a complete infrastructure for building CADES. In DFT4FTT, a privilege node called Node Manager (NM) is responsible for continuously monitoring the operational context and, if a change jeopardizes the correct operation of the system, searching and applying a new configuration in a timely manner.

As part of the DFT4FTT project, we are analyzing the potential of using DRL-based methods to find system configurations. A key element of these configurations is the tasks allocation, i.e. the distribution of tasks into the available nodes. Finding a suitable tasks allocation can be time-consuming and non-trivial, since it might be an NP-hard combinatorial optimization problem. As discussed in section IV, several studies propose DRL approaches to tackle these types of problems. In this paper, we build on those studies to present a solution that illustrates the potential of using such approaches in CADES. We present a model capable of learning to allocate a variable length list of tasks with variable cost. As discussed later, some aspects of our problem formulation and approach differ from previous studies, as well as specific considerations to its application in CADES.

The term DRL refers to the set of Reinforcement Learning (RL) techniques that make use of deep neural networks (DNNs). Reinforcement Learning (RL) is the subfield 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* [2].

The RL problem is formalized using a decision process (Fig. 1), 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 $S_{t+1}$.

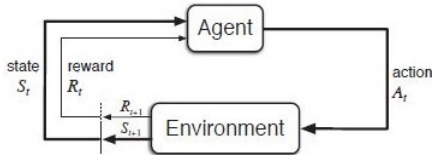


Fig. 1. The agent–environment interaction in a RL decision process [2]

The mechanism that the agent uses to decide which action to take is called *policy*. A policy is a mapping from states to probabilities of selecting each possible action [2], which we denote by $\pi(A \mid S)$. In some RL approaches, the policy is a *parametrized policy*, meaning that such mapping between states and action probabilities is a parametrized 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 $\theta$, are modified based on the reward obtained in the interactions with the environment, using techniques like the *policy gradient method*, which we use in this paper.

One way to parametrize the policy that is followed by the agent is with a DNN, since a DNN is effectively a function characterized by its architecture and parameters $\theta$, that takes

an input (in this case, the state) and produces an output (in this case, the probability of taking each possible action).

When compared to solvers and heuristic based algorithms, we consider that DRL has several advantages that make it a worth-investigating option for CADES. Namely:

- DRL methods have proved to be near as good or even better than many popular heuristics in solving different combinatorial optimization tasks [3], [4].
- 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 specially relevant when using high-dimensional states 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 CADES with real-time requirements.
- Recent developments in Tiny ML [5], [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 might be a good solution for resource hungry CADES.

As said earlier, in this paper we illustrate the potential of using DRL approaches in CADES. We do so by presenting a DRL-based solution for tasks allocation in section II, and comparing the results with heuristic-based solutions in section III. Finally, in section IV, we discuss our approach's background and related studies.

## II. Problem formulation and approach

In our tasks allocation problem, a *state* is a set of tasks, each of them with a cost. The system has a set of nodes, each of them with the same capacity $C$. A task's cost represents the node's capacity required to run that task. The agent must allocate tasks to nodes so that no node receives more tasks than it can handle, while trying to minimize the total number of *active nodes* (i.e. nodes that receive at least one task). A specific mapping of tasks to nodes is therefore the *action* in the RL framework.

We have chosen to minimize the number of active nodes in this initial study because it results in us essentially trying to solve a version of the well known bin-packing problem [7], meaning that we can use several well studied algorithms to compare with our results. In addition, minimizing the number of active nodes could have useful applications, such as reducing the total energy consumption of the system or increasing the likelihood that there will be enough free capacity in the event of receiving a new task with a high cost.

### A. 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.

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, once it is trained, can be applied to sets of tasks of different sizes.

### B. Input - output representations approach

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

The output (i.e. the action) of our parametrized policy is the tasks allocation, which is represented with the allocation order $A$. $A$ is a sequence of $n$ integers $a_i \in [1, n]$, $i \in [1, n]$, where $a_i$ represents the position of the task in the input sequence $S$ (Fig. 2). Tasks are allocated in the order given by $A$ following a Next-Fit (NF) rule: 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. Therefore, the allocation order sequence implicitly dictates how tasks are grouped together.

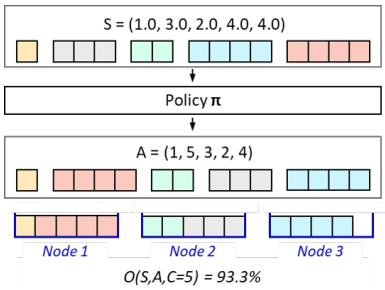


Fig. 2. Input-output representation

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 [3], restricting as much as possible the equivalence class for the outputs leads to better results. Moreover, the chosen representation together with the NF rule ensure that all generated solutions comply with the nodes maximum capacity constraint.

### C. Policy parametrization approach

We use a *pointer network* [8] to parametrize our policy $\pi(A \mid S)$. 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. The allocation order $A$ 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 pointer-network encoder-decoder architecture means that at each step $i$, $a_i$ is a function of the input sequence $S$ as well as of the previous outputs $a_1, ..., a_{i-1}$. If we define $p(\pi(a_i) \mid a_1, ..., a_{i-1}, S)$ as the probability that was assigned by $\pi$ to the sampled value for $a_i$, the probability of obtaining an specific allocation order A given a state S can be calculated using the chain rule:

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

Our exact DNN architecture follows that proposed by [3], which includes slights modifications to the original pointer network architecture.

For inference, the policy becomes a deterministic policy $\pi(S)$ by greedy selecting the $a_i$ with highest probability (rather than sampling from the distribution as we do during training).

### D. Training approach

We use the so-called REINFORCE stochastic gradient ascent algorithm, based on the policy gradient theorem [2], to converge towards a policy that yields the maximum reward. The algorithm progressively updates the parameters $\theta$ of the DNN on each training step $t$ as:

$$\theta_{t+1} = \theta_t + \alpha * (O(A_t, S_t) - b(S_t)) \nabla_\theta p(\pi_t(A_t \mid S_t)) \quad (2)$$

$O(A_t, S_t)$ is the avg. occupancy ratio obtained at training step $t$. The baseline $b(S_t)$ is an approximation of the expected reward for the state $S_t$ using the policy $\pi$ with the parameters at step $t$. We model it using another DNN, typically known as *critic*, which is also trained on each step $t$. The critic's architecture and the training algorithm is based on [3], where more detail can be found. $\alpha$ is simply a scalar known as the learning rate.

## III. Experiments

We have conducted experiments to evaluate how well the policy learns to generate allocations with high $O$, as well as some basic tests to evaluate the inference latency. For this preliminary study, we have considered two different problem conditions (Fig. 3).

| Problem label | # of tasks | Min. task cost | Max. task cost | Nodes capacity |
|---|---|---|---|---|
| Problem 1 | 24 | 1 | 6 | 8 |
| Problem 2 | 40 | 3 | 8 | 10 |

Fig. 3. Problem conditions in our experiments. Tasks costs are sampled from the uniform distribution over the interval [Min. task. cost, Max. task. cost]

For comparison purposes, we also evaluate the avg. occupancy ratio obtained with three popular heuristics [7] used to solve this combinatorial optimization problem.

- *Next-Fi* (NF): Tasks in the input set are considered in an arbitrary order. The first node is "opened" 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): Tasks in the input set are considered in an arbitrary order. 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 "opened" unless they are completely full. On each allocation step all "opened" 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.

NF and FF do not require to process the full input set prior to the start of the allocation, making them faster than the FFD. NF is the fastest as it does not have to check all "open" nodes. NF was chosen in order to have a "lower bound" for the avg. occupancy ratio. As explained earlier, in our outputs representation approach, tasks are allocated according to the output $A$ following a NF heuristic. Therefore, the NF avg. occupancy ratio is what our policy would score if it does not manage to learn at all.

Our DNN's hidden dimension size (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 CADES, we wanted to test the performance of a lighter DNN.

For each of the experiments, we have trained 3 models during 10,000 training steps (Fig. 4) 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. The batch size was 128. On every training step a new batch of 128 randomly sampled sets of tasks were created.


Fig. 4. Problem 2: batch average occupancy ratio (%) after each training step

No hyper-parameter tuning was done, and no decoding strategies other than greedy decoding were explored. Hyper-parameter tuning and more sophisticated decoding strategies (e.g. beam search or those proposed by [3]) could potentially yield better results. Nevertheless, as shown in Fig. 5 our agent generates allocations with a higher avg. occupancy ratio than

the NF (as expected) and the FF heuristics, and gets close to the FFD performance.

| **Problem label** | ***Average occupancy ratio %*** | | | |
|---|---|---|---|---|
| | DRL Agent | Heuristics | | |
| | | NF | FF | FFD |
| Problem 1 | 92.3 | 77.5 | 89.2 | 93.8 |
| Problem 2 | 89.1 | 73.9 | 85.8 | 89.4 |

Fig. 5. Avg. occupancy ratio (%) comparison between our trained DRL-agent and the selected heuristics

The FFD heuristic requires a pre-sorting of the input tasks, which impacts the time required to generate a solution. Given an input set of tasks, we have compared the time it takes to our agent and to the FFD algorithm to generate a solution.

We have implemented both the DNN inference (including the final process of allocating tasks following a NF approach based on the DNN output) and the FFD algorithm as Tensorflow's [9] 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. For the Problem 2 conditions, on average, it takes *18 ms* to generate a solution using our DNN-based agent, while it takes *68 ms* with the FFD algorithm. 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. However, 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.

## IV. Related work

In this paper, we have discussed a solution to a combinatorial optimization problem using a DNN. The general term for this is approach is *neural combinatorial optimization*. A significant contribution to this area was made by [8], with the introduction of 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 (eliminating the need for optimal solution examples) was proposed by [3]. Our DNN architecture and DRL approach is based on theirs. However, they use it to solve other type of combinatorial optimization problems.

On the grounds of [3], several studies have tackled combinatorial optimization problems related with resource allocation [4], [10], [11], [12]. Perhaps, the most related to the problem solved in this paper are [12] and [4]. In [12], 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. In addition, the DNN architecture used is different, the problem constraints are different and so it is the strategy to enforce such constraints. In [4] they solve a 3D packing problem by minimizing the surface required to pack a set of items. Their approach inspired our choice to represent the actions space as the order in which tasks are allocated.

To the best of our knowledge, no prior studies have focused on exploring the use of DRL to find system configurations for CADES specifically, which implies taking into account aspects such as the inference latency. In addition, none have used a metric like the avg. occupancy ratio as the reward signal, which has proved to yield good results.

## V. Conclusions and further work

With the tasks allocation example, we have illustrated the potential of using DRL-based methods to find system configurations in CADES.

We have shown that our DRL yields better results than two popular fast heuristics. In addition, regarding the heuristic that our agent does not beat, the latency comparison supports our hypothesis that the agent's inference latency might be significantly lower than the heuristic's latency (which requires pre-sorting the inputs).

We are convinced that the full potential of DRL-based configurations search can be realized in more complex problems, where the operational context includes a wider range of variables (e.g. cost of reallocating a given task, task completion time, variable node capacity) and the reward function takes into account more complex dynamics. In those problems, finding heuristics or search strategies that deliver good results in a timely manner can be a really hard task, and that is where a DRL-trained policy can make a big difference. It is our intention to further explore this idea.

In addition to more complex operational contexts, it is our intention to experiment with other reward functions that favour fault-tolerant configurations, so as to analyze the potential of using DRL agents as a fault tolerance mechanism.

## References


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

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2005, vol. 16, no. 1.

[3] 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.

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

[5] J. Lin, W. M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, "MCUNet: Tiny Deep Learning on IoT Devices," no. NeurIPS, 2020.

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

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

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

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

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

[11] 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.

[12] 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.