FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Deep Reinforcement Learning for Real-time Scheduling

Bruno Daniel Durães Pereira Mendes

DISSERTATION

Mestrado em Engenharia Informática e Computação

Supervisor: Pedro F. Souto

Co-Supervisor: Pedro C. Diniz

July 26, 2024

Deep Reinforcement Learning for Real-time Scheduling

Bruno Daniel Durães Pereira Mendes

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Liliana Ferreira
External Examiner: Ali Awan
Supervisor: Pedro Souto

July 26, 2024

Abstract

Real-time systems are essential for the correct and safe operations of many devices that permeate our lives, from sophisticated wristwatches to automobiles and aeroplanes. To support their complex operation, they include control software that relies on scheduling multiple computational tasks. To make their operation possible, systems have been using scheduling algorithms supporting multi-programming since the 1970s, allowing complex and interdependent computational tasks to be deployed while meeting hard real-time constraints.

With the advent of complex, highly capable, resource-sharing real-time systems, it is no longer efficient to consider the same criticality level for all tasks, as traditional algorithms did. Using spare processor capacity for tasks of different criticality lowers power consumption and space requirements. As such, mixed-critical systems are an active area of research and depend highly on the systems designers' subjectivity in estimating low-criticality tasks' worst-case execution times.

In this work, we explored the feasibility of modern deep reinforcement learning (DRL) algorithms in developing intelligent scheduling algorithms geared towards mixed-critical systems to adjust the tasks' parameters based on observed behaviour at runtime. We simulated a real-time system and studied the impact of the agent's actions on the overall schedulability of the system, through a set of simulations of different task sets, generated with industry insight.

Our preliminary and limited experimental results reveal that the explored methodology based on a DRL scheduling agent can significantly reduce the number of task cancellations in most simulations in a reasonable amount of time, suggesting that reinforcement learning is a valid approach to improve the schedulability of mixed-critical real-time systems.

Keywords: deep reinforcement learning, scheduling algorithms, real-time computing, real-time operating systems

Resumo

Sistemas em tempo real são essenciais na sociedade, enquanto peça fundamental de aviões comerciais ou carros. Desde os anos 1970, a multiprogramação nestes sistemas é possível devido a algoritmos clássicos de escalonamento que permitem a execução de tarefas complexas e interdependentes, cumprindo restrições temporais rígidas de tempo real.

Com o advento de sistemas em tempo real mais complexos, poderosos e capazes de partilhar recursos, não é eficiente considerar que todas as tarefas são igualmente críticas. Na verdade, alocar capacidade de processamento não usada a tarefas de criticidades diferentes contribui para uma diminuição do consumo de energia e do espaço requerido. Sistemas de criticidade mista são uma área de pesquisa ativa e dependem em boa parte da subjetividade dos designers de sistema no momento da estimação do tempo de execução no pior caso das tarefas de baixa criticidade.

Providos de algoritmos modernos de aprendizagem por reforço profunda, exploramos a viabilidade da introdução de um agente inteligente em sistemas de criticidade mista com vista a ajustar os parâmetros das tarefas, com base no comportamento observado em tempo de execução. Simulamos um sistema em tempo real e estudamos o impacto das ações do agente na escalonabilidade do sistema, através de uma série de simulações de escalonamento de conjuntos de tarefas gerados com base em artigos da indústria.

O agente é capaz de reduzir de forma significativa o número de cancelamentos de tarefas num espaço de tempo bastante razoável, o que sugere que aprendizagem por reforço é uma abordagem válida para melhorar a escalonabilidade de sistemas de tempo real de criticidade mista.

Palavras-chave: deep reinforcement learning, algoritmos de escalonamento, computação em tempo real, sistemas operativos em tempo real

Agradecimentos

Começo por demonstrar a minha gratidão aos meus orientadores: ao professor Pedro Souto, pelo pronto e fenomenal apoio, e ao professor Pedro Diniz, pela amigável disponibilidade.

Este trabalho marca o fim de um percurso longo de cinco anos, pelo que não podia deixar de mencionar também algumas pessoas importantes que tornaram esta jornada possível:

os meus pais, quais faróis, que fomentaram desde cedo em mim um incansável espírito crítico

os meus avós, para sempre a minha segunda casa

o José, o Fernando e demais amigos, que fizeram da minha vida quotidiana na lindíssima cidade do Porto (e fora dela) uma experiência coletiva de qualidade

A todos estes, um genuíno obrigado.

À Faculdade de Engenharia e suas gentes, ao negro impecável das capas e cinzento aborrecido dos corredores que tantas vezes foram azul e branco e mil cores no meu coração, um saudoso obrigado.

Bruno Mendes

"Beauty is no quality in things themselves:
It exists merely in the mind which contemplates them."

David Hume

Contents

List of Figures

List of Tables

Abbreviations

ACET	Average-case Execution Time
ANN	Artificial Neural Network
API	Application Programming Interface
BCET	Best-case Execution Time
DDPG	Deep Deterministic Policy Gradient
DL	Deep Learning
DRL	Deep Reinforcement Learning
DQN	Deep Q-Network
IoT	Internet-of-Things
MC	Mixed-critical
MDP	Markov Decision Process
ML	Machine Learning
NN	Neural Network
DNN	Deep Neural Network
OS	Operating System
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
RM	Rate Monotonic
RT	Real-time
RTOS	Real-time Operating System
WCET	Worst-case Execution Time

Chapter 1

Introduction

This chapter aims to contextualize the problem (Section 1.1), present our motivation for pursuing this research (Section 1.2) and enumerate our goals (Section 1.3).

1.1 Context

Embedded systems have been an industry of hundreds of millions of dollars since the late 1970s [27]. Their microcontroller-based architecture allows for significant cost reductions both in production [1] and deployment (e.g. regarding energy consumption). In the current Internet-of-Things (IoT) era, embedded systems play a pivotal role in industrial and healthcare settings, gathering and transmitting data whose analysis leads to more efficient processes and decision-making.

However, the applications of embedded systems are not limited to simple data collection and integration. Perhaps the most well known and developed field where embedded computing leverages intelligent physical control of devices is the avionics industry. The availability of autopilot and multiple self-controlling systems has led to a significant increase in the safety of passengers and the commodity of pilots: in the first 20 years of the 21st century, the number of accidents with aeroplanes in the U.S. decreased by about 60% [23].

The design of life-critical systems found in modern aeroplanes and cars requires careful consideration of timing and correctness. For instance, the failure or delay in executing the process responsible for triggering a car's brakes in the event of an imminent crash may be catastrophic. This motivated the advent of real-time computing, presented in Section 2.1, as a set of analysis tools to be performed at design time that prove that no task can ever be denied of a time budget that it needs.

Modern embedded systems are complex and often fail to be put in a single criticality level. One may be surprised to find that recent commercial cars boost entertainment systems that are more

[1] Here, we are assuming that the microcontroller may be application-tailored and that it is less powerful than a general-purpose microprocessor with added peripherals.

capable than general-purpose computers of just a few years ago [2], and yet do not compromise on delivering an ample set of driving assistance. With this heterogeneous nature in mind, mixed-critical systems, studied in Section 3.1.1, have gained traction in academia and the industry [5].

1.2 Motivation

Mixed-critical systems are an active area of research [4, 5]. Most complex embedded systems found in the avionics and automotive industries are gradually evolving to mixed-critical designs "in order to meet stringent non-functional requirements relating to cost, space, weight, heat generation and power consumption" [10].

In mixed-critical systems, the time allocated to low-criticality tasks is supposed to be deemed safe but is not subject to mathematical proof. As such, there is room for subjectivity. While reducing the time budget of a low-criticality task may allow for scheduling more tasks in the same time window and/or computational equipment, it may also be overly optimistic and lead to the task overrunning its given time. The action taken in such cases is highly dependent on the context and the subjectivity of the system designers but typically involves at least killing the task or even dispensing with the execution of all low-criticality tasks, effectively hindering the quality of service for some time. In contrast, being overly pessimistic when estimating the execution times of tasks implies an inefficient use of resources and, ultimately, loss of cost-cutting opportunities.

The novel heuristics for scheduling mixed-critical systems, analysed in Section 3.1.2, depend on the aforementioned worst-case execution time of tasks, provided *a priori*. This offline-only design fails to account for the actual behaviour of those tasks in the system at runtime, often leading to execution times very different to the estimated ones.

In the meantime, artificial intelligence has emerged from years of stagnation and has seen extraordinary success recently in addressing many practical problems. We are particularly amazed by the appearance of autonomous agents, capable of taking precise actions and cognisant of the environment in which they reside, whose inner workings we present in Section 3.2.2.

The motivation for this research is to study the feasibility of introducing an intelligent agent in a mixed-critical real-time system responsible for adjusting at runtime the parameters of the tasks based on past behaviour in order to reduce costs, e.g. by reducing the number of cores in the system, or improve quality of service, by avoiding mode changes and task cancellations. This idea comes from the fact that important runtime synergies may be unavailable to engineers at design time. For example, in a system with a shared cache, if task B uses the same memory region as task A and is often activated immediately after task A, the observed execution time of task B will likely be significantly lower than the expected worst-case time due to the effect of a cache hit.

As of the time of writing, we are unaware of any machine learning-based online real-time scheduling policy and hope this research is a valuable contribution to the field.

[2] In December 2021, Tesla installed AMD RDNA 2-based graphics-processing units in their Model 3 and Model Y. According to Tesla, they provide up to 10 teraflops of compute power [22].

1.3 Objectives

Throughout this work, we seek to answer the following questions:

1. **How can we use state-of-the-art deep reinforcement learning (DRL) to adjust a real-time system's schedule in runtime?** How to model the problem as a Markov Decision Process (explained in Section 2.2.2)? What actions can we take that may trigger a positive rearrangement of the system scheduling?

2. **How do different DRL models compare, regarding execution time and overall impact?** What are the computational requirements and impact on the schedulability of the system of each model?

3. **Is the time allocated to the DRL task worth it?** As we shall allocate time for the algorithm to run periodically, we lose time possibly useful for application-related tasks. Are the possible gains worth the investment?

For this to be possible, we propose to simulate a real-time system to serve as a testbed for experimenting with a DRL agent. We describe our methodology in detail in Chapter 4.

Before that, in Chapter 2, we review fundamental concepts in real-time computing and machine learning. The reader familiar with these fields may skip this chapter.

In Chapter 3, we present recent work in mixed-critical systems and deep reinforcement learning algorithms. Although not mandatory, we appreciate that this chapter is not overlooked for a complete picture of this work.

Chapter 4 describes the problem in hand with more detail, as well as our proposed methodological approach.

After that, in Chapter 5, we explain the inner working details of the tasks runtime simulator and the agent that resides in that environment.

Chapter 6 explains how our test scenarios are generated and analyses the impact of the developed agent in each of them.

Finally, Chapter 7 summarizes the work developed and outlines future work.

Chapter 2

Background

To help readers grasp the concepts behind our research, this chapter reviews relevant theory in real-time computing and machine learning. Section 2.1 introduces the timing restrictions in real-time systems and presents the scheduling classic algorithms honouring them. Then, Section 2.2 reviews the taxonomy of machine learning and studies its techniques and conventions. At last, Section 2.3 closes the chapter with a summary of reviewed concepts.

2.1 Real-time Computing

Real-time (RT) Computing is a broad term designating software and hardware under timing constraints. The correctness of a computation in a real-time system depends not only on its logical correctness but also on the time at which its results are produced [51].

Real-time systems comprise a set of mostly *periodic* tasks with deadlines by which they must be invoked and executed. Based on their criticality, tasks may be put into three categories:

- *Hard-constrained tasks* must be executed before their deadline to avoid catastrophic consequences. For example, controlling a car's brakes has a hard deadline.
- *Firm-contrained tasks* produce results that are only useful if they execute before their deadline. For example, periodically monitoring a car's speed and writing it to a database will not be useful if access to the database takes too long and speed changes quickly.
- *Soft-contrained tasks* have decreased utility as the time after their deadline passes. For example, an aperiodic task triggered by the user changing channels in a TV Box's command will still be useful, although annoying, if it lags considerably.

With that in mind, real-time systems must be carefully designed to be reliable and *predictable* even in a worst-case scenario. In fact, "the architecture of each node, the communication subsystem, the operating system, and the programming languages have to support the notion of guarantee at all levels of abstraction" [51].

Safety-critical RT systems undergo strict tests and must obey a series of best practices to go into production. For instance, many are programmed in *MISRA C*, a set of rules for the C programming language, initially aimed at the automotive industry but later standardized, that provides "world-leading best practice guidelines for the safe and secure application of both embedded control systems and standalone software" [6].

2.1.1 Task Scheduling

In real-time systems, the timeliness of tasks' completion depends on the algorithms used for task scheduling. They often characterize a task by:

- a worst-case execution time C (interchangeably referred to as $WCET$)
- a period T, for periodic tasks, or a minimum inter-arrival time, for sporadic tasks
- a first activation instant ϕ, for periodic tasks
- a deadline D

Task scheduling algorithms work under certain assumptions to guarantee that hard real-time constraints are met.

Some algorithms assume that tasks are *preemptable* at all times, i.e. they can be suspended if a task of higher priority must run. This may not be true if the tasks share resources. Suppose task A of lower priority holds a lock of a critical section that task B of higher priority demands, and task B is *activated*. In that case, it must be *blocked* as long as A holds the lock, which effectively constitutes a priority inversion problem. To maximize schedulability under such circumstances, priority inheritance protocols, such as the priority ceiling protocol, raise the priority of a task to the highest priority level of all the tasks it may block during the execution of a critical section [50].

Tasks may also be assumed to be independent, meaning that their activation does not require the completion of other tasks. If not, we must use *task graphs* to enable partial ordering on executing tasks [19].

2.1.1.1 Single-core scheduling

Single-core scheduling algorithms date back to the work of Liu and Layland [35]. They studied *static* scheduling, performed offline, provided that all tasks maintain their timing requirements throughout the program execution. Rate-monotonic (RM) and Deadline-monotonic (DM) are offline scheduling algorithms that consider the period and the deadline of tasks, respectively, for determining fixed priorities.

Dynamic or online scheduling adapts better to varying execution times and activation instants of tasks. The scheduler may be invoked on each clock tick [1] to determine the task to execute, usually with the earliest deadline, as given, for instance, by the Earliest Deadline First (EDF)

[1] In fact, the scheduler only needs to run upon arrival or termination of tasks, assuming that tasks are not suspended.

algorithm, a well-known algorithm of this class. This means that the relative priorities of two tasks can change at runtime, contrary to RM or DM. For instance, consider two tasks with D=10 and D=20, respectively: if a job of the first task arrives 15 time units after an activation of the second task, under EDF, the second job continues executing since its deadline is 5 time units ahead; under DM, the first job would interrupt the second.

EDF is known to be optimal for uniprocessor systems [35], but requires a more complex implementation and analysis, as presented in Section 2.1.1.2.

Instead of prioritizing by the deadline, Least Laxity First (LLF) and First Come, First Served (FCFS) prioritize by laxity (the difference between the absolute deadline and the remaining execution time) and time of arrival, respectively. They may be more straightforward, although less optimal, alternatives to EDF.

2.1.1.2 Schedulability analysis

A schedule may be *feasible* if and only if the CPU utilization U is less than 1 (Equation (2.1)):

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.1}$$

A schedule is guaranteed to be feasible if the least upper bound of CPU utilization is respected (Equation (2.2)).

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1) \tag{2.2}$$

This is, however, a sufficient but not necessary condition. In a rate-monotonic schedule, one can prove the schedulability of a system by assessing for all tasks that their worst-case response time, occurring when they are activated simultaneously with all higher-priority tasks (Equation (2.3)), is less than their period [28].

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \tag{2.3}$$

In EDF scheduling, the maximum execution time requirement for all tasks that have both their arrival time and their deadline in a contiguous interval of length t, or the *processor demand function* (Equation (2.4)), compared with t, is a sufficient and necessary test but must be done for several instances t.

$$h(t) = \sum_{i=1}^{n} \max\left(0, 1 + \left\lfloor \frac{t - D_i}{T_i} \right\rfloor\right) \cdot C_j \tag{2.4}$$

The *Quick convergence Processor-demand analysis* (QPA) [14], as illustrated in Figure 2.1, iteratively determines all t needed for the test. It makes $t = min(L_a, L_b)$, where L_a is defined by Equation (2.5) and L_b is the length of the synchronous busy period of the task set: the maximum length of any possible busy processor period in any schedule. If $h(t) < t$, it assigns $h(t)$ to t

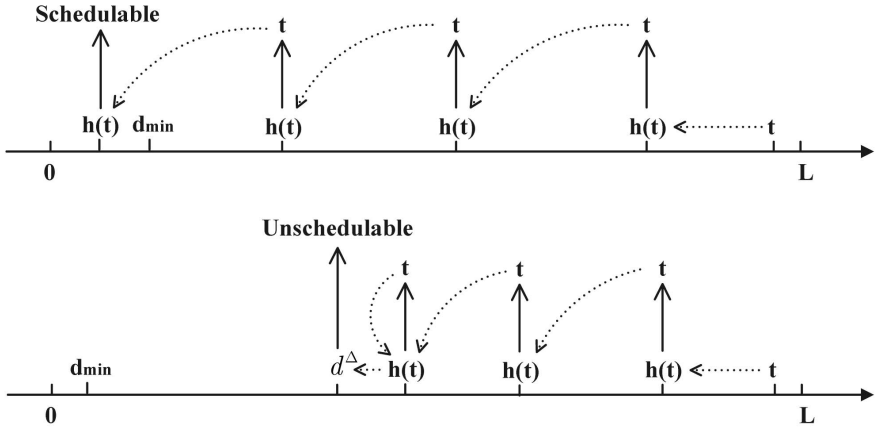


Figure 2.1: The QPA algorithm. [14]

and repeats. The task set is determined to be schedulable if $h(t)$ becomes less than the smallest deadline.

$$L_a = max\left(D_1,...,D_n, \max_{1\leq i\leq n}(T_i - D_i)\cdot\frac{U}{1-U}\right) \tag{2.5}$$

2.1.1.3 Multi-core scheduling

With the rapid development of multicore processors, even in the embedded systems field, to reduce power consumption while allowing the execution of highly parallel computational jobs, single-core scheduling no longer suffices. The challenges of task scheduling in a multicore system are vast: the search space is ample, and thus the problem is NP-hard.

Multicore scheduling algorithms may be *global*, if a single task queue for all processors is used and any processor may be assigned to execute any task, or *partitioned*, if each task is assigned to a single processor. An overview of standard scheduling approaches, including multi-core compatible ones, is presented in Figure 2.2.

Partitioned approaches, such as P-EDF and P-FP, derive from their single-core counterparts only needing to assign sets of tasks to cores as an extra first step, which is an instance of the *bin packing problem*, for which several heuristics exist [26].

Global approaches, such as G-EDF, are different in that the migration of tasks may occur, i.e. a task may be resumed in a different core after exiting a preemption state.

2.1.1.4 Performance metrics

The effectiveness of multi-core scheduling approaches may be determined using empirical measures, such as the percentage of tasksets found to be schedulable, or the number of migrations in preemptions obtained in a simulation [9].

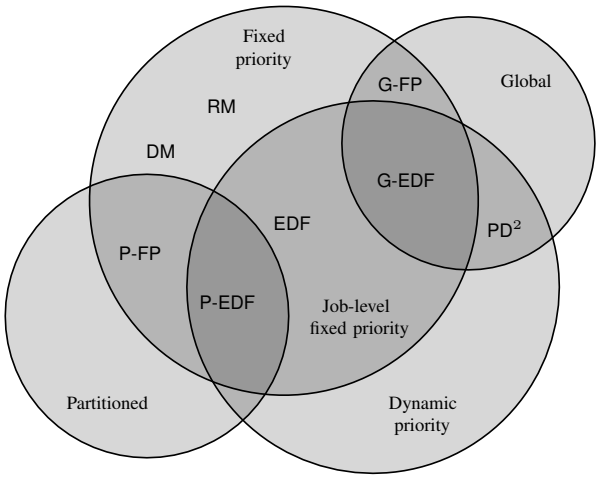


Figure 2.2: Venn diagram of single and multi-core scheduling policies. [26]

The utilization bound, already discussed, or an *approximation ratio* to an optimal algorithm are alternative performance metrics. Equation (2.6) represents the approximation ratio $\mathscr{R}$ of an algorithm A compared to the optimal solution O regarding the number of processors used M.

$$\mathscr{R} = \frac{M_A}{M_O} \tag{2.6}$$

2.1.2 Real-time Operating Systems

A real-time operating system (RTOS) is, by definition, an operating system that supports the operation of a real-time system, thus meeting both timing and logical correctness requirements. As such, one usually includes support for setting task priority levels and deterministic preemption.

As with any other operating system, however, it must also manage the hardware resources of the device in which it is deployed and provide a layer of abstraction from the hardware to the programmer in the form of an *application programming interface* (API).

Most contemporary RTOS, such as Zephyr OS or FreeRTOS, include some POSIX compliance. This ensures that input/output, memory sharing, timers, and inter-process communication are available under a familiar interface [12].

An RTOS strives to be minimal and provides little more than a small kernel capable of fast context switches [2] and interrupt handling [3]. Since many embedded systems do not integrate a Memory Management Unit (MMU), the separation between user and kernel space is often thin.

Full-blown, general-purpose operating systems such as Linux can also be adjusted to meet real-time requirements, ditching their usual round-robin scheduling policy with real-time patches

[2] A context switch involves all operations necessary for preserving the temporary state of a routine for later restoration. This involves pushing a set of registers to the stack depending on the system architecture.

[3] An interrupt handler attends to a traditionally asynchronous hardware event, e.g. implementing device driver functionality or a software trigger. It also involves context saving.

[47]. These are typically aimed at supporting the operation of mixed-critical systems and are less reliable for hard real-time systems compared to RTOS.

2.2 Machine Learning

Machine Learning is a field of Artificial Intelligence concerned with automatically detecting data patterns to predict future data or perform other kinds of decision-making under uncertainty [42]. Traditionally, one can identify three types of machine learning [17]:

- *Supervised learning* is the task of determining a classification (a categorical target) or regression (a numerical target) from labelled training data.
- *Unsupervised learning* is the task of discovering knowledge in unlabelled data, e.g. segregating data in clusters.
- *Reinforcement learning* is the task of teaching an agent how to behave in an environment so that a cumulative reward is maximized.

2.2.1 Supervised Learning

In *Supervised Learning*, the goal is to learn a mapping from an input x to an output y, also called a *model*.

$$y = f(x) \tag{2.7}$$

Considering x a labelled dataset, each x_i is a vector of features (or *attributes*). y_i is then a categorical or nominal vector characteristic that the model shall predict. For example, given the salary and average balance of a bank's customers, a model could be trained to predict whether they would be trustworthy for conceding loans.

The process of training a model is that of an algorithm minimizing a loss function between the predicted and the actual y values for a training *labelled* dataset, that is, one that provides y_i alongside x_i for each row of the dataset. The mean absolute error (Equation (2.8)) is an example of a loss function commonly used for regression tasks.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{2.8}$$

A good model can *generalize* and performs well when fed with new, unseen data.

The *bias-variance tradeoff* (Figure 2.3) is related to the fact that an increase in a model's complexity, despite reducing its erroneous assumptions and making it perform better on the training dataset - low bias -, has a chance of *overfitting* - high variance. An ML engineer would strive for a low variance and bias in an ideal scenario.

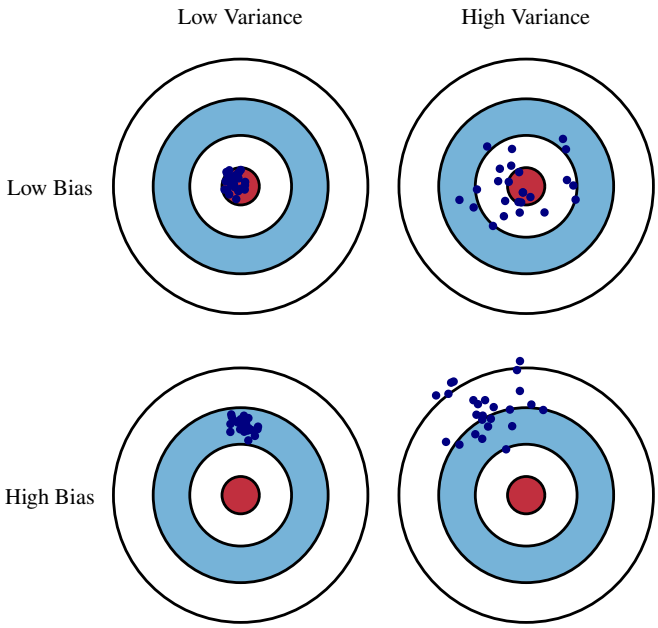


Figure 2.3: The bias-variance tradeoff, as illustrated in [16]. In the bulls-eye diagram, the centre represents a correct prediction by the model.

Typical algorithms for Supervised Learning include Decision Trees, Naïve Bayes and Support Vector Machines [8]. Artificial Neural Networks (ANN) is an alternative that will be further discussed in Section 2.2.3.

2.2.2 Reinforcement Learning

Reinforcement Learning (RL) is concerned with finding the optimal behaviour for an agent in an environment. Contrary to Supervised Learning, it is fed with unlabelled data - a batch of observations o, possibly partial descriptions of the environment's state s throughout time.

RL systems, modelled in Figure 2.4, empirically infer the dynamics of the environment through trial and error. After executing an action a, the agent receives a reward r that may hint whether the action was good or not. For example, the rewards for an agent training to escape a maze with traps, as illustrated in Figure 2.5, could be -100 after hitting a trap (red tile), -1 after moving to a neutral tile (white tile) and 1 for finding the exit door (green tile).

The goal of the agent is to find a policy π, a mapping of states to actions (Equation (2.9)), that maximizes some long-run measure of reinforcement [29]. In the maze scenario, it would be one that yields a path as short and trapless as possible, ending in an exit tile.

$$a_t \sim \pi_\theta(s_t) \tag{2.9}$$

RL problems often fit the definition of a Markov Decision Process (MDP), consisting of: [29]

- a set of states S

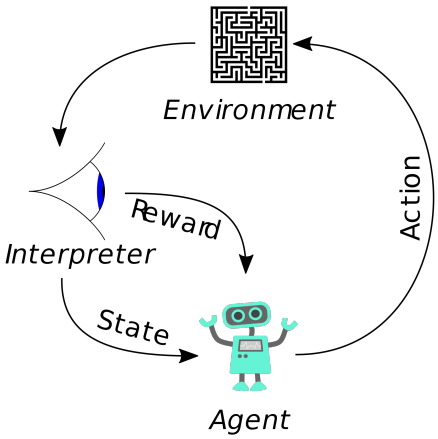


Figure 2.4: Agent-environment interaction loop. [39]

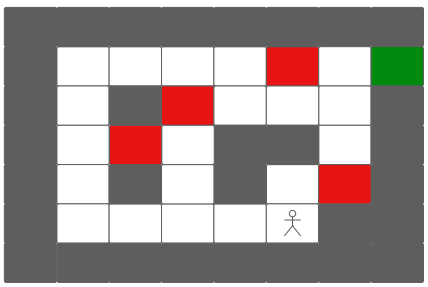


Figure 2.5: An agent in a maze. The action space is discrete: at a given time, it may go left, up, right or bottom, as long as the tile is not grey.

- a set of actions A
- a reward function $R : S \times A \rightarrow \mathbb{R}$ [4]
- a state transition function $T : S \times A \times S \rightarrow [0,1]$.

2.2.2.1 Tranining Dynamics

RL agents train in *episodes* or *trajectories* as long as some stopping criteria, such as a maximum number of episodes or success rate, are not met. An episode is simply a sequence of states and actions in the world.

An agent strives to maximize the cumulative reward gained during an episode. As such, it considers the *value* or *utility* of a state (Equation (2.10)) as the expected sum of rewards obtained during the following states, experienced in previous episodes. The discount factor γ, between 0 and 1, constrains the weight of rewards set far in the future.

$$V(s) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \tag{2.10}$$

The value of a state can be calculated using dynamic programming concepts. The Bellman Equation (Equation (2.11)), from 1953 [43], develops the above statement and considers the value of a state as the immediate reward plus the value of the following state s', reached after executing the best available action, and thus is fit for the RL domain. $T(s,a,s')$ represents the probability of transitioning to state s' after executing action a in state s.

[4] It is not always trivial to design a reward function without sufficient knowledge of the environment. While outside the scope of this dissertation, the reader might be interested to know that reward signals might be determined "via online gradient ascent, where the gradient is that of the high-level objective function" [56]. The high-level objective function in our problem would be the number of mode changes and task cancellations.

$$V(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s,a,s') V(s') \right) \tag{2.11}$$

The best policy is then the one that yields an action that maximizes the value function of a state. Since it depends on assessing the reward dynamics throughout the state-action space in reasonable computational time, an agent should balance *exploration* - executing a random action to get to know the environment - and *exploitation* - choosing an action previously shown to be the best. This tradeoff is extensively studied in multi-armed bandit problems.

2.2.2.2 Taxonomy

RL algorithms try to learn an optimal policy by manipulating it directly (policy-based) or deriving it from the value function (value-based). They may also be categorized as model-free or model-based according to their ability to predict state transitions and rewards, i.e., having a model of the environment.

Classical, value-based RL algorithms are *temporal-difference methods*, since updates are performed according to the difference in the measured future value and the estimated future value [11] of states. Exploration is typically performed using an $\varepsilon - greedy$ strategy: exploration is chosen if a random value between 0 and 1 falls below a current ε that will decrease with time, effectively favouring exploitation later on in the training process.

Q-learning, an example of value-based RL originated in a study on behavioural ecology [61], works by building a value function $Q(s,a)$ that is stored in a table of finite size $S \times A$ and updated iteratively with a learning rate α and new values calculated using the Bellman equation [5] (Equation (2.12)).

$$Q_{new}(s,a) \leftarrow (1-\alpha) \cdot Q(s,a) + \alpha \cdot \left(r + \gamma \cdot \max_{a'} Q(s',a') \right) \tag{2.12}$$

where s' is the state if one takes action a in state s and r is the reward by taking action a in state s.

The *REINFORCE* algorithms [62], sometimes called Vanilla Policy Gradient algorithms, are policy-based in that they make weight adjustments in their "connectionist networks" in a direction lying along the gradient of expected reinforcement. Equation (2.13) is the gradient ascent update rule of these algorithms, where θ are the policy parameters, α is the learning rate, and $\nabla_\theta J(\theta)$ is the gradient of the expected return with respect to the policy parameters.

$$\theta_{t+1} \leftarrow \theta_t + \alpha \nabla_\theta J(\theta) \tag{2.13}$$

Table 2.1 presents some standard RL algorithms. Algorithms utilizing Deep Learning techniques will be further explored in Section 3.2.2.

[5] The Bellman equation formulates the value of a decision problem recurrently in regards to the value after taking action and receiving some form of payoff. As such, its use depends on the context.

Approach	Category	Iteration	Example Algorithms
Q-Learning	Model-Free	Value	Q-learning, Deep Q Networks (DQN)
SARSA	Model-Free	Value	SARSA, Expected SARSA
REINFORCE	Model-Free	Policy	Vanilla Policy Gradient (REINFORCE)
DDPG	Model-Free	Policy	Deep Deterministic Policy Gradient (DDPG)
PPO	Model-Free	Policy	Proximal Policy Optimization (PPO)
A2C	Model-Free	Policy	Advantage Actor-Critic (A2C)
MCTS	Model-Based		Alpha Zero

Table 2.1: Common RL Algorithms.

2.2.3 Artificial Neural Networks

Closely resembling the human brain [20], *Artificial Neural Networks* (ANN) is a means of doing Machine Learning via densely interconnected, simple processing nodes placed in layers.

The output of a node is a function of the weighted sum of its inputs, optionally adding a bias term (Equation (2.14)). Such a function is called an *activation function*. The logistic or sigmoid function (Equation (2.15)) is an example that may be useful for approximating complex functions due to its nonlinearity.

$$a = g\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{2.14}$$

$$g(x) = \frac{1}{1+e^{-x}} \tag{2.15}$$

ANNs start their training process with random weights and biases and adjust them backwards to minimize a loss function, usually using a variant of the stochastic gradient descent algorithm.

Most ANNs are *feedforward* (Figure 2.6) since data flows in one direction only, from input to output. Alternatively, *recurrent* or *feedback* ANNs (Figure 2.7) include loops.

The architecture of an ANN comprises aspects such as the number of layers and nodes per layer. It is typically determined empirically by *hyperparameter tuning*.

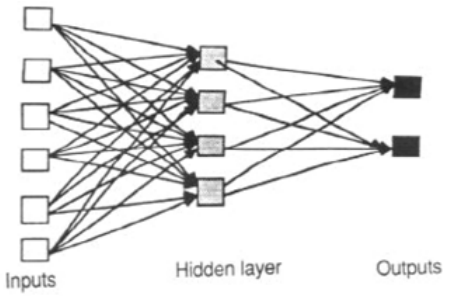


Figure 2.6: Feedforward network. [60]

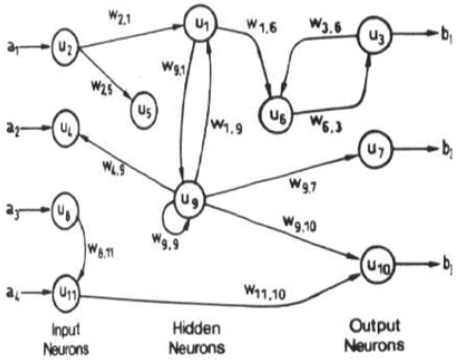


Figure 2.7: Recurrent network. [60]

2.2.4 Hyperparameter Tuning

Hyperparameter tuning (or hyperparameter optimisation) refers to adjusting the parameters used to control the learning process, e.g. the learning rate for stochastic gradient descent methods. These are different from the model parameters, such as the weights of a NN, which are not provided beforehand.

Hyperparameter tuning is typically performed to improve the learning speeds or the model's fitness potential.

The simplest, most common way of doing hyperparameter tuning is through a more or less exhaustive search of the values space, which may be provided *ad hoc* or randomly walked through. Although slow, this process is embarrassingly parallel.

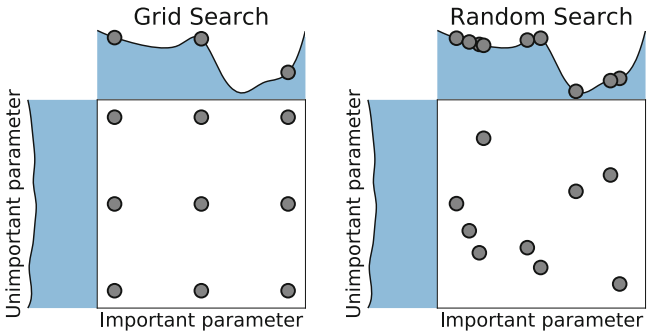


Figure 2.8: Grid Search versus Random Search in Hyperparameter Optimisation [15]. Suppose some hyperparameters are more important than others. In that case, a random search may be able to find higher-quality solutions.[6]

Walking through the hyperparameter values space more efficiently is not trivial because there is usually no "access to a gradient of the loss function with respect to the hyperparameter" [15], as we do not know the best hyperparameters beforehand. Bayesian optimisation and evolutionary approaches (Figure 2.9) are alternatives that improve the search process by reasoning about the utility of exploration versus exploitation.

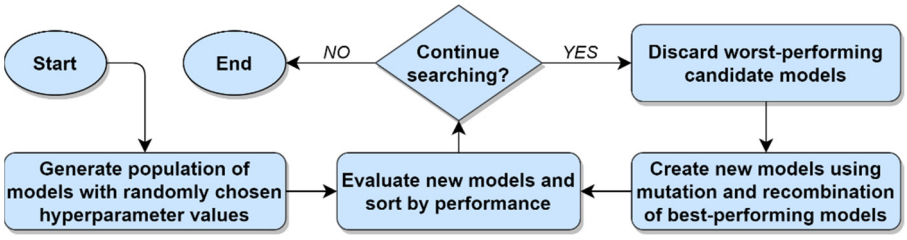


Figure 2.9: The evolutionary approach to hyperparameter tuning [54], using a genetic algorithm.

[6]It is not straightforward to identify hyperparameters that are more important than others. While a random walk may provide relevant insight through the monitoring of the model fitness according to the provided hyperparameters, one may also start from a package's defaults and resort to studies on tunability ranges and heuristics [46].

2.3 Summary

Real-time systems are everywhere in society and require that strict timing requirements, alongside logical correctness, are met. For that, scheduling policies orchestrate the order by which competing computational tasks are executed. In a multi-core system, the algorithms for achieving this job are more complex due to the enlarged search space.

Real-time scheduling policies are typically in place in real-time operating systems, although they can also be found in patched mainstream systems such as Linux.

The search problem for an optimal scheduling policy can be tackled using machine learning. Although supervised learning methods are famous for making predictions, reinforcement learning is the most viable method for controlling an agent in an environment and, thus, is fit for online control of scheduler policies.

The choice of algorithm hyperparameters may influence the potential quality of the resulting ML model. As such, the ML operator should strive to optimise them properly.

Chapter 3

State of the Art

After reviewing the most important concepts used throughout our work, this chapter presents itself as a study on the state of the art of scheduling algorithms and reinforcement learning. Section 3.1 analyses modern real-time task scheduling algorithms, while Section 3.2 studies the techniques and environments widely used in deep reinforcement learning, which will serve as the foundation of our optimization efforts. Section 3.3 presents related work. Finally, Section 3.4 contains the relevant takeaways of this chapter.

3.1 Modern Scheduling Algorithms

Modern real-time embedded systems are complex and usually comprise functions of different criticality. However, the scheduling algorithms analysed in Section 2.1.1 assume that no task execution time can surpass their worst-case execution time. As tasks' worst-case execution time shall be a provable upper bound, this leads to an inefficient use of resources on the average case.

Recent research considers the specificities of mixed-critical systems and adjusts the traditional scheduling algorithms to suit their requirements better.

3.1.1 Mixed-critical Systems

Mixed-critical (MC) systems are computational systems with tasks with heterogeneous criticality. Consider a commercial aeroplane. A deadline miss by a highly critical function, such as the auto-pilot subsystem, may be catastrophic, but a slight delay in the passenger's entertainment subsystem will hardly cause significant annoyance. Hence, the worst-case execution time analysis of the non-critical functions does not need to be exact, and complex control flow graph-based methods [13] may be replaced by more straightforward empirical measurements with a lower degree of confidence.

Vestal's model, from 2007 [59], extends the traditional task model by adding the level L of assurance that the actual execution time of a task will not exceed its C, where level A is the highest degree of assurance.

3.1.1.1 Optimal priority assignment

Vestal [59] demonstrates that the deadline monotonic priority assignment is not optimal for mixed-critical systems, as illustrated in Table 3.1, and presents two algorithms, possibly used in conjunction, for improving the allocated utilization time.

	$T = D$	L	C_B	C_A
Task 1	2	B	1	2
Task 2	4	A	1	1

Table 3.1: Characteristics of two tasks. Under DM, task 1 is assigned a higher priority, and the schedule is not feasible. However, the schedule is feasible if we consider task 1's worst-case execution time for level B and set task 2 as the highest priority. [59]

The first algorithm is based on period transformation. It executes run-time slicing for every j task, making $T'_j = T_j/n$ and $C'j = C_j/n$ for an integer n that is below the period of every lower-criticality tasks than j. This is shown to increase the allocated utilization up to 11% compared to DM.

The second algorithm is a modification of Audsley's priority assignment algorithm [3]. It assigns priorities to each task, one by one, starting with low priorities and increasing the priority after each assignment. At each step, the chosen task is any that will be schedulable if assigned the next higher priority. Combined with the previous algorithm, this is showcased to increase utilization up to 59% versus DM.

3.1.1.2 Schedulability concerns

Mixed-criticality offers new static schedulability analysis challenges compared to the simple response-time analysis presented in Equation (2.3), depending on the priority assignment scheme used, which might be one of [7]:

1. *Partitioned Criticality*: all jobs of criticality L1 < L2 are assigned lower priority than all jobs with criticality L2. Thus, all lower-priority jobs are, by definition, interrupted if a more critical one arrives.

2. *Static Mixed Criticality (SMC)*: allows priorities of different jobs to be interleaved, improving schedulability. A job's execution time is monitored in runtime and killed if it surpasses its representative WCET.

3. *Adaptive Mixed Criticality (AMC)*: Takes SMC one step further, by dispensing with the execution of all jobs of criticality L if any job of equal or lower criticality than L surpasses

its WCET. That is called a *mode change*: in SMC, there are as many criticality modes as the number of unique criticality levels of tasks in the taskset.

AMC can "obtain enhanced performance over the static scheme" [7] due to better schedulability. However, with AMC, there is a need to check not only the response time of jobs in all criticality levels (when their criticality is equal or higher than the criticality of the mode) but also the schedulability of the criticality change itself. For a two-criticality level scheme, we have:

$$R_i^* = C_i^H + \sum_{\tau_j \in hp(i) \cap \text{LO}} \left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j^L + \sum_{\tau_j \in hp(i) \cap \text{HI}} \left\lceil \frac{R_i^*}{T_j} \right\rceil C_j^H \tag{3.1}$$

Equation (3.1), used by *AMC-rtb*, states that the maximum response time of a job i with high criticality depends on its $WCET$ in that high criticality mode, as expected, but also on the possible interference of higher-priority jobs in high criticality mode and the possible interference of higher-priority jobs in low criticality mode, assuming that a runtime monitor kills them immediately once they surpass their $WCET^L$.

It is worth noting that Equation (3.1) is a recurrence, as its result depends on the result of its previous iteration. To test for the feasibility of an AMC schedule using this relation, one might:

1. Use a reasonable first guess for R_i^*, such as the $WCET^L$ of the task.
2. Iterate on R_i^* until either $R_i^*(n) \leq R_i^*(n)$, where n is the number of the iteration, or $R_i^*(n) > D_i$.
3. The schedule is guaranteed to be feasible if finally $R_i^* < D_i$ for all jobs i belonging to the task set; in other words, if the recurrence converged.

The above equation does not consider that all jobs in low criticality mode are killed once one of them exceeds the expected time budget - note that higher priority, low criticality jobs still "virtually" run according to the second parcel - and thus it is possible to bound the interference of those jobs using the instant s when the criticality change occurs, as is the case with *AMC-max*:

$$R_i^* = WC_i^H + \sum_{j \in \text{hpL}(i)} \left(\left\lfloor \frac{s}{T_j} \right\rfloor + 1 \right) C_j^L + \sum_{\tau_j \in hp(i) \cap \text{HI}} \left\lceil \frac{R_i^*}{T_j} \right\rceil C_j^H \tag{3.2}$$

Computing Equation (3.2) is more expensive than Equation (3.1) because it may require the solution of several recurrences, whereas the latter requires the solution of only one recurrence per higher-criticality task.

3.1.2 Efficient Mixed-critical Scheduling

Recent research focuses on improving scheduling under Vestal's model by incorporating details about the system's architecture, such as the use of caches or contention between cores when accessing main memory, or by adjusting the budget of task servers [1].

[1] A task server refers to a periodically recurring time window, in which some sporadic tasks are scheduled.

3.1.2.1 Dynamic redistribution of shared resources

Awan et al. [4] integrate the effects of the last-level cache, shared among all cores in a multi-core system with a single shared main memory, under a dual-criticality Vestal model. They realize that the worst execution time of a task depends on the number of its hottest pages allocated in the last-level cache (Figure 3.1) and seek to minimize the system utilization in L-mode via integer linear programming (Equation (3.3)).

$$Minimize \sum_{\forall \tau_i \in \tau} \sum_{j=0}^{\sigma^T} UL_{i,j} \times U_i^L(j) \tag{3.3}$$

where $UL_{i,j}$ is 1 if j pages are allocated to task τ_i in L-mode and 0 otherwise; τ is set of tasks; σ^T is the total number of cache pages; $U_i^L(j)$ is the utilization of task τ_i in L-mode when it is allocated j cache pages.

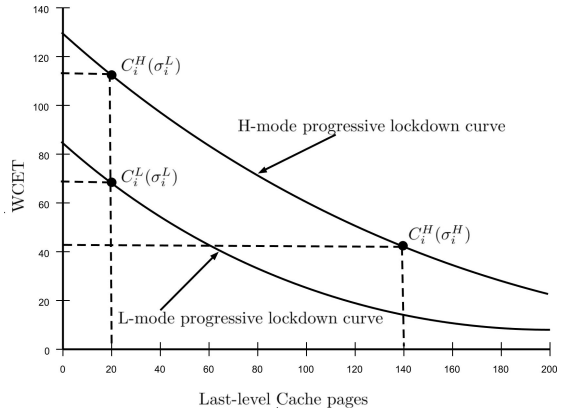


Figure 3.1: The relation of the number of pages locked in the last-level cache to the worst-case execution time of a task [4], determining the scaling factor for C from L-mode to H-mode.

The system's H-mode only guarantees that no H-task misses its deadline and dispenses with executing L-tasks. Hence, the pages reclaimed from idled L-tasks at the switch from L-mode to H-mode depend only on the H-tasks (Equation (3.4)).

$$Minimize \sum_{\forall \tau_i \in \tau} \sum_{j=0}^{\sigma^T} UH_{i,j} \times U_i^H(j) \tag{3.4}$$

where $UH_{i,j}$ is 1 if j pages are allocated to task τ_i in H-mode and 0 otherwise; $U_i^H(j)$ is the utilization of task τ_i in H-mode when it is allocated j cache pages.

The dynamic redistribution of cache pages at mode-switch outperformed approaches that do not redistribute cache pages with varying parameters like cache size, number of cores or fraction of H-tasks.

3.1.2.2 Mode-dependent server execution budgets

Task servers traditionally provide scheduling isolation via time partitioning: served tasks cannot use processor time when the server is not activated.

The fact that a given server can have very different processing needs in different modes requires careful server ordering and time budget allocation for better efficiency. Awan et al. [5] determine dynamic server budgets using two approaches.

The first, most straightforward approach assigns L-mode budgets (X_i^L) to all servers, proportionally to the minimum feasible L-mode budget X_i^{min} for each server, identified via a binary search algorithm. All server offsets and H-mode budgets are derived from these and depend on the selected server order.

The second method is based on *simulated annealing*. As each iteration, it selects two random servers, increases the budget for the first and decreases the budget for the second. If the overall weighted schedulability of the system improves, that is accepted as the new solution; otherwise, it is accepted with a probability directly related to a *temperature* factor that decreases over time.

The server ordering is given by non-decreasing $X_i^H - X_i^L$, which has been shown to perform slightly better than non-increasing X_i^H / X_i^L.

Supporting varying server budgets in different modes is worthwhile, as it improved schedulability ratios by up to 52.8% vs. static budgets.

3.2 Deep Reinforcement Learning Techniques

Since the advent of early RL algorithms, there has been interest in applying RL techniques to highly complex and dynamic environments, previously impossible due to the space restrictions of traditional methods. Nowadays, this is attained by the use of Deep Learning.

3.2.1 Deep Learning

The start of the 21st century was marked by the transition from shallow neural networks to complex, multi-layered neural networks, commonly referred to as Deep Neural Networks (DNN). In his breakthrough paper from 2006, Geoffrey Hinton, a British-Canadian computer scientist and cognitive psychologist, introduced the concept of Deep Learning (DL) [63].

Typical machine learning methods usually require careful hand-designed feature extraction to solve the *selectivity-invariance* dilemma: one that processes data so that important features are distinguishable and irrelevant details are removed. On the contrary, DL methods can transform raw data into high-level representations via the composition of simple but non-linear modules that can learn very complex functions. The hidden layers of a DNN typically detect motifs and features in a structured way, but they do so as part of the learning process and not as a result of human assembly [33].

DL is versatile because it can be applied to various areas of machine learning, from supervised to reinforcement learning.

In particular, convolutional neural networks - a type of feedforward neural networks - have been proven helpful for computer vision, pattern recognition and natural language processing [2]. Designed to work with multiple arrays as input, they are armed with pooling layers that merge semantically similar features into one, facilitating the detection of highly correlated, distinctive local motifs [33].

3.2.2 Deep Reinforcement Learning

With the successes of Deep Learning in recent years, its conjunction with Reinforcement Learning was inevitable. In 2017, Google Deepmind's AlphaZero beat Stockfish, the world's strongest chess engine at the time, after just seven hours of training through self-play.

3.2.2.1 AlphaZero

AlphaZero is a generalization of AlphaGo Zero, Google's previous RL-based playing agent. Provided with the model of the environment (in the case of chess, the rules of chess), AlphaZero trains a deep neural network $f_\theta(s) = (p, v)$ with parameters θ, that takes the board position s and outputs a vector of tuples of move probabilities p and the expected outcome v of the game.

At each state, the deep neural network suggestions are used, if exploited, as the next move of a Monte Carlo Tree Search (MCTS) routine that yields, at a leaf node, the result of the game, which is used to train the network in a backward fashion [53]. Figure 3.2 illustrates how the neural network outputs change as simulation rewards are obtained.

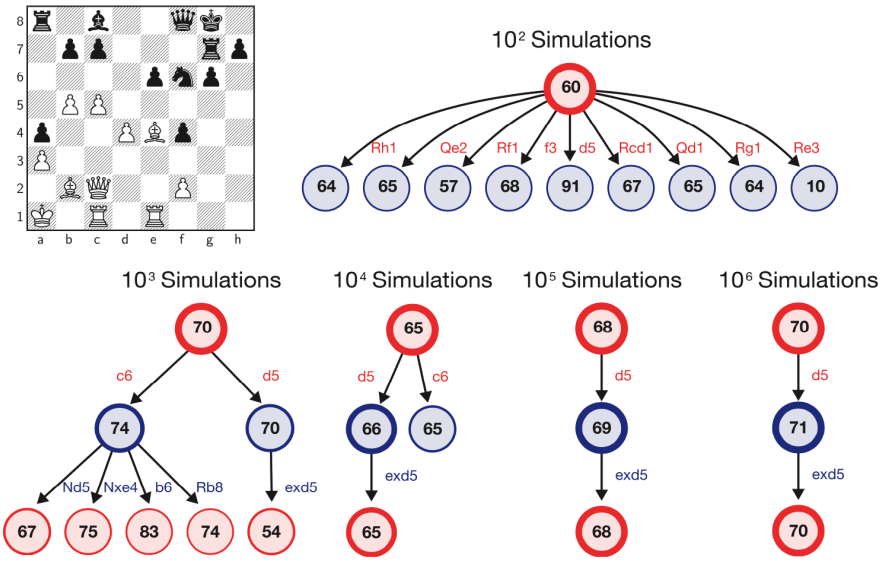


Figure 3.2: The most visited nodes in an AlphaZero MCTS search, as the neural network updates during the computer's time to move. Adapted from [53].

3.2.2.2 Deep Q-Network

Without a known model of the environment, a Deep Q-Network (DQN) works by approximating the Q-value function through a deep neural network [2]. This approach scales better than Q-learning (described in Section 2.2.2.2) in applications with large state spaces, such as a video game, in which states may be represented as a sequence of frames, as is the case in the Atari games first used to showcase the algorithm in 2013 [41]. In that approach, frames were preprocessed to reduce the input dimensionality by converting the RGB representations to greyscale and reducing the resolution of the images.

An essential challenge of using consecutive frames as the state input representation is the possibility of a high correlation between the frames hindering the correct understanding of the environment. As such, the original DQN paper suggests the use of an *experience replay memory*, a dataset of agent experiences $e_t = (s_t, a_t, r_t, s_{t+1})$, to sample experiences from during training, making it an off-policy method [3].

The original DQN [41] updates the network parameters only after a discrete number of steps to achieve stability. In spite of this, it uses the same network for selecting and evaluating actions, which could lead to over-optimistic value estimates. Recent algorithm modifications such as Double Deep Q-Learning decouple these two tasks in two different neural networks [21]. Updates are only applied to one of them randomly at each step.

3.2.2.3 Policy gradient methods

Instead of estimating the value of a state-action pair and using it to estimate the policy, policy gradient methods directly compute a parameterizable policy $\pi[a_t|s_t,\theta]$, where θ is a set of policy parameters [45]. They aim to maximize the expected return over many trajectories τ (3.5) and use a gradient ascent update, with learning rate α, to adjust the parameters θ accordingly (Equation (3.6)).

$$\theta = \underset{\theta}{\operatorname{argmax}}[\mathbb{E}_\tau[r[\tau]]] = \underset{\theta}{\operatorname{argmax}}\left[\int Pr(\tau|\theta)\cdot r(\tau)\,d\tau\right] \tag{3.5}$$

$$\theta \leftarrow \theta + \alpha\cdot\frac{\partial}{\partial\theta}\int Pr(\tau|\theta)\cdot r(\tau)\,d\tau \tag{3.6}$$

Including the probability of a trajectory Pr in this formula ensures that already likely trajectories with positive rewards do not influence the policy as much as newer, unseen ones, as seen in Figure 3.3.

Several policy gradient methods have been proposed since the early REINFORCE algorithm, described in Section 2.2.2.2. They are primarily *actor-critic* methods, combining an actor with

[2] This network may have convolutional layers, as in the example used in DQN's original paper, for extracting relevant features of an image. That is not the case in our problem.

[3] Off-policy and on-policy are terms used to describe how the interaction policy relates to the learned policy. Off-policy means that the agent can learn from data generated by a different policy (often more exploratory) than the one it currently follows. In DQN, this is enabled by the experience replay memory and how updates are applied to the target network, a "frozen" copy of the Q-network, while experiences are gathered from the most current network.

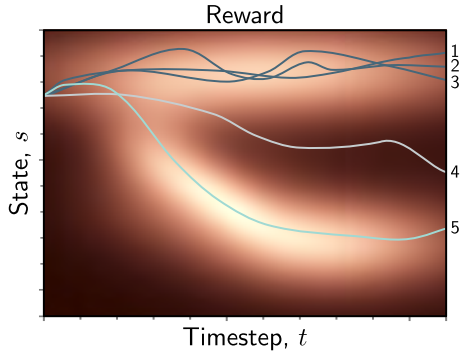


Figure 3.3: A set of trajectories [45]. Brighter colours indicate higher rewards. Trajectories 1, 2 and 3 generate high rewards, such as 5, but occur with higher likelihood, as denoted by the overlap of states. As such, traversing 5 results in a steeper change to the policy, following Equation (3.6).

policy parameters with a critic capable of learning a value function to help it in the direction of performance improvement [31].

Deep Deterministic Policy Gradient (DDPG) was proposed in 2016 [34] as a means to adapt DQN to a continuous action space, characteristic of environments with real-valued actions. DQN can not be used in such environments because it strives to find the action that maximizes the action-value function, which requires an expensive iterative optimization process at every step in a continuous action space. Instead of discretizing the action space and losing precision, DDPG tackles the issue by following an actor-critic approach (Figure 3.4). This includes, as in DQN, target networks; however, the update rules for both the actor-network Q and the critic-network μ are softened by a scaling factor τ (Equation (3.7) and Equation (3.8)).

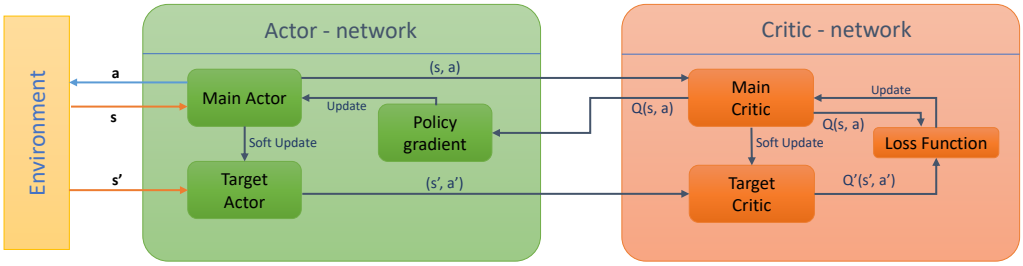


Figure 3.4: The DDPG algorithm structure, as presented in [55]. The critic-network estimates the Q-value of a state-action pair sampled from the actor-network. The actor-network then uses it to update the policy.

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \tag{3.7}$$

$$\theta^{\mu'} \leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \tag{3.8}$$

Another popular DRL algorithm is the Proximal Policy Optimization (PPO) algorithm, proposed in 2017 [49]. It formulates a novel objective function (Equation (3.9)) that enables multiple

epochs of minibatch updates, meaning that the algorithm performs multiple updates using the same batch of collected experiences. Over several iterations, a set of actors run policy π_{old} in the environment and compute the advantage estimate A_n - the difference between the observed return and the expected value of the state -, progressively optimizing the objective function L. Clipping the policy change factor ensures that the algorithm is less sensitive to hyperparameters and has better sample complexity, i.e., trains faster and more stably.

$$L(\theta) = \mathbb{E}_t \left[\min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A_t, \text{clip} \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon \right) A_t \right) \right] \tag{3.9}$$

The performance of each of these algorithms is highly dependent on the characteristics of the environment. An empirical study from 2023 [52] found that DDPG trains in less time than PPO and achieves higher maximum rewards in an autonomous driving environment. Due to the latter's absence of an experience replay buffer, the trained driving agent suffered from "catastrophic forgetting" and crashed frequently after successful episodes.

3.3 Related Work

There are multiple tentatives in applying Deep Reinforcement Learning to scheduling problems. They are generally successful in highly dynamic environments such as instances of the job-shop scheduling problem in which machine breakdowns or material problems may impact the job's time [36].

However, it must be noted that the time available for performing these scheduling mutations in an environment such as a textile industry is much higher than in a real-time computer system, where each time unit is precious for the feasibility of the schedule. As such, we see this success with a grain of salt and avoid making strong analogies to our work.

The relative absence of relevant related work is arguably due to the novelty and certification challenges of using ML in runtime with critical systems such as hard real-time systems. As such, RL must be carefully adapted to our domain.

3.4 Conclusions

Modern and complex real-time systems are mixed-critical in that they comprise tasks of multiple criticality. Under the umbrella of the Vestal model, there is significant research on scheduling optimization, namely via architecture considerations.

The most used DRL algorithms nowadays are just a few years old. Actor-critic approaches such as DDPG or PPO have improved on DQN and are widely used in the industry.

Much work applies DRL to scheduling problems, but usually not in the computational field, in which substantial timing requirements require further consideration. The relative absence of related work might enhance the potential of this work.

Chapter 4

Methodology

This chapter formulates our hypothesis (Section 4.1) and the problem we will investigate to prove it (Section 4.2). A proposed solution is presented in Section 4.3. Our methodological approach is described in Section 4.4. Lastly, we explain how evaluation should be performed (Section 4.5).

4.1 Hypothesis

With this dissertation, we hope to improve the efficiency of real-time task scheduling on a single-core platform. Realizing the complexity of modern real-time systems and the suitability of Vestal's mixed-criticality model to improve their efficiency, our work will focus on a mixed-critical platform.

We seek to find whether a DRL agent will be able to achieve this goal by realizing execution patterns at runtime and acting upon them. In summary, we hope to prove empirically that:

> *Using Deep Reinforcement Learning in a mixed-critical real-time system to orchestrate the scheduling process can improve the overall schedulability of the system.*

The *overall schedulability of the system* is mainly given by the processor utilisation, provided that real-time guarantees are met: the higher the utilisation, the better, since the system may perform more activities with the same computational resources, reducing the number of cores and decreasing cost and power consumption.

In a dual-mode mixed-criticality setting, the overall schedulability is also related to the number of task cancellations and/or mode changes: the fewer task cancellations there are, the longer the system operates seamlessly with all functions intact.

4.2 Problem Statement

A significant issue when designing mixed-critical, hard real-time systems is determining the L-mode time budget (or the $WCET^L$ for a task) in a way that is both reasonably safe and uses platform resources in the best way possible. In other words:

> *It is not trivial to assign time budgets to tasks in a mixed-critical, real-time system due to the trade-off between perceived safety and potential resource utilisation.*

As such, we will reproduce a scenario where this problem exists, to be able to test our hypothesis.

4.2.1 Platform

We assume a single-core platform. We expect a hardware timer to interrupt the processor with a high frequency so that task scheduling code may be run and a context switch can be performed if needed.

4.2.2 Task Model

We use Vestal's model with two criticality levels, high (H) and low (L). Each task can be critical (H-task) if it is essential for the system's operation and cannot be dispensed with during runtime or non-critical (L-task) if it can be dispensed with at runtime during relatively short periods of system overload.

A H-task is provided with two estimates of its worst-case execution time: $WCET^H$, which is determined with a level of assurance according to its criticality, and $WCET^L$, which may be determined with a lower level of assurance (for instance, the worst-case execution time observed over a thousand runs, possibly slightly scaled up).

The system may run in two modes - L-mode and H-mode - and uses a slight variation of AMC. The system boots in L-mode. In L-mode, all tasks are assigned an execution budget equal to their $WCET^L$. If it is observed that a task takes more time to execute than its $WCET^L$, the action performed depends on the nature of that task: an L-task is simply killed so that other instances of it and other L-tasks are scheduled as usual. This deviates from the original definition of AMC, in which all L-tasks are killed and a mode change is immediately triggered, and is done to minimize mode changes *a priori*.

In the case of an H-task, a mode change is triggered since H-tasks cannot be killed. We must guarantee that, regardless of when the mode change occurs, there is no way that any H-task may miss its deadline. This is done by proper *mode-change time-analysis* as explored in Section 3.1.1.2.

In H-mode, only H-tasks are scheduled, using their $WCET^H$. The system switches back to L-mode when idle, i.e. no task is being executed.

4.2.3 Scheduling Algorithm

The system is scheduled internally using fixed-priority scheduling. Tasks are prioritized according to their period: tasks yielding more frequent jobs will have a higher frequency, as in RM.

4.3 Proposed Solution

We propose introducing Deep Reinforcement Learning in a real-time system using our mixed-criticality model as a sporadic L-task. This guarantees that there is a limit to the time the agent can require and that time can be accounted for during scheduling analysis for certification purposes.

The environment in which the agent executes is the processor with its scheduled tasks. It *observes* the execution time of tasks and *acts* by changing their budgets - more specifically, their execution budgets.

4.4 Methodological Approach

We propose to test the hypothesis as mentioned earlier by setting up a simulated, proof of concept real-time environment in a general-purpose programming language, using a mixed-criticality model. This is done so we can focus on the effects of the DRL agent without worrying about OS details and/or lack of support for the proposed environment. For instance, when writing, popular RTOS such as Zephyr or FreeRTOS didn't yet support Vestal's model and would require a major refactor to accommodate that.

We will generate a set of critical and non-critical tasks to be scheduled and provide auto-generated $WCET$ estimates and deadlines. At runtime, tasks will take a simulated time sampled from a probability distribution with some parameters derived from its estimated $WCET$.

We shall test different DRL models that will be able to mutate the time L-mode time budgets of tasks in the system. As a proof-of-concept, we assume that the DRL agent is *omniscient* and knows everything happening in the system, namely where and for how long tasks have been executing (which would not be the case if working with a real OS kernel). We will perform hyperparameter tuning and experiment with different actions and rewards. Following RL principles, negative rewards should be generated each time the system switches from L-mode to H-mode or an L-task is killed, since these are the events that we are trying to prevent from happening.

4.5 Evaluation Approach

Our goal is to test whether there is an increase in the quality of service and/or cost reductions. The former requires evaluating the number of mode changes, while the latter is related to processor utilisation: the more the better, meaning that less tasks are dispensed with and idle time decreases.

We expect our testing scope to be large. We need to test different sets of actions and different models with an ample set of hyperparameters.

Chapter 5

Runtime Orchestration Environment

This chapter explains the challenges and considerations (Section 5.1) that led to our system design decisions (Section 5.2). It then uncovers relevant implementation details in the simulator (Section 5.3) and the agent (Section 5.4). It ends with a summary of the implementation work developed (Section 5.5).

5.1 Challenges and Initial Considerations

Proving the hypothesis proposed in Chapter 4 brings up a handful of design challenges:

1. No open-source OS or RTOS supports the Vestal mixed-criticality model, much less our AMC variant.

2. Certifying a solution for real-time systems typically requires providing hard-bounded timing and memory constraints, which is much harder using comprehensive RL frameworks.

3. It is hard to guess industry expectations of a mixed-criticality system due to intellectual property concerns.

The issue regarding the absence of production OSs following our scheduling model requires either implementing an OS, modifying an existing one or implementing a simulator of a system executing a bunch of tasks. The first two are not the main focus of our work and are extremely time-consuming. As such, we implemented a simulated real-time system that we will refer to as the *simulator*, detailed in Section 5.3.

The potential for certification is vital to target production systems. We find that using a bare-bones ML library such as libtorch [24] [1], providing simple ANN operations and optimisers, and implementing the DRL logic on top is more transparent and easier to reason about compared to using a black-box library such as *TensorFlow Agents* [18].

The final point recommends using a reliable source to create authentic mock tasks. We tackle this issue in Chapter 6.

[1] *libtorch* is the C++ distribution of PyTorch, a famous Python-first ML library.

5.2 System Design and Architecture

Figure 5.1 presents a high-level view of the system's design. It includes a simulator following the scheduling algorithm detailed in Chapter 4, with a set of simulated tasks that are activated according to their priorities and consume a pseudo-random amount of time for each activation. One of these tasks is our DRL agent, a sporadic L-task of the system.

The agent contains data structures necessary for performing the DQN algorithm, which is adjusted to the problem, and an interface to communicate with the simulator. As explained in Section 5.3, the simulator informs the agent of relevant events, which shall be translated to environment state and transitions when the agent activates and inference runs.

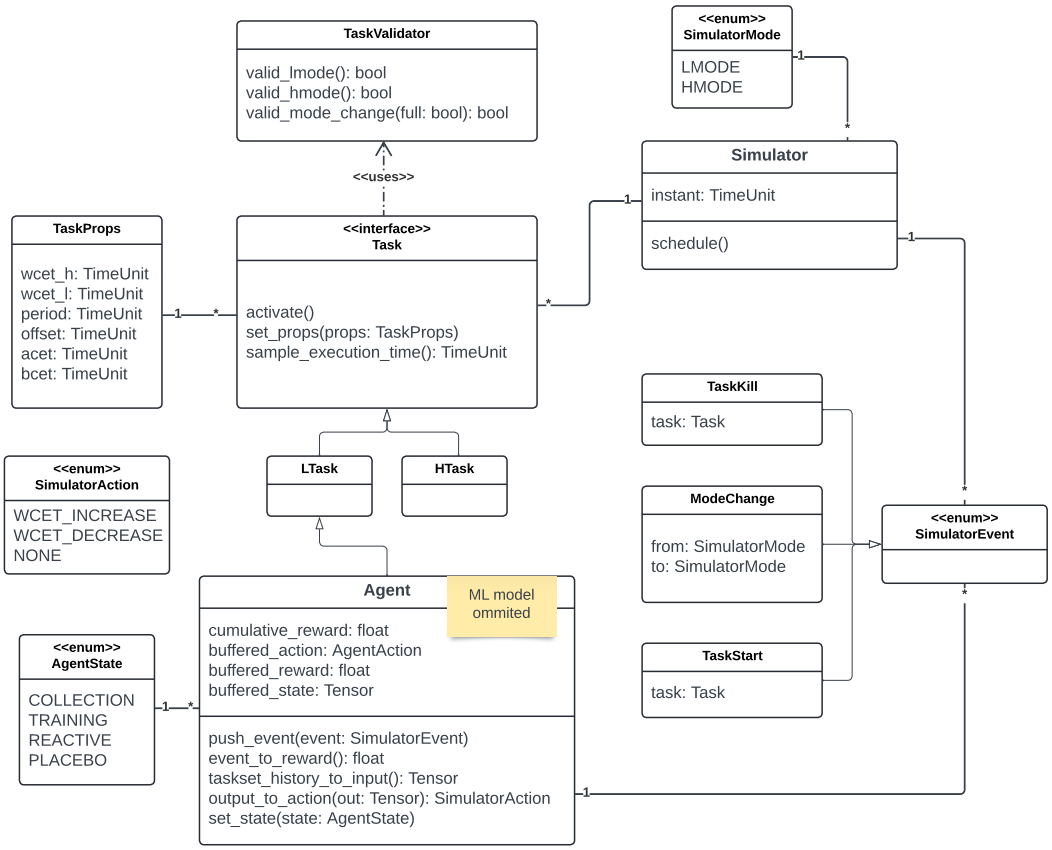


Figure 5.1: High-level class diagram of the system.

The full system is implemented in the Rust programming language [48]. Rust is an emerging systems language with zero-cost abstractions in the likes of C++ and designed with memory safety in mind. It's also comparable in terms of energy efficiency [44] and has a rich package ecosystem and developer tooling.

5.3 Simulator Environment

The simulator is a module that simulates the execution of a real-time system, generating the arrival of jobs of tasks with a set of parameters according to a fixed-priority scheduling scheme during a desired time interval.

The simulator starts by generating the first job arrival events and pushing them to a queue. Once the time to run these jobs comes, the event is popped from the queue, and a new arrival event for the next job arrival of the same task is created. Every time that job is *able* to run (in an instant that may not coincide with its arrival, in case a higher-priority job is already running), its execution time is discounted until it is finished. Whether it is finished or not is a matter of the given job execution time, analysed in Section 6.1.2.

In our AMC variation, discussed in Chapter 4, L-tasks cannot overrun their given time budget and are immediately cancelled if that happens. Conversely, if H-tasks overrun their L-mode time budget, a mode change to H-mode is triggered. These system events, as the other ones mentioned above, are signalled to the intelligent agent as they happen, but will only be processed by it once it activates, which happens only when the system is idle. This decision is explained in Section 5.3.2.2.

To speed up execution time, we only process instants in which events (such as start or end) are scheduled. In a real kernel scheduler, the main loop may be called in every HW timer interrupt, or only after the expiration of a timer that may be set for the next start event, to reproduce this optimisation.

Due to similar performance-related reasons, the code does not check for the occurrence of two jobs of the same task in the running queue simultaneously, which would render the schedule unfeasible. Thus, any task set fed to the simulator must be mathematically proven feasible.

Algorithm 1 presents a simplified pseudo-code of the simulator.

5.3.1 Event Recording

The agent is notified of relevant events throughout the simulation, such as the start of a job, a mode change or a task kill, via the *push_event* interface (see lines 14, 17, 22 and 29 of Algorithm 1). These will be assigned negative or positive rewards that shall be processed when the agent is allowed to run and serve as a debugging tool to assert the correctness of the simulation via unit tests.

The agent is informed immediately of any events and pushes them to an internal queue. Our analysis considers this "signal" to take negligible time and thus be considered part of "context-switch" or kernel code, which we will omit from consideration.

Algorithm 1: Simulator

```
1  current_mode ← LMode;
2  current_running_jobs ← [];
3  task_start_events ← collect_start_events(self.tasks);
4  time ← 0;
5  while time < duration do
6      new_jobs ← determine_new_jobs(task_start_events, current_mode, time);
7      current_running_jobs.append(new_jobs);
8      current_running_jobs.sort_by_priority();
9      if current_running_jobs.is_not_empty() then
10         job ← current_running_jobs.first();
11         if job.is_starting() then
12             next_start_event ← find_start_event(task_start_events, job);
13             update_start_event(next_start_event, job);
14             push_event(agent, job, time, START, job.id);
15         if job.has_finished() then
16             remove_job(current_running_jobs, job);
17             push_event(self.agent, job, time, END, job.id);
18         else if current_mode == LMode ∧ job.running_for ≥ job.task.wcet_l then
19             if job.task == HTask then
20                 current_mode ← HMode;
21                 retain_htasks(current_running_jobs);
22                 push_event(self.agent, job, time, MODE_CHANGE, HMODE);
23             else
24                 kill_task(current_running_jobs, job, time);
25                 push_event(agent, job, time, TASK_KILL, job.id);
26     else
27         if current_mode ≠ LMode then
28             current_mode ← LMode;
29             push_event(self.agent, job, time, MODE_CHANGE, LMODE);
30         run_agent(agent, time);
31     time ← next_start_event(task_start_events, time);
```

5.3.2 Agent Restrictions

5.3.2.1 Actions on the Environment

As discussed in Section 2.1.1.2, to guarantee that an AMC schedule is feasible, both the standard response time analysis for L- and H-mode and the analysis of the response time of H-tasks upon a mode change must be valid. As such, any attempt by the agent to change the environment that violates this analysis is *discarded*.

We discard the agent's action if Equation (5.1) would be violated as a consequence. Recall that the convergence of this recurrence is a sufficient condition for asserting the feasibility of AMC schedules, as presented in Equation (3.1).

$$C_i^H + \sum_{\tau_j \in hp(i) \cap \text{LO}} \left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j^L + \sum_{\tau_j \in hp(i) \cap \text{HI}} \left\lceil \frac{R_i^*}{T_j} \right\rceil C_j^H \leq D_i \tag{5.1}$$

Solving this recurrence may take some time. As time is critical in hard real-time systems, we can use Equation (5.2) instead:

$$C_i^H + \sum_{\tau_j \in hp(i) \cap \text{LO}} \left\lceil \frac{R_i^{LO}}{T_j} \right\rceil C_j^L + \sum_{\tau_j \in hp(i) \cap \text{HI}} \left\lceil \frac{D_i}{T_j} \right\rceil C_j^H \leq D_i \tag{5.2}$$

The revised equation replaces R_i^* by D_i in the third parcel. Since $D_i \geq R_i^*$, the upper bound is greater, but the interference by higher priority H-tasks, i.e. the last term on the above inequality, may also increase. It is then unclear which equations would lead to better overall scheduling results, but it is safe to say that this version is faster and, thus, will be preferred.

In RL terms, this environment is non-deterministic to the agent: attempting to change a system property (in this case, the $WCET^L$ of a task) will not always have the desired effect.

5.3.2.2 Activation

A more significant issue is when to activate the agent. Running the agent arbitrarily (or, e.g. as a periodic task) is not possible since it may decrease the $WCET^L$ of any task τ_i that in its current busy period may have already executed for more than the new $WCET^L$ assigned by the agent, rendering the system unschedulable.

The agent is modelled as a sporadic task of the system, which may only be activated when the system is idle in any mode. Note that for the task to be considered sporadic in an RT system, there must be a minimum interval between activations, which is not specified here. However, since we do not run the scheduling loop for multiple consecutive time instants [2] (due to line 32 in Algorithm 1), that interval is at least equal to the minimum execution time of a task in the task set.

If the system is highly loaded, the DRL agent may be able to intervene only upon the switch from H- to L-mode for reasons similar to the ones that prevent us from changing the $WCET^L$ while there is any job running. This may ultimately affect the agent's responsiveness and our idea's success.

5.4 Agent Behaviour

Although with restrictions, the agent may attempt to change the properties of the tasks as it sees fit. As a proof of concept, it can increase or decrease the $WCET^L$ on any task.

The agent uses a deep reinforcement learning algorithm to train and perform actions. More specifically, it uses an implementation of DQN initially based on an open repository [58] and hand-tweaked, e.g. to support multiple hidden layers and complete parameter configuration.

[2] Of course, that would not be the case in a real HW timer interrupt scenario, requiring further consideration.

The choice of DQN is mostly based on ease of implementation. However, research is sparse on the space and time complexity of different DRL algorithms, and one can note that actor-critic methods are less space-efficient due to the multiple network architecture, and policy gradient methods arguably perform more time-consuming computations.

A fully-trained agent, ready to be deployed, performs only inference on its policy network. Figure 5.2 presents the architecture of a policy network with one hidden layer in a system with 3 tasks. The input nodes represent the state and are better detailed in Section 5.4.2.1. The output nodes represent the Q-values of each available action in the system. The set of available actions is analysed in Section 5.4.2.2.

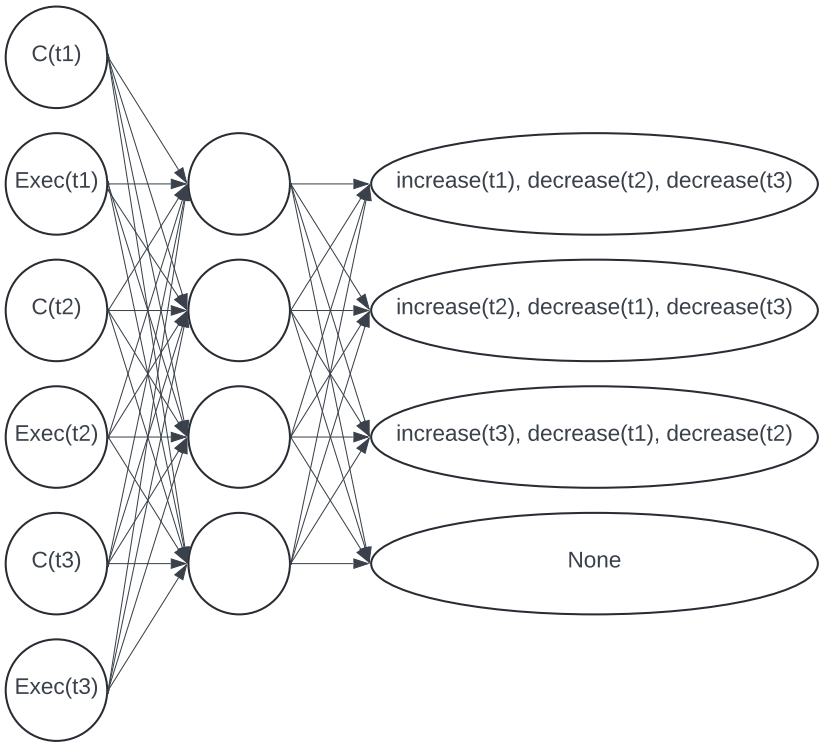


Figure 5.2: The policy network of an agent in a system with three tasks t1, t2 and t3.

5.4.1 DQN Implementation

The implementation of DQN that served as the base of our work uses *libtorch* bindings for the Rust language, provided by the *tch* package [38], which allows running *libtorch* functionality in the CPU or the GPU, out of the box, in Rust. The CPU is preferred since target real-time devices provide no dedicated processing unit besides the CPU [3].

[3] An idea that came to us as we were discussing this chapter is to use an external GPU to run the agent on more sophisticated platforms, e.g. Tesla cars. Since RT tasks likely do not use the GPU, the agent's interference might be minimal.

5.4.1.1 ANN Barebones

Implementing the barebones of a neural network is straightforward, having *libtorch* set up. It provides *tensors*, abstractions for multidimensional matrices containing elements of a single datatype. Given that performing training and inference is a matter of matrix multiplications and calculating gradients (derivatives) of a loss function with respect to the model parameters, we can abstract a neural network using tensors as primitives, as seen in Figure 5.3.

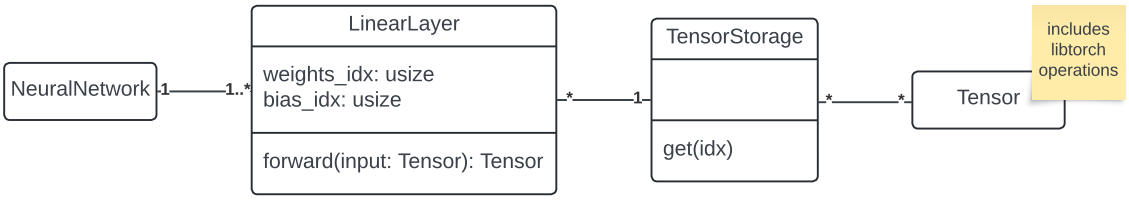


Figure 5.3: Abstracting over *libtorch* tensors. *Tensor* is provided by *libtorch*, while the others are interfaces provided by us.

5.4.1.2 DQN Infrastructure

A simplified pseudo-code of the agent task is presented in Algorithm 2.

The basic foundation of DQN is inferring the best action in a given state using a policy network (recall the theoretical explanation in Section 3.2.2.2). While constructing a state representation from task events is detailed in Section 5.4.2, it is pivotal to understand how inference and training work.

The agent executes differently depending on its stage:

1. **Data collection:** the agent fills the replay memory with transitions collected from training experience, to be sampled later for training.
2. **Training:** the agent refreshes the replay memory with newer transitions and trains a policy network by computing the loss against a target network.
3. **Reactive:** the agent is no longer training; it determines actions only via a forward pass to a frozen policy network. This is the "deployable" stage.
4. **Placebo:** the agent executes all the code of reactive mode but always chooses to do nothing. Useful for testing.

The data collection stage is necessary for preparing training, given the need for *replay memory*. As explored in Section 3.2.2.2, the replay memory is used to sample a batch of transitions that represent *motion* or *causality* in the environment, which is fed during training to both the policy and target networks to compute the loss (lines 17 to 23 of Algorithm 2).

The training stage is where the adjustment of the policy network weights occurs. To avoid "catastrophic forgetting" and stabilize learning, the target Q-values are retrieved from a pass to

Algorithm 2: Agent activation

```
1  state ← history_to_input(events_history, simulator);
2  action ← None;
3  if self.stage ≠ Placebo then
4      action ← epsilon_greedy(memory_policy, policy_network, epsilon, state, simulator);
5  action.apply(tasks);
6  if action ≠ None then
7      if ¬ feasible_schedule_online(tasks) then
8          action.reverse().apply(tasks);
9  if self.buffered_action ≠ None then
10     reward ← calculate_reward(events_history, simulator);
11     push_transition_replay_mem(buffered_state, buffered_action, reward, state);
12     if replay_memory.is_filled() then
13         stage ← Training;
14 buffered_action ← action;
15 buffered_state ← state;
16 if self.stage == Training then
17     sample_batch ← replay_memory.sample_batch(sample_batch_size);
18     qvalues ← policy_network.forward(memory_policy, sample_batch.state).gather();
19     target_values ← target_network.forward(memory_target, sample_batch.next_state);
20     expected_values ← sample_batch.reward + gamma × max_target_values;
21     loss ← mean_squared_error(qvalues, expected_values);
22     backward(loss);
23     memory_policy.apply_grads_adam(learning_rate);
24     if reward_history_len % update_freq == 0 then
25         target_network.update(memory_policy);
26         update_epsilon();
```

a frozen target network that gets updated once in a while from the current values of the policy network (lines 24-26). These are then compared to the Q-values retrieved from the policy network to adjust the latter's weights via a gradient-based learning algorithm (in this case, Adam [4].).

The remaining two stages are straightforward: reactive is a production-ready state transformer and action inference algorithm with no further adjustments; placebo infers actions but does not apply them to the environment.

[4] Adam is "an algorithm for first-order gradient-based optimization of stochastic objective functions" that is considered to be very memory efficient [30]

5.4.2 Transition Model

5.4.2.1 State representation

Line 1 of Algorithm 2 constructs a state from the history of events. This is not done to provide the agent with state evolution throughout time - that is already done by the processing of batching some experiences at a time - but to be able to feed relevant information, such as the last task execution time. Had this not been done, we would be left with a static representation of the taskset, including, e.g., only the task properties, that could be more safely processed at design time with some heuristic, which is not what we propose.

We represent a state of an environment with a task set $t = \{t_1, t_2, \ldots, t_n\}$ as a sequence of tuples $s = \{(WCET^L(t_1), LastExec(t_1)), \ldots, (WCET^L(t_n), LastExecTime(t_n))\}$, where $LastExec(t)$ is the difference in time between the last start and end events of task t, if found, and -1 otherwise.

This representation is a proof-of-concept framework and may be augmented with further information. For example, in a non-simulated setting, it may be interesting to include recent memory page usages to try to uncover deeper patterns similar to what is suggested in the literature (refer to Section 3.1.2.1).

5.4.2.2 Action set

The output nodes of the policy network are the sequential Q-values of each action available in the environment. Due to the characteristics of DQN, there is a need to discretize the set of available actions.

The actions available in an environment with a task set $t = \{t_1, t_2, \ldots, t_n\}$, where $n \geq 3$, is the set $\mathscr{A} = \{(\text{increase}(x), \text{decrease}(y), \text{decrease}(z) \mid x, y, z \in t, x \neq y \neq z\} \cup None$. In other words, for each activation, the agent may choose to:

1. Increase the L-mode time budget of task x by 10% while decreasing the L-mode time budget of tasks y and z by 5% each.

2. Do nothing.

The rationale for increasing one task's budget while decreasing the budget of another two stems from the assumption that some tasks' budgets will err by excess and others will err by defect. Conversely, this contributes to relative maintenance of the processor utilisation of the task set, which would not be the case if we were to only increase or decrease budgets at a time.

The *None* action is there to be preferred when the agent considers that the budgets for all tasks are already fair considering past behaviour.

For determining the action to apply in a given agent activation, we run inference via a forward pass to the policy network, starting from the environment state representation explained in Section 5.4.2.1, and select the output node with the higher Q-value, at index i. Since the set $\mathscr{A}$ is calculated *a priori* when the agent is bootstrapped (when the simulation begins), it suffices to index $\mathscr{A}$ at position i. If i equals the length of $\mathscr{A}$, we apply *None* instead.

5.4.2.3 Rewards and Transition Assembly

A RL algorithm such as DQN collects transitions between states, which include not only the input and output states, but also the action taken and the received reward.

A more typical RL problem would be able to assemble a transition in a simple way, since there is normally access to the next iteration of the environment, or the next state, immediately after the action is performed. Unfortunately, due to agent activation restrictions discussed at Section 5.3.2.2, the agent needs to wait for its next activation to peek into the next state of the system, which may take some time, since it is activated only when the system is idle.

As such, rewards are *delayed* until the next activation of the agent. To accommodate that, a previously chosen action and previous state is saved in memory (lines 14 and 15 of Algorithm 2) until the next activation, that is, until there is information about the evolution of the environment after the action is taken. At that time, a transition is assembled from the saved action and state, a calculated reward and the next state of the environment.

The rewards are retrieved from the set of new, unprocessed simulator events, notified in the meantime. The reward of a given action in a given state is the sum of the rewards of the unprocessed simulator events. An event is mapped to a reward as indicated:

$$\text{event\_to\_reward(event)} = \begin{cases} 0.1 & \text{if event} = \textit{Start} \\ -1.0 & \text{if event} = \textit{TaskKill} \\ -2.0 & \text{if event} = \textit{ModeChange(LMode to HMode)} \\ 0.0 & \text{otherwise} \end{cases}$$

The rationale for assigning the above rewards is arguably very intuitive. We assign a small positive reward to task start events since they mean that the processor is not idle and computational resources are being used. Conversely, task kills and mode changes are assigned high negative rewards since these are the events we are trying to prevent in the system with our approach.

5.5 Summary

There were major challenges to be addressed in the implementation of the system. With a well-defined scope, the designed system comprises a task scheduler simulator and a DRL agent interfacing with it.

The simulator applies our variant of AMC and activates the agent when it can, in a way that must be proven mathematically to be safe for certification purposes. Following the AMC directives, we determined it can do so only when no task is running, which may hinder some of the agent's effectiveness.

The agent runs the DQN algorithm modified to accommodate the delayed information characteristic of the agent's sporadic activation. Transitions are pushed in the activation after an action has been executed, and rewards are intuitive given the events we strive to prevent from happening.

Chapter 6

Experimental Evaluation

After reviewing the most important implementation details, this chapter presents an experimental system evaluation. Section 6.1 details how we base our benchmarks on industry research, while the following sections focus on the computational requirements of the developed agent (Section 6.2) and its impact on the quality of service (Section 6.3). Finally, Section 6.4 discusses the obtained results and their significance for the project's overall impact.

6.1 Tasks Dataset Generation

A significant issue that may arise when evaluating an innovative approach with potential industry utilization is the closed-source nature of business operations. Realistic benchmarks are often unavailable due to intellectual property concerns, significantly hindering a fair understanding and comparison of technological breakthroughs from academia or the industry.

Fortunately, corporate research done in Robert Bosch GmbH [32] unveiled a set of application characteristics of a specific real-world automotive software system. Returning to Chapter 1, the reader may recall that the automotive industry is one of our motivations for this research. As such, our test scenarios and all data used for their generation will be based on that benchmark.

6.1.1 Typical Automotive Taskset Features

Kramer et al. [32] mention that a typical automotive system is decomposed into components, e.g. AUTOSAR Software Components. The smallest decomposition of a component that is subject to scheduling is a *runnable*. Runnables are activated sporadically, in case of angle-asynchronous activities such as the rotation of the crankshaft or other external interrupts, or periodically, in the remaining cases. Since we are not provided with information regarding interrupt frequency and variations in engine rotation during a car journey, we discard sporadic runnables from our scenarios. We also do not care about the intra-/inter task communication details; we only acknowledge that they affect their expected execution times.

Runnables with the the same period are mapped to the same task. Therefore, we construct a task by adding the properties of 2 to 5 runnables (as per Table 6.1) with the same period. For example, a task C composed of 2 runnables A and B, given that $WCET^H(A) = 4$ and $WCET^H(B) = 5$, will have $WCET^H(C) = 9$. This is pessimistic, but honouring the absolute timing certainty that hard real-time systems require is necessary.

Number of Runnables	Share
2	30 %
3	40 %
4	20 %
5	10 %

Table 6.1: Runnables per task. [32]

The average-case execution time (ACET) of a runnable is given in Table 6.2. It is then multiplied by a factor f oscillating between f_{min} and f_{max}, as shown in Table 6.3, to obtain the best-case execution times (BCET) and worst-case execution times.

Period	Average Execution Times (μs)		
	Minimum	Average	Maximum
1 ms	0.34	5.00	30.11
2 ms	0.32	4.20	40.69
5 ms	0.36	11.04	83.38
10 ms	0.21	10.09	309.87
20 ms	0.25	8.74	291.42
50 ms	0.29	17.56	92.98
100 ms	0.21	10.53	420.43
200 ms	0.22	2.56	21.95
1000 ms	0.37	0.43	0.46

Table 6.2: Runnable ACETs. [32]

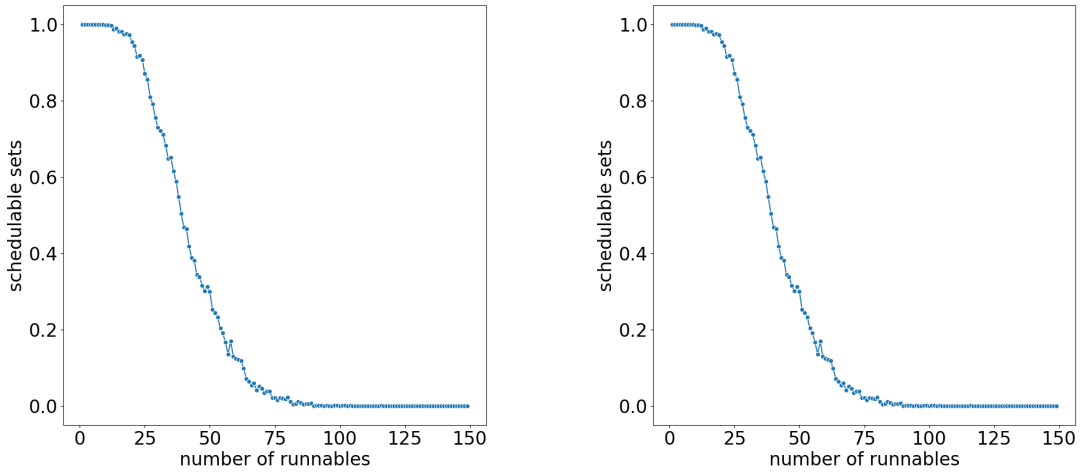
Period	Best		Worst	
	f_{min}	f_{max}	f_{min}	f_{max}
1 ms	0.19	0.92	1.30	29.11
2 ms	0.12	0.89	1.54	19.04
5 ms	0.17	0.94	1.13	18.44
10 ms	0.05	0.99	1.06	30.03
20 ms	0.11	0.98	1.06	15.61
50 ms	0.32	0.95	1.13	7.76
100 ms	0.09	0.99	1.02	8.88
200 ms	0.45	0.98	1.03	4.90
1000 ms	0.68	0.80	1.84	4.75

Table 6.3: Factors for determining runnable BCETs and WCETs based on its ACET. [32]

The authors [32] argue that a typical automotive system comprises around 1000 to 1500 runnables. Assuming an average of 3 runnables per task (Table 6.1), we would expect a task set of around 500 tasks to be schedulable. Nevertheless, choosing periods for tasks as indicated in Table 6.4 results in task sets that are not mathematically proven to be feasible, according to Equation (3.1) [1], once their size reaches around 80 runnables, as presented in Figure 6.1.

Period	**Share**
1 ms	3 %
2 ms	2 %
5 ms	2 %
10 ms	25 %
20 ms	25 %
50 ms	3 %
100 ms	20 %
200 ms	1 %
1000 ms	4 %

Table 6.4: Runnable distribution among periods [32]. Note that shares do not add up to 100% because angle-asynchronous periods are omitted.



(a) 30% L-tasks

(b) 60% L-tasks

Figure 6.1: Percentage of feasible task sets, according to the requested number of runnables. Increasing the number of L-tasks renders the sets slightly more schedulable.

It is worth noting that we are far from matching the mentioned 1000 to 1500 runnables in the system. This is because, according to [32], "the given values already assume a multicore architecture". It is also worth mentioning that we are considering a mixed-criticality model, and

[1] Of course, a task set could be schedulable without this equation holding, but that is the only reliable way we have for validating task sets *a priori*.

how we are setting the different WCETs, described in Section 6.1.3, is likely to be the main reason for what we observe in Figure 6.1.

"Adaptations to smaller and bigger platforms can be achieved by scaling the given values", either by "scaling the total number of runnables, or by scaling the execution times of the runnables". We chose to scale down the total number of runnables.

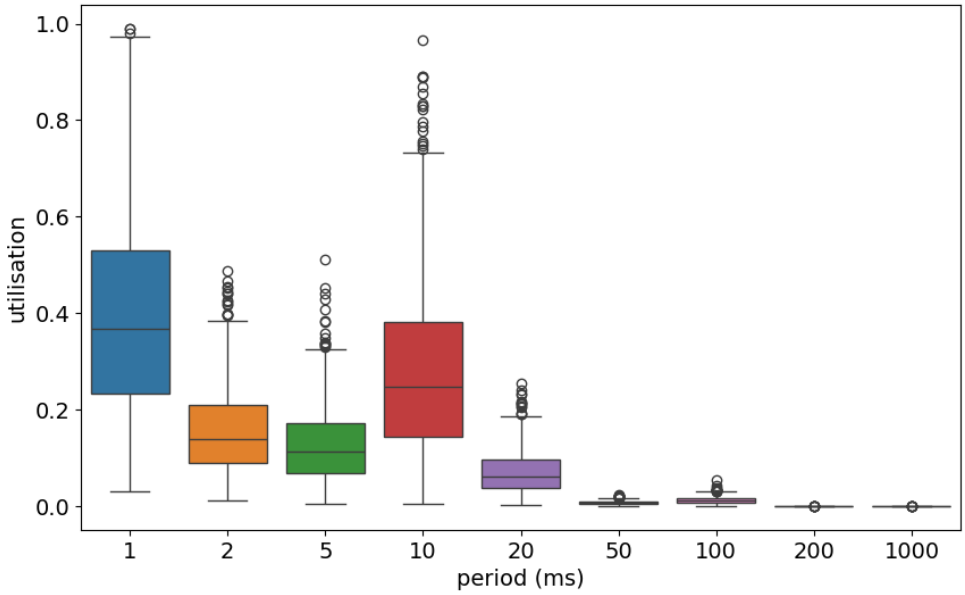


Figure 6.2: Worst-case CPU utilization of generated tasks, according to their period. Note the high utilisation for period=1ms and period=10ms.

Figure 6.2 presents the CPU utilisations of a large set of tasks, according to their period, with properties generated as detailed in Section 6.1.3. The high utilisations of tasks with lower periods contribute to the limit encountered in task set size.

6.1.2 Generating Execution Times of Simulated Tasks

At runtime, the simulator must determine the execution time of an arriving job based on the properties it knows about the corresponding task. This may be done by sampling from a truncated normal distribution centred around the WCET with the BCET and WCET as minimum and maximum, respectively. However, it is not trivial to arbitrate a standard deviation of values from the average.

According to the Project Management Book of Knowledge [25], the *PERT* distribution is useful for estimating the duration of an activity when relying on subjective, informed guesses and dealing with uncertainty factors such as traffic in a car trip which is important for solving

the travelling salesman problem. It assumes the expected value μ to be a weighted average of a pessimistic guess a, a mode b [2] and an optimistic guess c (Equation (6.1)).

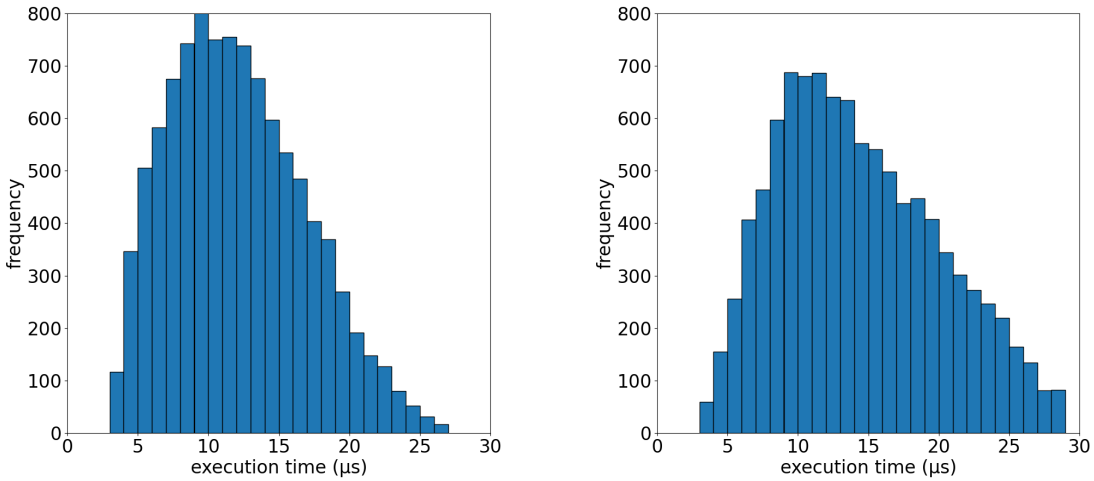
$$\mu = \frac{a+4b+c}{6} \tag{6.1}$$

The analogy with our timing uncertainty, due to factors such as cache hits or processor pipelining, is evident. Thus, we use *PERT* as the main source of execution times during the simulation runtime.

The triangular distribution is a simpler, arguably less reliable alternative to *PERT*. As per Equation (6.2), it places equal emphasis on the extreme values, which are less well-known than the average. In the task model used throughout this dissertation, the WCET are also not very well-known, so we expect this distribution to be less realistic than *PERT*.

$$\mu = \frac{a+b+c}{3} \tag{6.2}$$

Considering a runnable with an ACET equal to 10 μs, WCET equal to 30 and BCET equal to 3 μs, sampling from a *PERT* distribution yields comparable results (Figure 6.3a), over 10000 tries, to sampling from a triangular distribution (Figure 6.3b), using mathematical distribution implementations from the Rust *probability* package [57], although slightly smoother on the *PERT* side, as expected. As such, *PERT* will be used in favour of the triangular and the truncated normal distributions.



(a) PERT Distribution

(b) Triangular Distribution

Figure 6.3: Frequency of sampled task execution times.

While our approach aligns with task scheduling expectations in domains like project management, it fails to consider architecture-induced execution patterns, such as using memory pages in

[2] We use the ACET of a task as the "mode" in the PERT distribution. This is arbitrary, but arguably less than estimating a standard deviation in a truncated normal distribution.

the cache. These patterns, which the agent could potentially respond to due to the dynamic nature of reinforcement learning, are difficult to simulate and analyse without detailed research insights. Therefore, they were not the focus of our studies.

6.1.3 Taskset Parameters

While we detailed suggested task set features in Section 6.1.1, we did not go through how we convert these to actual properties in the generated tasks.

The ACET of a runnable is sampled from a PERT distribution with bounds and mode shown in Table 6.2. Then, we find a tuple (f^B, f^W), where f^B is a random value between f_{min} and f_{max} for the best periods and f^W is a random value between f_{min} and f_{max} for the worst periods, presented in Table 6.3. The ACET of a runnable is multiplied by f^B to determine its BCET and by f^W to determine its $WCET^H$. Finally, the $WCET_L$ is a random value between the BCET and the ACET.

With the properties of the runnables set, the properties of tasks are simply the sum of the properties of its composing runnables.

6.1.4 Simulated Time Resolution

With the ACET, BCET, $WCET^L$ and $WCET^H$ properties set for each task in the task set, one needs to map these time units to the "instants" in the simulator. In a HW timer scenario, that would depend on the frequency of interrupts; here, we can arbitrate a correlation.

Based on the precision of the values provided in Table 6.2, we simulated time with a resolution of 10 ns; in other words, one second of simulated time takes 10^8 iterations by the simulator. That means that the simulation will take some time to run: in our testing, a simulation run for 10 simulated seconds, i.e. 10×10^8 instants, takes about 1 minute on an Intel® Core™ i7-2600, averaged over 10 runs.

The high simulation time is due to the simulator not resorting to a full discrete-event simulation, i.e. "a chronologically nondecreasing sequence of event occurrences" [1]. In the actual implementation, except for idle moments, the simulator uses incremental time progression, which is admittedly slower. Although this does not impact the quality of the deployed agent, it is not ideal for training time and should be improved in the future.

6.2 Computational Requirements of Different Models

Timing and space requirements are just as fundamental to evaluating the feasibility of the solution as the impact on the quality of service. If the time needed to execute the agent hinders the execution of other tasks, there is no point in diminishing mode changes. Furthermore, if the agent is not deployable in embedded devices due to high memory requirements, the solution's potential impact is limited.

To test the computational requirements of the agent on tiny devices, we deployed the system on a Raspberry Pi 4 Model B. While this is not an embedded device and thus does not run an RTOS,

it is considered an entry-level mini-computer by today's standards, with just 2 GB SDRAM and four 1.8GHz processor cores, comparable to a medium-level embedded platform of today.

Raspberry Pi 4 can run Raspberry Pi OS [37], a Debian-based GNU/Linux distribution. Compiling and executing Rust programs is done as usual in a Debian system.

6.2.1 Time Requirements

Whereas time hardly matters while training, it's important that a deployed agent's activation is short enough not to interfere with the arrival of other tasks after the idle period.

On our Raspberry Pi, the activation time of the agent, as presented in Figure 6.4, is in the order of microseconds independent of the size of the policy network, which is good enough considering that the minimum period of a runnable in the task set is 1 ms (as shown in Table 6.3).

Note that preemption does not occur in any way during agent execution and its job does not subtract simulated time in the simulation. Considering the short job runtimes observed, this may not be a major issue; nevertheless, on a production system, the agent must be modelled as a proper sporadic L-task, which is not considered in our proof-of-concept analysis.

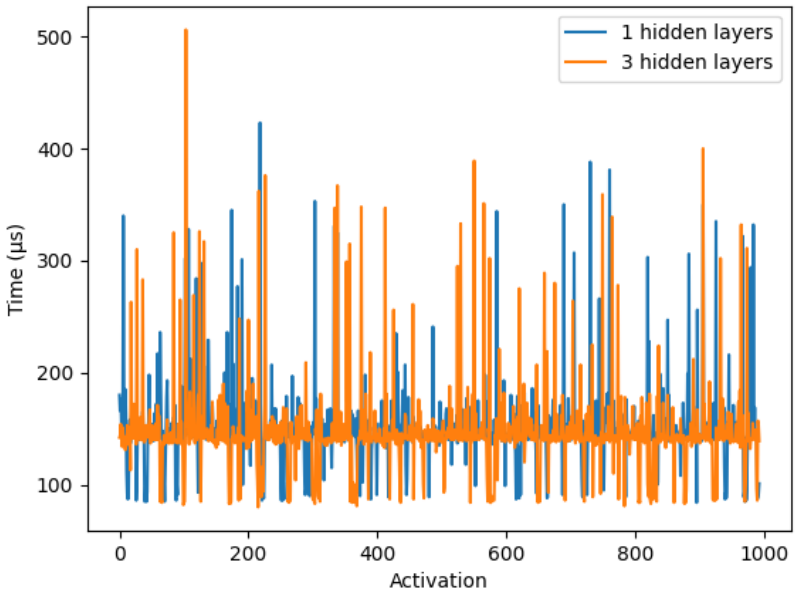


Figure 6.4: Agent's activation times in a Raspberry Pi 4.

6.2.2 Memory Requirements

We resort to Rust's *memory-stats* crate [40] to track the full memory usage of the program, which queries */proc/self/smaps* on a supported Unix kernel. As of now, we are not able to track the memory usage of the agent only, with the chosen architecture. In retrospect, a possibly better design for evaluating this metric would be developing a separate program for the agent only and

having it communicate with the simulator via some interprocess communication mechanism such as named pipes.

Figure 6.5 presents the physical memory usage throughout the program, which is predictably constant since the only relevant memory allocations - the policy network's tensors - are performed at the start of the simulation. The event queue is very thin and can be neglected as far as memory usage is concerned.

Note that the program includes code for gathering statistics, such as these metrics or the reward over time, which also allocates memory. As such, there is room for further minification in production.

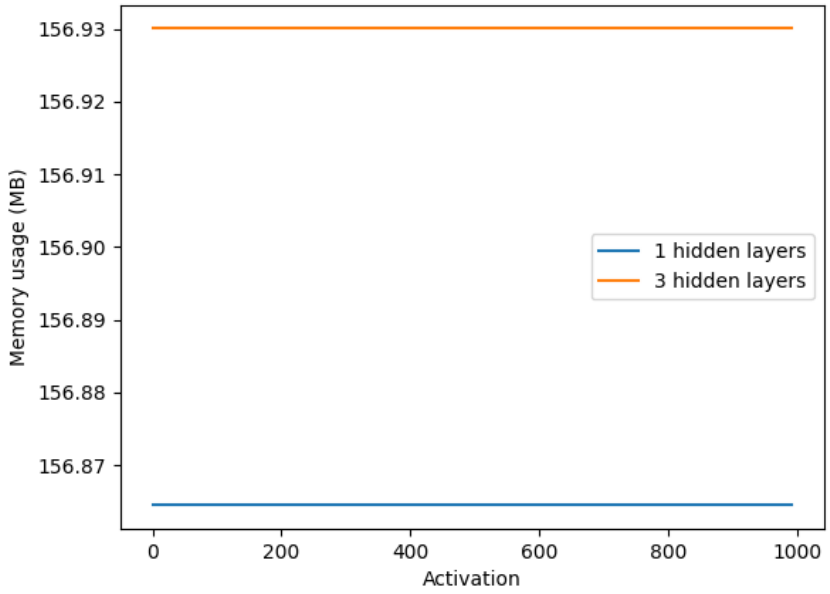


Figure 6.5: Program's memory usage in a Raspberry Pi 4.

6.3 Agent's Impact on the Quality of Service

The agent can impact the overall quality of service of the system by executing actions that result in a change in the number of task cancellations and mode changes. Whether it is beneficial is the question that this section proposes to answer.

To understand the solution's utility in different scenarios, we analysed the agent's behaviour for task sets with 25 runnables and in task sets with 100 runnables. As presented in Figure 6.6 and Figure 6.7, the task sets with fewer runnables show an average lower number of mode changes and a similar average number of task cancellations in a simulation of 10 seconds without any intelligent agent. This hints that the introduction of the agent might be more worthwhile when the number of runnables is higher.

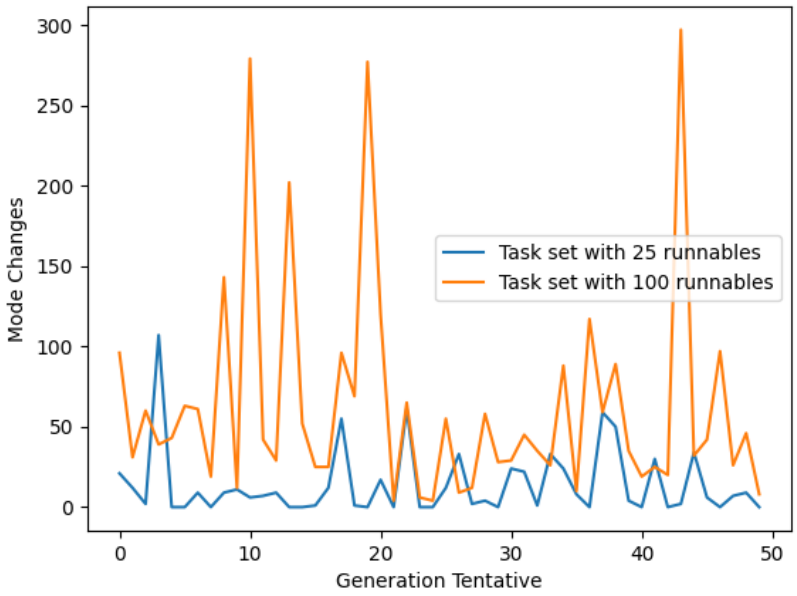


Figure 6.6: Mode changes in Placebo stage.

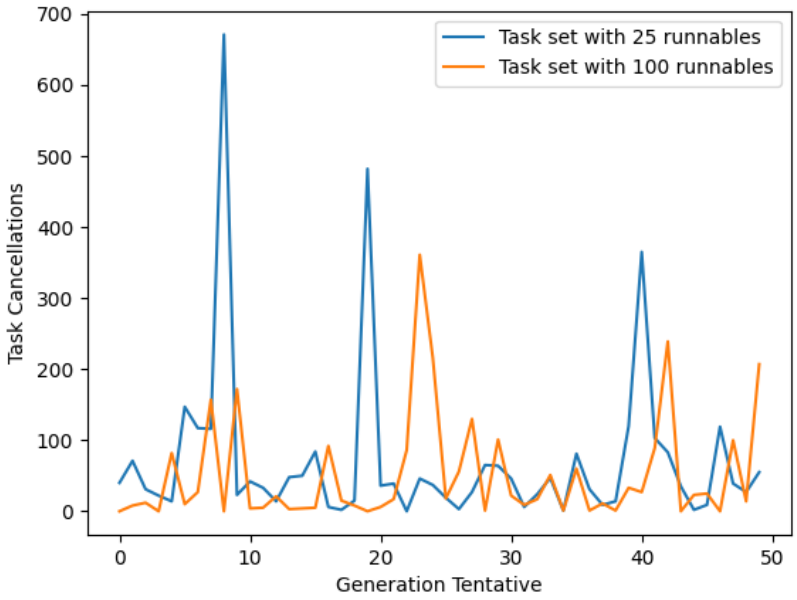


Figure 6.7: Task cancellations in Placebo stage.

The hyperparameters are chosen to train the agent's model for each task set via a parallel grid search, prioritizing an equal reduction in mode changes and task cancellations. In other words, the best hyperparameters are the ones that yield a model of the agent that leads to a higher reduction in task cancellations and mode changes in runtime.

Table 6.5 presents the hyperparameter search space, with n referring to the length of the task set. Note that some hyperparameters are fixed to avoid search time explosion.

Hyperparameter	**Values Space**
maximum memory size	200
minimum memory size	20
gamma	0.99
network update frequency	5, 10, 20
learning rate	5×10^{-5}
hidden layers sizes	$[n/2], [n, n/2], [n, n/2, n/4]$
sample batch size	3, 6, 12
network activation function	sigmoid, relu, tanh

Table 6.5: The search space of hyperparameter values used for hyperparameter tuning. n is the number of tasks in the task set.

Choosing a different set of hyperparameters and building a different model depending on the task set might seem counterintuitive. Traditional ML often hopes that a model can *generalise* to any data set of the same format to behave well in future, unknown scenarios. In this case, we know the entire task set beforehand and know it will not change in production. Thus, we can fit the model for the specificities of the task set without fear of overfitting. Consequently, the model must be retrained from scratch if the task set changes. [3]

The following subsections compare relevant events in the simulator with 25 different, example-generated task sets when running an agent in the reactive stage (with proper hyperparameter tuning) and when not, in a single 2-second simulation [4]. The order of the task sets is the same in all figures; in other words, the task set 0 is the same task set in all test scenarios, and so is the task set 1 and the following.

6.3.1 Impact on Mode Changes

The number of observed mode changes (from L-mode to H-mode), presented in Figure 6.8, is very low when the system is being orchestrated by the agent. While this is not very relevant in simulations where the number of expected mode changes is already low, it makes a considerable difference if the original number of mode changes is high, such as in task sets 11 and 24.

6.3.2 Impact on Task Cancellations

The number of observed L-tasks cancellations, as shown in Figure 6.9, is originally high in the simulation of the example task sets and considerably lower when the deployed agent is running, except in a few task sets (task sets 11, 12 and 21), where the difference is negligible.

[3] We do not consider training time very relevant, but it is not significantly higher than a placebo run since the DQN training steps are very fast.

[4] Due to time constraints, this time is quite low and simulation is only run once.

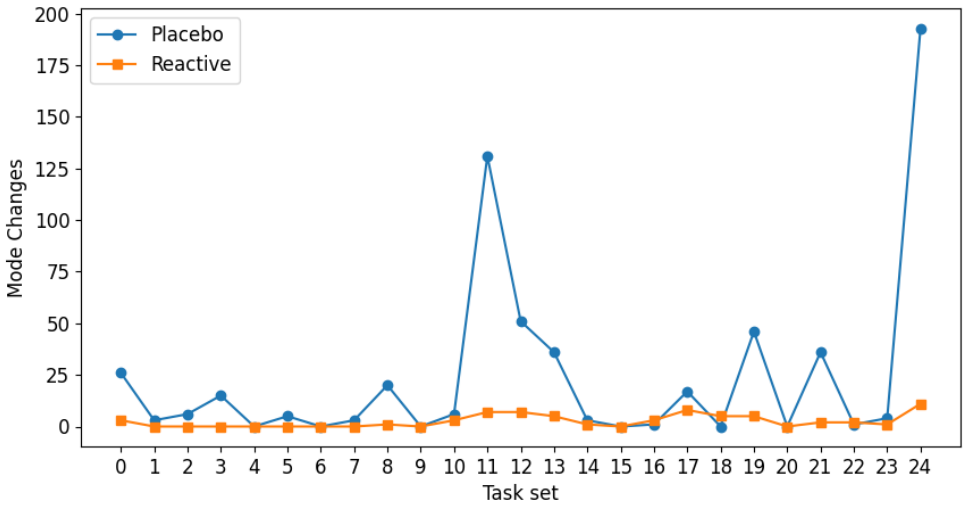


Figure 6.8: Agent's impact on mode changes.

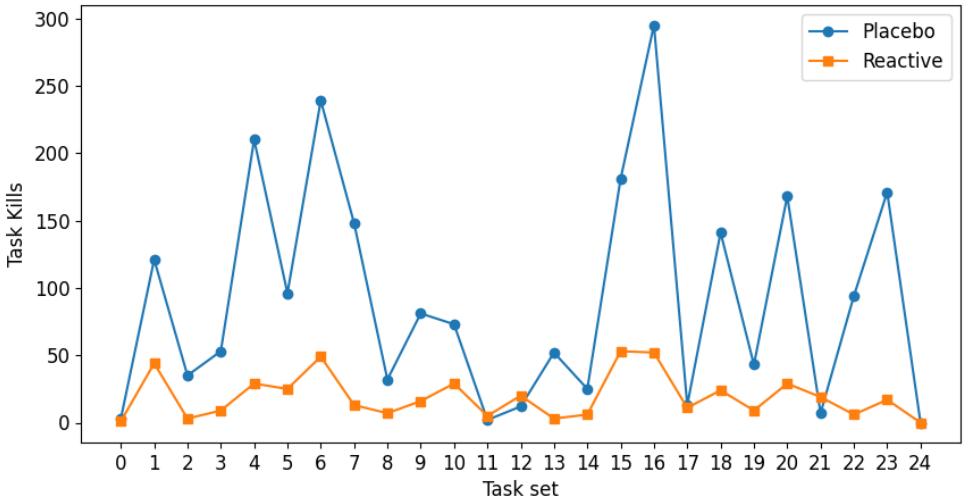


Figure 6.9: Agent's impact on task cancellations.

This translates to a higher quality of service in production since tasks are less likely to be dispensed with, as are the functions needed by the user.

6.3.3 Impact on Task Starts

The impact of the agent on task starts, as presented in Figure 6.10, is a positive side effect of the reduction of mode changes, but also a desire of the agent by itself (recall the small positive reward assigned to these events in Section 5.4.2.3). While the difference is irrelevant in most example task sets, task sets 3 and 13 are impacted considerably.

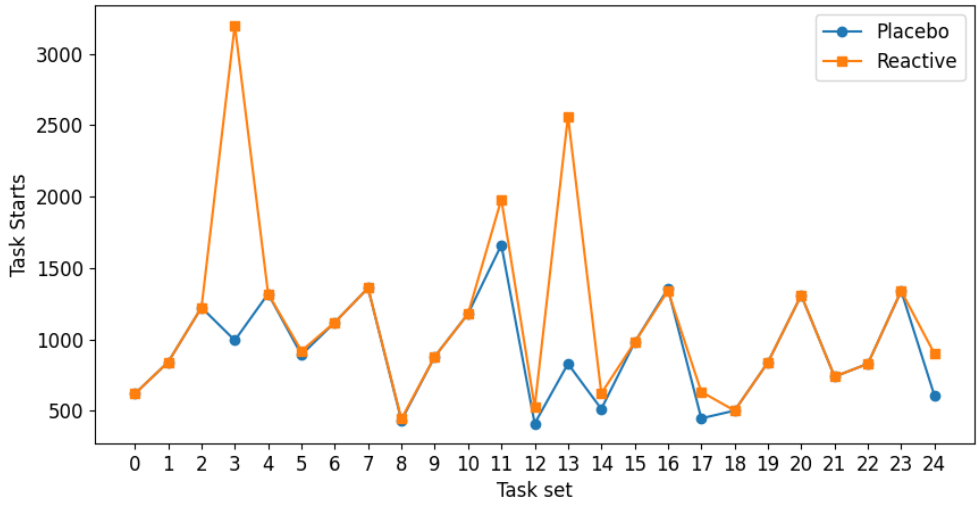


Figure 6.10: Agent's impact on task starts.

6.3.4 Reward Comparison

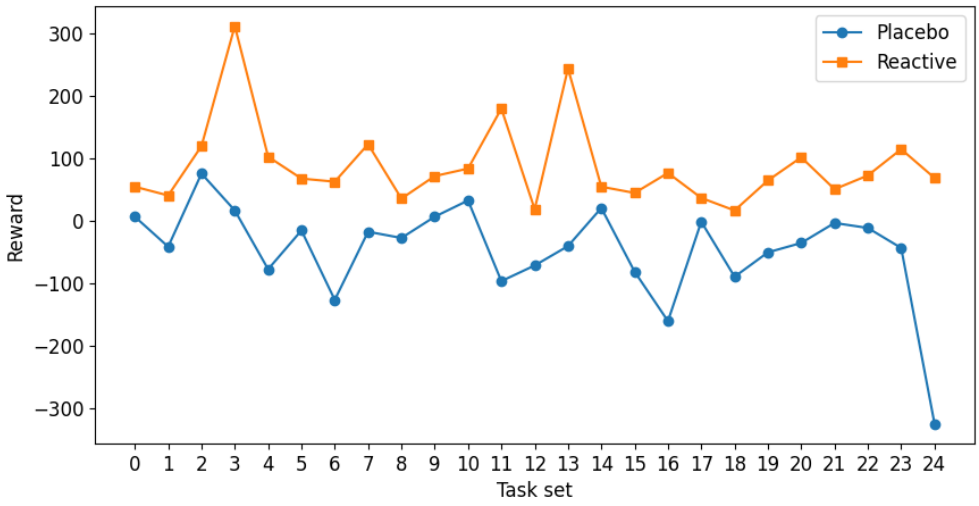


Figure 6.11: Cumulative reward observed in simulation with and without an agent's actions.

The cumulative reward obtained by the agent throughout the simulation is not necessarily related to the schedulability of the system but rather an artefact of the RL process. However, comparing it in reactive and placebo stages, as in Figure 6.11, suggests that the agent successfully reduces the penalty we assign to the scheduling system due to bad events such as mode changes and task cancellations.

Recall that in placebo mode, the agent's actions are ignored, and the rewards are a direct consequence of the events happening in the system as if the agent was not running.

6.4 Discussion and Summary

Our experimental evaluation measures the impact of the agent in various scenarios using simulations with auto-generated task sets inspired by industry research. The PERT distribution serves as the sampler for execution times throughout the simulations.

Our approach aligns with task scheduling expectations in domains like project management, although it is not certain that the PERT distribution is a good predictor for time in computation systems.

The results show that the agent significantly decreases task cancellations in most simulations and can run under 500 microseconds per activation, which is arguably short enough. However, when deployed on less capable platforms, the agent's memory consumption might be an issue.

Chapter 7

Conclusions

We now reflect on this thesis work, from the literature review to the experimental evaluation. We revisit the research questions stated at the start of the document (Section 7.1), highlight the main contributions of the work (Section 7.2) and discuss some of the limitations and strengths of the developed artefacts (Section 7.3). We outline future research directions in Section 7.4 and close with some final remarks in Section 7.5.

7.1 Research Questions Revisited

In Chapter 1, we outlined several research questions which we now address.

1. **Q: How can we use state-of-the-art deep reinforcement learning (DRL) to adjust a real-time system's schedule in runtime?** A: We can use an algorithm such as DQN and develop a way to represent system state based on asynchronous event communication.

2. **Q: How do different DRL models compare, regarding execution time and overall scheduling impact?** A: The different DQN models are similar in time and space requirements. The size of the network architecture bigger is the most determining factor for its performance.

3. **Q: Is the time allocated to the DRL task worth it?** A: As the agent runs only when the system is idle and does it very quickly and with a positive scheduling impact, the allocated time is worth it.

7.2 Main Contributions

Overall, this research work introduced the concept of deep reinforced learning in the scheduling of tasks in mix-critical systems. While the results are limted by the scope of this work, they are encouraging, showing that it is feasible to include machine learning techniques in domains where it has been less prevalent with positive results. Specifically, this dissertation make several contributions, namely:

1. A thourough contextualization of the real-time computing and machine learning fields to newcomers with basic computer science knowledge.
2. A review of state-of-the-art real-time scheduling approaches and deep reinforcement learning algorithms.
3. A study on the constraints required for applying reinforcement learning in an AMC mixed-critical system to ensure it does not hinder certification capability.
4. A novel approach to convert asynchronous environment signals to state in the real-time scheduling domain.
5. The development of a realistic task set generator, based on industry insights.
6. An experimental evaluation of the developed agent's impact on the schedulability of the real-time system.

7.3 Critical Discussion

Our goal was to identify better scheduling solution without relying on hand-crafted heuristics, as seen in other research, to easily generalize our approach to a broader range of scheduling scenarios. This proved not to be an easy task as it is non-trivial to know what to optimise at design time, knowing only a handful of properties of real-time tasks. Furthermore, classic scheduling algorithms proved to be either optimal or near-optimal. Certifying black-box operations in critical systems such as real-time systems was also particularly problematic.

As such, we focused our efforts on optimising the scheduling in systems whose design leaves room for subjectivity and where the miss of a timing deadline is not catastrophic for the correct operation of the system. That is the case of the L-mode of dual-mode systems using adaptive mixed criticality. Even in this scenario, it was crucial to ensure that the schedules derived usign our approach did not hinder the certification capabilities of the system. To put it another way, the design time scheduling feasibility analysis must always be held. After careful analysis, we determined that this implied the agent could only operate when the system was idle, requiring it to reconstruct a state using past events emitted possibly long before the inference task.

Despite that limitation, the experimental results are arguably encouraging. Not only does the agent run fast enough not to raise schedulability concerns, but it also significantly lowers the number of L-tasks cancellations, improving the overall quality of service without any noticeable drawback.

This proof-of-concept trial is, nevertheless, far from optimality. Choosing DQN was a pragmatic decision that allowed us to focus on the intricacies of the domain due to ease of implementation. Still, it required us to discretize the action space, meaning that the agent could be even more useful using a continuous space algorithm such as DDPG. On the other hand, our solution is now limited to the AMC mixed-criticality schema, which may reduce its applicability in the industry where mixed-criticality is not viable or useful.

7.4 Future Work

There are several avenues for future work. Clearly, experimenting with a different DRL algorithm might lead to immediately better results without a comprehensive change in the system's design. Even using the same algorithm, the hyperparameters in place alter the learning potential considerably. Given our time limitations, we focused on optimising a few parameters for a few hours each on a single machine. This could be improved with more time and computing resources.

Similarly, activating the agent more frequently will lead to a quicker application of needed changes, but that will require a different, possibly more sophisticated scheduling analysis and proof compared to the one that led to our decision to trigger the agent only in idle time instants.

Lastly, it is evident to us that our concept has limited applicability. It would be interesting to find out whether it is possible to use deep reinforcement learning in a more general way in the real-time scheduling field, for example, without a mixed-criticality schema.

7.5 Final Remarks

In summary, this research work main objective of optimising the scheduling of mixed-critical real-time systems subject to subjective time budget assignments using deep reinforment learning techniques was clearly met. We could empirically show the efficacy of the developed solution in a handful of scenarios backed up by industry insight. To the best of our knowledge, this approach has not yet been explored and reported in the literature.

Bibliography

[1] *An Introduction to Discrete-Event Simulation - Roger Dannenberg*. URL: https://www.cs.cmu.edu/~music/cmsip/readings/intro-discrete-event-sim.html (visited on 07/09/2024).

[2] Divya Arora, Mehak Garg, and Megha Gupta. "Diving deep in Deep Convolutional Neural Network". en. In: *2020 2nd International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. Greater Noida, India: IEEE, Dec. 2020, pp. 749–751. ISBN: 978-1-72818-337-4. DOI: 10.1109/ICACCCN51052.2020.9362907. URL: https://ieeexplore.ieee.org/document/9362907/ (visited on 12/18/2023).

[3] N C Audsley. "Optimal Priority Assignment And Feasibility oF Static Priority Tasks With Arbitrary Start Times". In: (1991).

[4] Muhammad Ali Awan et al. *Mixed-criticality Scheduling with Dynamic Redistribution of Shared Cache*. Apr. 28, 2017. arXiv: 1704.08876[cs]. URL: http://arxiv.org/abs/1704.08876 (visited on 01/22/2024).

[5] Muhammad Ali Awan et al. "Techniques and Analysis for Mixed-criticality Scheduling with Mode-dependent Server Execution Budgets". In: *ACM Transactions on Embedded Computing Systems* 18.5 (Oct. 8, 2019), 109:1–109:23. ISSN: 1539-9087. DOI: 10.1145/3358234. URL: https://dl.acm.org/doi/10.1145/3358234 (visited on 01/23/2024).

[6] Roberto Bagnara, Abramo Bagnara, and Patricia M. Hill. "The MISRA C Coding Standard and its Role in the Development and Analysis of Safety- and Security-Critical Embedded Software". en. In: *Static Analysis*. Ed. by Andreas Podelski. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 5–23. ISBN: 978-3-319-99725-4. DOI: 10.1007/978-3-319-99725-4_2.

[7] S.K. Baruah, A. Burns, and R.I. Davis. "Response-Time Analysis for Mixed Criticality Systems". In: *2011 IEEE 32nd Real-Time Systems Symposium*. 2011 IEEE 32nd Real-Time Systems Symposium. ISSN: 1052-8725. Nov. 2011, pp. 34–43. DOI: 10.1109/RTSS.2011.12. URL: https://ieeexplore.ieee.org/document/6121424 (visited on 01/23/2024).

[8] Michael W. Berry, Azlinah Mohamed, and Yap Bee Wah, eds. *Supervised and unsupervised learning for data science*. eng. Unsupervised and semi-supervised learning. Cham: Springer, 2020. ISBN: 978-3-030-22474-5.

[9] Alan Burns and Robert Davis. "A Survey of Hard Real-Time Scheduling for Multiprocessor Systems". In: *ACM Computing Surveys* (2010).

[10] Alan Burns and Robert I Davis. "Mixed Criticality Systems - A Review". In: (2022).

[11] Mark C. Mckenzie and Mark D. Mcdonnell. "Modern Value Based Reinforcement Learning: A Chronological Review". In: *IEEE Access* (2022). URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9982454 (visited on 12/26/2023).

[12] Walter Cedeño and Philip Laplante. *An Overview of Real-Time Operating Systems*. 2007. DOI: 10.1016/j.jala.2006.10.016. URL: https://journals.sagepub.com/doi/epub/10.1016/j.jala.2006.10.016 (visited on 01/03/2024).

[13] Jakob Engblom et al. "Worst-case execution-time analysis for embedded real-time systems". In: *International Journal on Software Tools for Technology Transfer* 4.4 (Aug. 1, 2003), pp. 437–455. ISSN: 1433-2787. DOI: 10.1007/s100090100054. URL: https://doi.org/10.1007/s100090100054 (visited on 01/18/2024).

[14] Fengxiang Zhang and A. Burns. "Schedulability Analysis for Real-Time Systems with EDF Scheduling". In: *IEEE Transactions on Computers* 58.9 (Sept. 2009), pp. 1250–1258. ISSN: 0018-9340. DOI: 10.1109/TC.2009.58. URL: http://ieeexplore.ieee.org/document/4815215/ (visited on 01/17/2024).

[15] Matthias Feurer and Frank Hutter. "Hyperparameter Optimization". In: *Automated Machine Learning*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Series Title: The Springer Series on Challenges in Machine Learning. Cham: Springer International Publishing, 2019, pp. 3–33. DOI: 10.1007/978-3-030-05318-5_1. URL: http://link.springer.com/10.1007/978-3-030-05318-5_1 (visited on 05/29/2024).

[16] Scott Fortmann-Roe. *Understanding the Bias-Variance Tradeoff*. 2012. URL: https://scott.fortmann-roe.com/docs/BiasVariance.html (visited on 12/20/2023).

[17] Vincent François-Lavet et al. "An Introduction to Deep Reinforcement Learning". en. In: *Foundations and Trends® in Machine Learning* 11.3-4 (2018), pp. 219–354. ISSN: 1935-8237, 1935-8245. DOI: 10.1561/2200000071. URL: http://www.nowpublishers.com/article/Details/MAL-071 (visited on 11/27/2023).

[18] Sergio Guadarrama et al. *TF-Agents: A library for Reinforcement Learning in TensorFlow*. https://github.com/tensorflow/agents. [Online; accessed 25-June-2019]. 2018. URL: https://github.com/tensorflow/agents.

[19] Rajiv Gupta and Madalene Spezialetti. "A compact task graph representation for real-time scheduling". In: *Real-Time Systems* 11.1 (July 1, 1996), pp. 71–102. ISSN: 1573-1383. DOI: 10.1007/BF00365521. URL: https://doi.org/10.1007/BF00365521 (visited on 01/02/2024).

[20] Larry Hardesty. *Explained: Neural networks*. en. Apr. 2017. URL: https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414 (visited on 12/21/2023).

[21] Hado van Hasselt, Arthur Guez, and David Silver. *Deep Reinforcement Learning with Double Q-learning*. Dec. 8, 2015. arXiv: 1509.06461[cs]. URL: http://arxiv.org/abs/1509.06461 (visited on 01/06/2024).

[22] Sean Hollister. *AMD confirms it's powering the gaming rig inside Tesla's Model S and Model X*. The Verge. June 1, 2021. URL: https://www.theverge.com/2021/6/1/22462660/amd-tesla-model-x-s-plaid-ryzen-radeon-rdna-2 (visited on 01/26/2024).

[23] *Home & Community Safety: Airplane Crashes*. National Safety Council. 2021. URL: https://injuryfacts.nsc.org/home-and-community/safety-topics/airplane-crashes/ (visited on 01/26/2024).

[24] *Installing C++ Distributions of PyTorch — PyTorch main documentation*. URL: https://pytorch.org/cppdocs/installing.html (visited on 05/30/2024).

[25] Project Management Institute. *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*. 7th. Newtown Square, PA: Project Management Institute, 2021.

[26] Philipp Ittershagen et al. "Hierarchical real-time scheduling in the multi-core era — An overview". In: *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*. 16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013). ISSN: 2375-5261. June 2013, pp. 1–10. DOI: 10.1109/ISORC.2013.6913241. URL: https://ieeexplore.ieee.org/document/6913241 (visited on 01/03/2024).

[27] Manuel Jiménez, Rogelio Palomera, and Isidoro Couvertier. *Introduction to Embedded Systems: Using Microcontrollers and the MSP430*. New York, NY: Springer New York, 2014. DOI: 10.1007/978-1-4614-3143-5. URL: https://link.springer.com/10.1007/978-1-4614-3143-5 (visited on 01/26/2024).

[28] M. Joseph and P. Pandya. "Finding Response Times in a Real-Time System". In: *The Computer Journal* 29.5 (Jan. 1, 1986), pp. 390–395. ISSN: 0010-4620. DOI: 10.1093/comjnl/29.5.390. URL: https://doi.org/10.1093/comjnl/29.5.390 (visited on 01/17/2024).

[29] L.P. Kaelbling, M.L. Littman, and A.W. Moore. "Reinforcement learning: A survey". In: *Journal of Artificial Intelligence Research* 4 (1996), pp. 237–285. ISSN: 1076-9757. DOI: 10.1613/jair.301.

[30] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. Jan. 29, 2017. arXiv: 1412.6980[cs]. URL: http://arxiv.org/abs/1412.6980 (visited on 06/10/2024).

[31] Vijay Konda and John Tsitsiklis. "Actor-Critic Algorithms". In: *Advances in Neural Information Processing Systems*. Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 1999. URL: https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

[32] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. "Real World Automotive Benchmarks For Free". In: (2015).

[33] Yann LeCun, Y. Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521 (May 2015), pp. 436–44. DOI: 10.1038/nature14539.

[34] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2016. arXiv: 1509.02971[cs, stat]. URL: http://arxiv.org/abs/1509.02971 (visited on 01/18/2024).

[35] C. Liu and James Layland. "Scheduling Algorithms for Multiprogramming in a Hard- Real-Time Environment". In: *Journal of the Association for Computing Machinery* 20.1 (1973).

[36] Chien-Liang Liu, Chuan-Chin Chang, and Chun-Jan Tseng. "Actor-Critic Deep Reinforcement Learning for Solving Job Shop Scheduling Problems". In: *IEEE Access* 8 (2020), pp. 71752–71762. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2987820. URL: https://ieeexplore.ieee.org/document/9066984/ (visited on 01/26/2024).

[37] Raspberry Pi Ltd. *Raspberry Pi OS*. Raspberry Pi. URL: https://www.raspberrypi.com/software/ (visited on 06/12/2024).

[38] Laurent Mazare. *LaurentMazare/tch-rs*. original-date: 2019-02-16T21:08:44Z. June 4, 2024. URL: https://github.com/LaurentMazare/tch-rs (visited on 06/04/2024).

[39] Megajuice. *English: Diagram showing the components in a typical Reinforcement Learning (RL) system. An agent takes actions in an environment which is interpreted into a reward and a representation of the state which is fed back into the agent.* Apr. 2017. URL: https://commons.wikimedia.org/wiki/File:Reinforcement_learning_diagram.svg (visited on 12/24/2023).

[40] *memory-stats - crates.io: Rust Package Registry*. Dec. 26, 2022. URL: https://crates.io/crates/memory-stats (visited on 06/25/2024).

[41] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. Dec. 19, 2013. arXiv: 1312.5602[cs]. URL: http://arxiv.org/abs/1312.5602 (visited on 01/06/2024).

[42] Kevin P. Murphy. *Machine learning: a probabilistic perspective*. Adaptive computation and machine learning series. Cambridge, MA: MIT Press, 2012. ISBN: 978-0-262-01802-9.

[43] Muddasar Naeem, Syed Tahir Hussain Rizvi, and Antonio Coronato. "A Gentle Introduction to Reinforcement Learning and its Application in Different Fields". en. In: *IEEE Access* 8 (2020), pp. 209320–209344. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.3038605. URL: https://ieeexplore.ieee.org/document/9261348/ (visited on 12/25/2023).

[44] Rui Pereira et al. "Energy efficiency across programming languages: how do energy, time, and memory relate?" In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SPLASH '17: Conference on Systems, Programming, Languages, and Applications: Software for Humanity. Vancouver BC Canada: ACM, Oct. 23, 2017, pp. 256–267. ISBN: 978-1-4503-5525-4. DOI: 10.1145/3136014.3136031. URL: https://dl.acm.org/doi/10.1145/3136014.3136031 (visited on 05/30/2024).

[45] Simon J.D. Prince. *Understanding Deep Learning*. MIT Press, 2023. URL: http://udlbook.com.

[46] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. "Tunability: Importance of Hyperparameters of Machine Learning Algorithms". In: (2019).

[47] Federico Reghenzani, Giuseppe Massari, and William Fornaciari. "The Real-Time Linux Kernel: A Survey on PREEMPT_RT". In: *ACM Computing Surveys* 52.1 (Jan. 31, 2020), pp. 1–36. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3297714. URL: https://dl.acm.org/doi/10.1145/3297714 (visited on 01/03/2024).

[48] *Rust Programming Language*. URL: https://www.rust-lang.org/ (visited on 05/30/2024).

[49] John Schulman et al. *Proximal Policy Optimization Algorithms*. Aug. 28, 2017. arXiv: 1707.06347[cs]. URL: http://arxiv.org/abs/1707.06347 (visited on 01/07/2024).

[50] L. Sha, R. Rajkumar, and J.P. Lehoczky. "Priority inheritance protocols: an approach to real-time synchronization". In: *IEEE Transactions on Computers* 39.9 (Sept. 1990), pp. 1175–1185. ISSN: 00189340. DOI: 10.1109/12.57058. URL: http://ieeexplore.ieee.org/document/57058/ (visited on 01/02/2024).

[51] K.G. Shin and P. Ramanathan. "Real-time computing: a new discipline of computer science and engineering". en. In: *Proceedings of the IEEE* 82.1 (Jan. 1994), pp. 6–24. ISSN: 00189219. DOI: 10.1109/5.259423. URL: http://ieeexplore.ieee.org/document/259423/ (visited on 12/26/2023).

[52] Sanjna Siboo et al. "An Empirical Study of DDPG and PPO-Based Reinforcement Learning Algorithms for Autonomous Driving". In: *IEEE Access* 11 (2023), pp. 125094–125108. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3330665. URL: https://ieeexplore.ieee.org/document/10309299/ (visited on 01/18/2024).

[53] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419 (Dec. 7, 2018), pp. 1140–1144. ISSN: 0036-8075, 1095-9203. DOI: 10.1126/science.aar6404. URL: https://www.science.org/doi/10.1126/science.aar6404 (visited on 11/27/2023).

[54] Daniel S. Soper. “Greed Is Good: Rapid Hyperparameter Optimization and Model Selection Using Greedy k-Fold Cross Validation”. In: *Electronics* 10.16 (Jan. 2021). Number: 16 Publisher: Multidisciplinary Digital Publishing Institute, p. 1973. ISSN: 2079-9292. DOI: 10.3390/electronics10161973. URL: https://www.mdpi.com/2079-9292/10/16/1973 (visited on 05/29/2024).

[55] Ebrahim Hamid Sumiea et al. *Deep Deterministic Policy Gradient Algorithm: A Systematic Review*. preprint. In Review, Nov. 6, 2023. DOI: 10.21203/rs.3.rs-3544387/v1. URL: https://www.researchsquare.com/article/rs-3544387/v1 (visited on 01/14/2024).

[56] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Second edition. Adaptive computation and machine learning series. Cambridge, Massachusetts: The MIT Press, 2018. 526 pp. ISBN: 978-0-262-03924-6.

[57] Ivan Ukhov. *probability - crates.io: Rust Package Registry*. 2024. URL: https://crates.io/crates/probability (visited on 05/27/2024).

[58] Vince Vella. *Deep Q-Networks in Rust from scratch*. Medium. July 25, 2023. URL: https://medium.com/@vincevella/deep-q-networks-in-rust-from-scratch-7c89a7b6c893 (visited on 06/04/2024).

[59] Steve Vestal. “Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance”. In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. 28th IEEE International Real-Time Systems Symposium (RTSS 2007). ISSN: 1052-8725. Dec. 2007, pp. 239–243. DOI: 10.1109/RTSS.2007.47. URL: https://ieeexplore.ieee.org/document/4408308 (visited on 01/18/2024).

[60] Dana Vrajitoru. *Neural Networks*. URL: https://www.cs.iusb.edu/~danav/teach/c463/12_nn.html (visited on 12/24/2023).

[61] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards”. In: (1989).

[62] Ronald J. Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning* 8.3 (May 1, 1992), pp. 229–256. ISSN: 1573-0565. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696 (visited on 01/18/2024).

[63] Keyuan Yu, Kun Jin, and Xiangyang Deng. “Review of Deep Reinforcement Learning”. en. In: *2022 IEEE 5th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*. Chongqing, China: IEEE, Dec. 2022, pp. 41–48. ISBN: 978-1-66547-968-4. DOI: 10.1109/IMCEC55388.2022.10020015. URL: https://ieeexplore.ieee.org/document/10020015/ (visited on 12/18/2023).