TU WIEN Informatics

SMT-as-a-Service at the Edge

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Masterstudium Software Engineering and Internet Computing

eingereicht von

Stefan Holzer, BSc
Matrikelnummer 01625724

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Mitwirkung: Univ.Ass. Pantelis Frangoudis, PhD
Dr. Christos Tsigkanos

Wien, 30. Juni 2022

Stefan Holzer | Schahram Dustdar

TU WIEN Informatics

SMT-as-a-Service at the Edge

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Master programme Software Engineering and Internet Computing

by

Stefan Holzer, BSc
Registration Number 01625724

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Assistance: Univ.Ass. Pantelis Frangoudis, PhD
Dr. Christos Tsigkanos

Vienna, 30th June, 2022

Stefan Holzer	Schahram Dustdar

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Stefan Holzer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 30. Juni 2022

Stefan Holzer

Danksagung

An dieser Stelle möchte ich mich bei meinem Betreuer Schahram Dustdar sowie Mitbetreuern Pantelis Frangoudis und Christos Tsigkanos bedanken. Großer Dank gilt hier den Mitbetreuern, welche mich äußerst schnell mit Feedback unterstützten und mich immer wieder in die richtige Richtung lenkten. Danke für die viele aufgewendete Zeit, für rasche E-Mail Antworten, aber auch für sehr hilfreiche Meetings. Ich fühlte mich während der gesamten Arbeit sehr gut betreut, was für den Abschluss sehr von Bedeutung war.

Neben den Betreuern möchte ich mich auch sehr bei Studienkollegen bedanken, die mir eine andere Sicht auf die Probleme während der Arbeit lieferten und ich dadurch weitere Fortschritte erzielen konnte.

Zu guter letzt gilt auch großer Dank meiner gesamten Familie, Freunden und speziell meiner Freundin und alle anderen, die mir mentale Unterstützung leisteten.

Acknowledgements

At this point, I would like to thank my advisor Shahram Dustdar as well as my co-advisors Pantelis Frangoudis and Christos Tsigkanos. A big thank you goes to the co-advisors who supported me very quickly with feedback and always put me on the right track. Thank you for all the time, for quick email responses and helpful meetings. I felt very well advised throughout the work, which was significant for the completion of the work.

In addition to the advisors, I would also like to thank my fellow students who gave me several times a different perspective on the problems during the work.

Last but not least, I would like to thank all my family, friends and especially my girlfriend and everyone else who gave me mental support.

Kurzfassung

Die steigende Anzahl der Geräte im Internet der Dinge (engl. Internet of Things, IoT) führt zu neuartigen technologischen Anwendungsbereichen wie Smart Cities, Industrie 4.0 oder E-Health. Ein Großteil der dort verwendeten Geräte verfügt nur über sehr limitierte Ressourcen wie Speicherkapazität und Rechenleistung, was bezüglich Kostenfaktor einen Vorteil darstellt, jedoch bei der Verarbeitung von großen Datenmengen in Echtzeit zu Problemen führt. Durch die Einführung von Cloud Computing wurde dieses Problem vorerst mit der Auslagerung der Datenverarbeitung gelöst. Der Umstand, dass die Daten auf den IoT Geräten entstehen und das Ergebnis der Verarbeitung in den meisten Fällen dort wieder benötigt wird, führt zu dem Paradigma von Edge Computing, bei dem die Verarbeitung wieder näher an die Endgeräte rückt.

Neben dieser Entwicklung gewinnt der formale Ansatz von Satisfyability Modulo Theories (SMT) immer mehr an Bedeutung, mit der Modellierung von Optimierungsproblemen oder Verifikationen im IoT Bereich. Diese Probleme sind zum Teil sehr rechenintensiv, was den Einsatz von Edge und Cloud Computing zur Folge hat. Auf der anderen Seite ist ein essenzieller Punkt die Verarbeitung in Echtzeit, welcher die Beachtung der Latenzen bedingt. Dies resultiert daher in die Notwendigkeit eines Systems, welches das gesamte device-to-cloud Kontinuum für die Lösung von SMT Problemen nutzt.

Die vorliegende Arbeit folgt genau dieser Notwendigkeit und präsentiert die Idee eines "SMT-as-a-Service at the Edge"-Systems. Der Kerninhalt der Arbeit ist ein Entscheidungsprozess, der entscheidet, in welchem Bereich des device-to-cloud Kontinuums die SMT-Probleme in Abhängigkeit von den aktuellen Bedingungen am effizientesten gelöst werden können. Dabei gilt es die Architektur des IoT-Umfelds zu beachten, die aktuellen Bedingungen, sowie die Ziele, die verfolgt werden wollen. Dieser Prozess agiert in einem sich wechselnden Umfeld und soll möglichst flexibel in verschiedene Umgebungen integriert werden können. Dazu wird in dieser Arbeit Reinforcement Learning in unterschiedlichen Ausprägungen verwendet, um den Anforderungen standzuhalten. In der Arbeit wird eine Proof of Concept Implementierung vorgestellt, wobei bei der Architektur großer Wert auf den breiten Einsatz in verschiedenen Szenarien und die einfache Konfigurierbarkeit gelegt wird.

In einer umfangreichen Evaluierung der prototypischen Implementierung vergleichen wir verschiedene Deployment Szenarien in einer umfassenden Testumgebung bestehend aus einem echten Roboter, Einplatinencomputern und virtuellen Maschinen in der Cloud.

Dabei kommen SMT Probleme aus verschiedenen Komplexitätsklassen zum Einsatz. Neben diesen Benchmarks stellen wir einen konkreten Anwendungsfall, nämlich “Path Planning for Fog-Supported Robots” vor. Die Ergebnisse zeigen, dass ein System mit intelligenter Entscheidungsfindung mittels Reinforcement Learning in der Lage ist, SMT Probleme effizienter zu lösen. Unser Ansatz ist daher ein vielversprechender Schritt zur Nutzung von SMT im IoT in Kombination mit Auslagerung von Datenverarbeitung.

Abstract

The increasing number of devices in the Internet of Things (IoT) is leading to new technological application areas such as smart cities, industry 4.0 or e-health. Most of the devices used there only have very limited resources such as storage capacity and computing power, which is an advantage in terms of the cost factor, but leads to problems when processing large amounts of data in real-time. With the advent of cloud computing, the problem of limited resources was addressed and solved by computational offloading. However, the fact that the data originate and the processing results are in most cases needed at the IoT devices causes the paradigm shift to edge computing, in which the processing moves closer to the edge devices again. Alongside this development, the formal approach of Satisfiability Modulo Theories (SMT) is gaining importance in IoT by being used to model optimisation or verification problems. These problems can become computationally intensive, which leads to the use of edge and cloud computing. On the other hand, real-time processing is an essential point, which requires attention to latencies. This results in the need for a system that utilises the entire device-to-cloud continuum.

In this work, we follow exactly these requirements and present the idea of an "SMT-as-a-Service at the Edge" system. The core content of the work is a decision making process that decides in which area of the device-to-cloud continuum the SMT problems can be solved most efficiently depending on the current conditions. It is important to consider the architecture of the IoT environment, as well as the goals that want to be pursued by the underlying systems. This process operates in changing environments and should be able to be integrated into different settings as flexible as possible. For this purpose, we use reinforcement learning in various forms to meet the objectives. We present a proof of concept implementation, whereby great importance is attached to the broad use in various scenarios and simple configurability. In an extensive evaluation of the prototype implementation, we compare several deployment scenarios over a fully-fledged testbed featuring a real robot, single-board computers and virtual machines in the cloud. Furthermore, we use SMT problems from different complexity classes. Besides these benchmarks, we present a concrete use case, namely "Path Planning for Fog-Supported Robots". The results show that our system using reinforcement learning is able to solve SMT problems more efficiently. Our approach is thus a promising step towards using SMT in IoT with the combination of computational offloading.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation 1
1.2 Aim of the Work 4
1.3 Structure 6

2 Background **7**
2.1 Edge Computing and Computational Offloading 7
2.2 Satisfiability Modulo Theories (SMT) 10
2.3 Decision Making with Reinforcement Learning 11
2.4 Q-Learning 17
2.5 Deep Q-Learning 21

3 Related Work **27**
3.1 SMT within IoT 27
3.2 Computational Offloading 29

4 Architecture Design **35**
4.1 Requirements 35
4.2 System Design 36

5 Implementation **43**
5.1 Service Architecture Implementation 43
5.2 Communication Module 47
5.3 Configuration Module 48
5.4 Decision Modules 48
5.5 Monitoring Module 53
5.6 SMT-Solver 53
5.7 Deployment 55

6 Evaluation **57**
6.1 Objectives . . . 57
6.2 Experiment Setup . . . 58
6.3 Evaluation Configurations . . . 61
6.4 Experiment Results . . . 70
6.5 Use Case: Path Planning for Fog-Supported Robots . . . 77
6.6 Summary . . . 83

7 Conclusion **85**
7.1 Adherence to Design Requirements . . . 85
7.2 Revisiting Research Questions . . . 86
7.3 Limitations and Future Work . . . 87

A Data Sets **89**
A.1 Simple Data Set . . . 89
A.2 Medium Problem Set . . . 90
A.3 Hard Problem Set . . . 91
A.4 Mixed Problem Set . . . 92

B Use Case SMT-LIB Encoding **95**
B.1 Simple Use Case . . . 95
B.2 Complex Use Case . . . 96

List of Figures **97**

List of Tables **99**

List of Algorithms **101**

Listings **101**

Bibliography **103**

CHAPTER 1

Introduction

1.1 Motivation

With the emergence of the Internet of Things (IoT) novel solutions, technologies and systems were introduced. According to Cisco, 500 billion connected devices are expected by 2030 [Eva11]. This results in extensive changes in different areas and smart cities, smart factories/industry 4.0 or smart homes are becoming state-of-the-art. Such scenarios lead to multiple IoT problems which need to be solved in real-time.

This leads to the new paradigm of edge computing where the computation is moved beyond the traditional boundaries of the cloud to the devices, where data are generated. Gartner predicts that "by 2025, 75% of enterprise-generated data will be created and processed at the edge outside a traditional centralised data center or cloud" [GR17]. According to a report by Grand View Research, Inc. "the global edge computing market size is anticipated to reach USD 61.14 billion by 2028, exhibiting a compound annual growth rate of 38.4%" [Gra21]. Even so, the edge will not displace the cloud, i.e. cloud and edge computing are complementary and should cooperate. This paradigm shift from constantly offloading computational tasks to solving tasks on edge devices is driven by a number of different factors. First, the hardware capabilities for edge devices or devices nearby at the edge are increasing, including better processors and GPUs that could facilitate the execution of computationally intensive tasks, making the offloading obsolete. However, this depends on the environment, as the hardware also brings a cost factor with it. Another factor is the emergence of lightweight virtualisation technology (like containerisation) allowing executing workload at lower resource costs compared to traditional virtual machines commonly used in the cloud [DRK14]. Third, with the advent of new advanced communication capabilities, including multi-connectivity, high-capacity and low-latency links as in 5G, the communication is taken to a new level. According to a November 2021 report from Ericsson, "5G will account for nearly half of all mobile subscriptions by 2027" [CLJC21]. We see that these are driving forces to use the whole

device-to-cloud continuum for computation tasks in a smart system. Finally, the new IoT use cases mentioned above face significant challenges, such as latency, network resource consumption, and privacy-related ones, among others. As we can see, there is no optimal strategy, as offloading to the cloud could bring disadvantages such as additional latency and using the edge infrastructure could bring drawbacks like limited hardware capabilities. Therefore, we need some kind of decision making to always choose the best options, as we propose in our work.

The applications and use cases of IoT are becoming more widespread, as it touches more and more areas. One common example for IoT in smart factories is predictive maintenance [PVB21]. For example, machines can be equipped with sensors, collecting data about the materials used. The data analysis component will alert if checkups or predictive maintenance is required and will reduce downtime and as a consequence production disruption. Another example of smart cities would be smart lighting [CJS13], where IoT sensors collect data about traffic and pedestrians. The goal is to reduce energy consumption, by using the collected data and providing lighting in an optimised way.

As the examples show, these are problems that need to be solved in (near) real-time. This results in the need for an IoT real-time problem-solving strategy utilising the whole device-to-cloud continuum. We propose an end-to-end distributed execution environment with the aim of solving problems that arise at the edge in real-time. This leads to a novel system acting as "Solver-as-a-Service". To concretise the workload and show some concrete architecture and examples, we focus on SMT workloads applied to fog robotics [PD21].

In the area of robotics, a general issue is the checking of environmental conditions. A specific example would be a smart vehicle that is equipped with sensors to check the road texture and adapt the speed based on the measured data. The conditions and requirements can become very complex and result in the necessity of formal verification [EKJ96]. Such problems can be modelled with Satisfiability Modulo Theories (SMT), which is an extension of the satisfiability problem (SAT). Additionally, SMT is a powerful tool for solving complex constraint satisfaction problems [DMB11]. For example, there exist SMT based approaches for the aforementioned motion planning problem for robots with complex constraints [IS19a]. Barret et al. provide an overview and summary of application areas of SMT and highlight the significance of SMT [BKM14]. The presented success stories range from scheduling and optimisation to computer security and software quality. Arxer et al. presents the usage of SMT techniques for different planning problems [EA+18].

Due to increasing complexity, the solver of such problems (called SMT solver) typically requires intensive computation and results in high energy consumption. Furthermore, robots and other IoT devices are limited in computational resources and energy/battery power. An essential requirement of such IoT systems is to support time-constrained applications and real-time processing. Therefore, the approach of computation offloading to the cloud, where resources are abundant can only be used to a limited extent, due to the additionally introduced latency as already mentioned above.

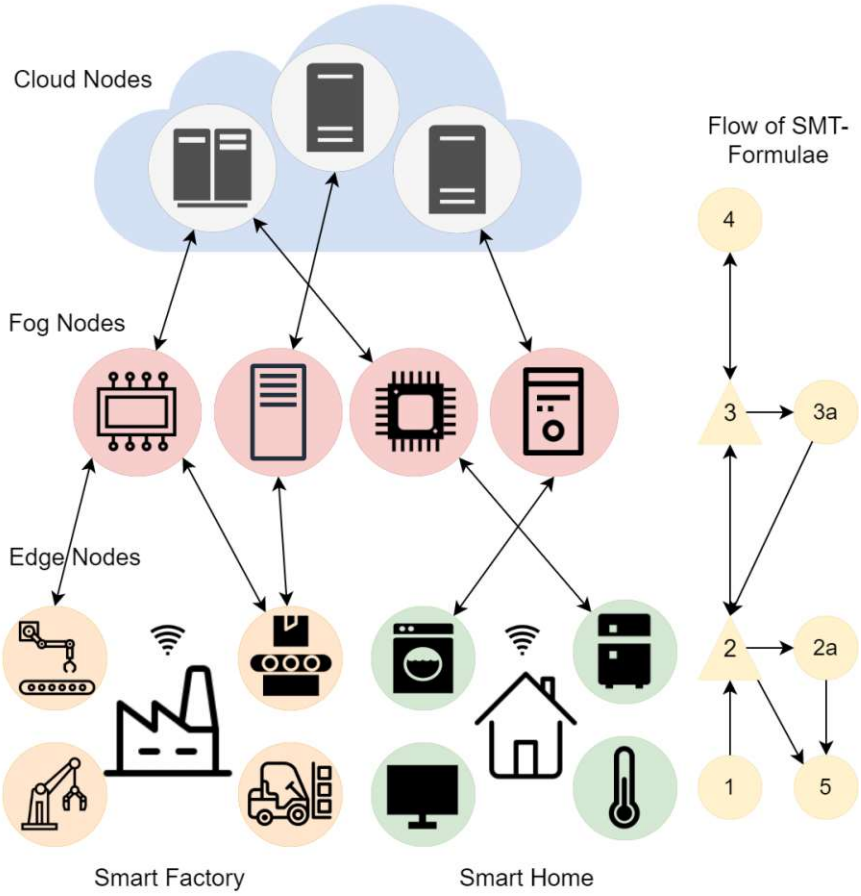


Figure 1.1: Use Case Architecture

1.1.1 Motivating Use Cases

Figure 1.1 shows a typical architecture of the device-to-cloud continuum. On the bottom, the areas of smart factories and smart homes are illustrated as examples. On the one hand, the smart factory could consist of smart robots and smart assembly lines, including smart transport vehicles, and on the other hand, the smart home could include a smart TV, a smart washing machine, a smart heating system or a smart fridge, to name only a few. These devices could be connected to a more powerful server in the smart factory or a single board computer (e.g. a Raspberry Pi) in the smart home. In the cloud, we could use other resources from different cloud providers connected to our nodes in the fog. These are just a few examples, our proposed system can be used in many other areas, such as smart agriculture, smart healthcare or smart cities.

On the right side in Figure 1.1, the flow of the SMT formulae is illustrated, which is described in detail below.

1. Origin of SMT formulae: At the edge devices, the SMT formulae are created. For example, the smart heating system creates a formula that represents all sensor conditions and if the formula is not satisfiable, there is some fault in the system. Another example would be the motion planning of smart transport vehicles. This could depend on the goods to be transported or other vehicles in the smart factory.

2. Decision on edge devices/edge nodes: We can identify three cases. 1. we always want to offload the SMT formulae, as the edge device is not able to call a local SMT solver, 2. we always want to solve the problems on the edge device as there is no stable Internet connection 3. there is an Internet connection and the device can solve the problem itself, but we want to solve it with an optimisation goal, like time or energy efficiency. Therefore, intelligent decision making is needed.

    a) Solving of SMT formulae on edge node: If we decided in step 2 to solve directly on the edge devices, we call the local SMT solver.

3. Decision on fog nodes: If we decided in step 2 to offload, we have again the same decision as in step 2.

    a) Solving of SMT formulae on fog node: If we decided in step 3 to solve directly on the devices, we call the local SMT solver.

4. Solving of SMT formulae on cloud nodes: If we decided in step 3 to offload, we call an SMT solver in the cloud.

5. Further, processing of the results: In step 2a, the results are directly forwarded to the edge device for further processing. Otherwise, the results flow back from the fog nodes to the edge nodes or, if the cloud nodes are used, from the cloud nodes to the fog nodes to the edge nodes.

This means that every time we have a system solving SMT formulae in the device-to-cloud continuum, our proposed service could improve the performance of the system.

1.2 Aim of the Work

The main expected outcome and aim of the work is a fully-fledged system that can solve problems that are modelled as SMT formulae. The problems are created by the edge devices, and the mapped formulae could represent specific desired properties and requirements of the device. Those can be related for example to planning or security/safety. A goal is to find a novel solution that considers the big amount of data and the corresponding necessity of real-time processing. Furthermore, the system needs to define a trade-off between latency and limited computational resources on the edge devices. Regarding this trade-off, a decision algorithm must be designed and implemented, which will determine on the fly whether a node should execute a solving task or forward

the task to the cloud or the edge. The requests (including the solving tasks) arrive at a high rate, which leads to the need for a lightweight and fast solution.

This work aims to answer the following research questions:

- **RQ. 1:** How to architecturally support SMT workloads in the device-to-cloud continuum?
- **RQ. 2:** How to provide offloading decision support for the evaluation of SMT formulae in specific deployment setups and concrete goals?
- **RQ. 3:** How can different decision making strategies improve the performance of SMT formulae evaluation within edge settings?

1.2.1 Contributions

We identify four main contributions in our work, where each attempts to assist in providing answers to the research questions.

1. **Architecture:** Our system is designed to allow separation of concerns, with a clear focus on extensibility and interchangeability. In concrete, we build different modules that are responsible for different parts of the workload and are designed in a way to be deployable in different application environments. This is intended to answer the question of how to manage the SMT workload and what an architecture might look like (see RQ. 1).
2. **Decision Engine:** The core of the whole work is to find an intelligent decision making approach for solving SMT formulae. This contribution helps us toward the second research question (see RQ. 2). We design two different intelligent approaches applicable to different types of hardware with configurable parameters to be able to meet different types of requirements. These approaches are based on machine learning, specifically reinforcement learning.
3. **Proof of Concept (PoC):** Furthermore, we develop a proof of concept solution that handles SMT workload in the device-to-cloud continuum. The system is fully configurable in terms of goals to meet the requirements of the underlying application. In addition, some parts of the system are containerised, allowing for simplified deployment and operation over different types of hardware. For the other parts, the focus is on a very lightweight solution, which is why we omitted the overhead of containerisation. In general, containerisation is considered as a more lightweight virtualisation technique compared to virtual machines, for example, but in extremely constrained environments such as those we target and feature in our PoC, we run our system there natively. Nevertheless, we take care that the deployment and configuration are as simple as possible. This aims to support us in answering all three research questions.

4. **Performance Evaluation:** Finally, we perform extensive benchmarks on our PoC system to be able to answer the third research question (see RQ. 3). We compare our proposed solution and architecture with several baseline approaches in different settings and environments.

1.3 Structure

The remaining chapters of this work are organised as follows. In Chapter 2 we give background information providing a good starting point to dive into the topic and the information is necessary to follow the rest of the work. We introduce the related terminology and definitions. Chapter 3 discusses the state of the art and concentrates on the related work that is relevant to ours as identified during our literature review. In Chapter 4 we present the conceptual solution of our work. In this chapter, we discuss in detail how the different components are designed and provide information about the models which are used. Then, Chapter 5 describes the concrete implementation of the system designed in Chapter 4 and provides further details about the implementation and the underlying technologies which are used. We present our fully-fledged testbed and the evaluation scenarios which are used for a qualitative evaluation of our proposed approach in Chapter 6. In addition, a concrete use case is presented in Chapter 6. The thesis is concluded in Chapter 7 with a discussion of our research questions and our findings. The last section provides an outlook for potential extensions in future work.

CHAPTER 2

Background

In this chapter, we outline the key concepts and paradigms used in this work. Furthermore, it presents the fundamentals necessary to understand our thesis. We introduce common terminology that will be used throughout the work.

We start by introducing the context of our application, including the fundamentals of edge and fog computing paradigms. After that, we outline the field of SMT including the purpose and solving techniques. Finally, we give an introduction to relevant decision making approaches, in particular reinforcement learning.

2.1 Edge Computing and Computational Offloading

2.1.1 Resource-constrained Devices

Statistics show that around 50 billion Internet of Things (IoT) devices will be in use around the world in 2030 [Numa]. It is natural, that all these devices are not equipped with the latest processor or lots of memory. IoT devices have a specific goal, which requires data storage and processing capabilities. Another characteristic of IoT devices is the ability to communicate with other entities. IoT has stimulated many areas like smart factories or in other words Industry 4.0 [SOM14], smart mobility, smart healthcare or smart cities. IoT devices are likely to be deployed in a large number and on that account minimising the production costs is one goal. IoT devices are often be equipped with low-power embedded computational devices like 8- or 16-bit microcontrollers and very little RAM and storage capacities [SPKS12]. For that reason, IoT devices are usually denoted as constrained devices. Since they are used far away from each other, battery power is another common limitation along with computing power. These limitations lead to some considerations and concepts, which are described below and are also an essential part of the whole work.

2.1.2 Edge Computing

In the mid-2000s, the cloud infrastructure was an emerging strategy to offer diverse services over the Internet [Sat17]. All the personal computers acted only as clients and the computationally intensive tasks were done in the cloud. The origin of edge computing was set by the introduction of content delivery networks (CDNs), which use "nodes at the edge close to users to prefetch and cache web content" to improve web performance [Sat17]. Edge computing generalises this concept and has the potential to address other aspects besides response time and latency, such as bandwidth cost savings, data safety/security or data privacy/protection [SCZ+16]. Data are increasingly produced at the edge of the network, consequently processing the data at the edge of the network becomes an upcoming paradigm. In other words, cloud services are moved from the cloud closer to the client/customer. A definition of edge computing follows, once again summarising the most important aspects, and Figure 2.1 illustrates the paradigm of edge computing. From the Figure 2.1 we can see that edge devices can act as data producers as well as data consumers. That means the edge devices do not only request services from the cloud, they can also perform computing tasks. The further capabilities of the edge are presented in the Figure 2.1.

> Edge computing refers to the enabling technologies allowing computation to be performed at the edge of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoT Services. Here, we define "edge" as any computing and network resources along the path between data sources and cloud data centres....The rationale of edge computing is that computing should happen at the proximity of data sources [SCZ+16].

2.1.3 Computational Offloading

To overcome the limits of resource-constrained devices (see Section 2.1.1), computational offloading is the paradigm to transfer resource-intensive tasks to more powerful instances. This could be the edge to use the benefits of edge computing or also the cloud. Computational offloading can be defined as follows:

> Computational offloading is a technique whereby a resource-constrained mobile device fully or partially offloads a computation-intensive task to a resource-sufficient cloud environment. Computation offloading is performed mostly to save energy, battery lifetime or due to the inability of the end device to process computation-heavy applications [TSM+17].

Computational offloading offers numerous advantages, but also has some limitations and disadvantages. Therefore, the decision between offloading and solving the task locally can become complex. The big advantage of computational offloading is, of course, that

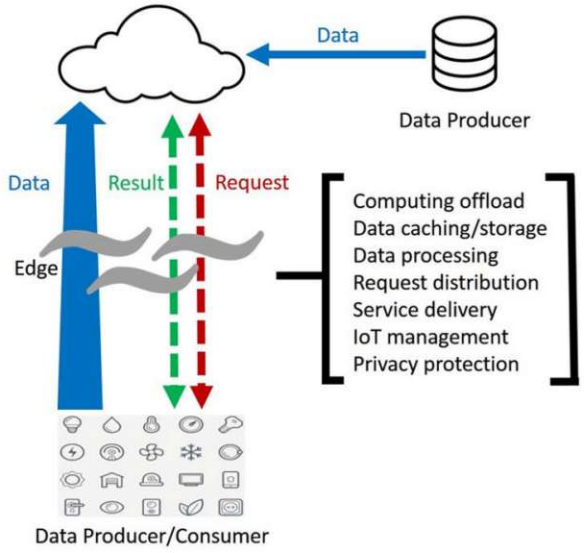


Figure 2.1: Edge Computing Paradigm [SCZ+16]

several tasks are solved much faster as the resources are more powerful, and we have higher computing power. In addition, the energy consumption of IoT devices could be reduced. On the other hand, for simpler tasks, the latency between the devices could play an important role. Moreover, the devices depend on other instances and the network, which could both have down-times. One further aspect is that computational offloading entails certain security risks.

2.1.4 Multi-Access Edge Computing (MEC)

MEC and edge computing are closely related. The main difference is that edge computing is a model (or a concept) and MEC is the practical application (or a standard architecture concept) of edge computing. Taleb et al. describe MEC as follows: [TSM+17]

> Multi-access edge computing (MEC) is an emerging ecosystem, which aims at converging telecommunication and IT services, providing a cloud computing platform at the edge of the radio access network. MEC offers storage and computational resources at the edge, reducing latency for mobile end users and utilizing more efficiently the mobile backhaul and core networks [TSM+17].

The standards institute European Telecommunications Standards Institute (ETSI) provides publications including specifications regarding MEC [goMaECM]. MEC offers cloud-computing capabilities at the edge with the benefits of decreased latency, scalability and much more and typical use cases are IoT, video analytics or location services [ETS].

In the further work, we will make use of the edge computing concept, computational offloading and its advantages. In addition, we will look at the drawbacks of computational offloading and find a way to use computational offloading only when it is the best solution.

2.2 Satisfiability Modulo Theories (SMT)

A central concept used in this work is the Satisfiability Modulo Theories (SMT). A common task is verification, which is necessary for a wide range of systems. E.g., proving the correctness of a program or verifying multiple conditions of sensors in the field of IoT or in concrete robotics. All conditions necessary for verification could often be translated into SMT formulae [BT18]. These formulae can be checked/evaluated with SMT solvers to solve the original verification problem. However, there are many other application areas of SMT in computer science such as planning, model checking or automated test generation.

De Moura et al. from Microsoft Research define SMT and SMT solver as follows:

> Satisfiability modulo theories (SMT) generalises boolean satisfiability (SAT) by adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories. An SMT solver is a tool for deciding the satisfiability (or dually the validity) of formulae in these theories. SMT solvers enable applications such as extended static checking, predicate abstraction, test case generation, and bounded model checking over infinite domains, to mention a few [dMB08].

We want to highlight one aspect mentioned in the definition, namely the fact that SAT could also be used for verification, but SMT is more expressive with the drawback of higher complexity and potential higher execution times of SMT solvers compared to SAT solvers.

For reasons of space, we do not go into the details of the underlying theories of SMT and the algorithms of SMT solver here. In the further work, the understanding of the details is also not necessary, because we will consider the formulae only as our workload, and we will use already existing SMT solvers as well as problems. In the following, we describe only one standard, that is important for the further work and give a brief overview of the most common SMT solvers.

2.2.1 SMT-LIB

SMT-LIB is an international initiative with the goal of advancing research and development in the field of SMT [BST+10]. The initiative was founded in 2003 and in order to achieve these aims, they have set the following specific goals [BST+10]:

- Provide standard rigorous descriptions of background theories used in SMT systems.

- Develop and promote common input and output languages for SMT solvers.
- Connect developers, researchers and users of SMT and build a relevant community.
- Build an extensive library of benchmarks for SMT solvers and make these benchmarks available to the community.
- Collect and promote software tools useful to the SMT community.

The SMT-LIB website[1] provides a large library including benchmarks and problems written according to the SMT-LIB standard, documents describing SMT-LIB in detail, specifications of background theories and links to SMT solvers and tools [SMT].

Every year, a competition[2] is held in which different SMT solvers compete against each other. There are various contributions with benchmarks that can be used to compare the different solvers. These benchmarks are used in our evaluation (see Chapter: 6.2.2).

The big advantage of SMT-LIB is that the creation and description of SMT formulae are independent of the actual solver if the SMT-LIB format is followed. In Table 2.1 there is one column that shows the support of SMT-LIB versions. Details about the syntax, semantics and underlying theories can be found in the documentation [BST+10]. The files containing SMT formulae according to the SMT-LIB specification usually have the extension ".smt2".

Comparison of Solvers

Table 2.1 summarises some features of some well-known SMT solvers [Hö14].

The list of built-in theories is not complete, and details can be found in the description of the SMT solvers. Furthermore, there are often combinations of the theories used. Table 2.2 describes briefly some theories mentioned in Table 2.1.

2.3 Decision Making with Reinforcement Learning

We first describe some general aspects of decision making in computer science and then dive into the topic of reinforcement learning in this section.

2.3.1 Decision Making - General

In this work, we define decision making as the process of choosing an action from several alternatives. If there is only one option, there is no need for decision making. In order to make good decisions, it is necessary to know a lot of information about the current situation, to clearly define the goals to be achieved and to anticipate/assess and plan the future situations/actions. With the advent of management information systems, decision

[1] https://smtlib.cs.uiowa.edu/index.shtml
[2] https://smt-comp.github.io/

Name	Coding Language	SMT-LIB	Built-in theories	Organisation	License
CVC4	C++	v1.0/ v2.0	QF_UF, QF_AX, QF_BV, QF_DL, Q_LA QF_NA, Quantifiers	Stanford University, University of Iowa, ...	BSD
MathSAT 5	C++	v1.2/ v2.0	QF_UF, QF_AX, QF_BV, Q_LA	Fondazione Bruno Kessler & DISI-University of Trento	proprietary
Z3	C++	v2.0	QF_UF, QF_AX, QF_BV, QF_DL, Q_LA QF_NA, Quantifiers	Microsoft Research	MIT
Yices 2	C	v1.2/ v2.0	QF_UF, QF_AX, QF_BV, Q_LA	SRI	GPLv3

Table 2.1: Comparison of SMT solvers

making is supported by technology in lots of businesses [OM06] and becomes a complex part of many systems using different approaches. Albar and Jetter present a model that divides the decision making process into three components [AJ09]: the *decision parameters*, where the parameters are defined that will be used in the next step of decision making. The selection of the *decision parameters* can be influenced by the information of past decisions and knowledge. The next step is the *decision making process*, where all alternatives are evaluated based on the parameters. In computer science, this component can be implemented with different types of algorithms, which will be explained in the next sections. The final component is the *decision implementation*, where the actions are planned and performed.

There are various methods for making decisions in computer science. A simple approach is to use heuristics, i.e. rules of thumb, which are based on common sense. For example, if you have multiple parameters that can be represented as numbers, you could define some ranges/thresholds (based on a rule of thumb), that lead to a decision. In such cases,

Name	Description
QF_UF	Unquantified formulae built over a signature of uninterpreted (i.e., free) sort and function symbols.
QF_AX	Closed quantifier-free formulae over the theory of arrays with extensionality.
QF_BV	Closed quantifier-free formulae over the theory of fixed-size bit vectors.
QF_DL	Difference Logic over the integers/reals. In essence, Boolean combinations of inequations of the form x - y <b where x and y are integer/real variables and b is an integer/rational constant.
QF_LA	Unquantified linear integer/real arithmetic. In essence, Boolean combinations of inequations between linear polynomials over integer/real variables.
QF_NA	Quantifier-free integer/real arithmetic.
Quantifiers	Quantified formulae

Table 2.2: Description of theories [C+11]

decisions can be chained (one decision for each parameter), resulting in decision trees. In a decision tree, there are multiple nodes and branches, where each node represents a decision about a parameter and each branch represents the result of the decision. The leaf nodes represent the final decision [SY15]. Decisions trees are commonly used for classification.

More complex decision making algorithms, which are part of artificial intelligence, are discussed in the next section. For better understanding, we want to briefly introduce the concept of computational (rational) agents here. A computational agent could be anything that makes decisions (acts in an environment), such as a person, a robot, a dog or a computer program [PM10]. The decisions (the actions) are based on past and current perceptions of the environment.

2.3.2 Reinforcement Learning - General

Reinforcement Learning (RL) is an area of machine learning that is a subset of artificial intelligence. The essence of learning is to learn by interacting with the environment and to gain knowledge about cause and effect, or in other words, the consequences of actions [SB18]. With these actions, we want to learn to achieve a goal. However, the actions, the environment and the goal could become very complex. In this section, we focus on the computational approach to learning from actions in a goal-oriented way. The objective is to let the machine learn, how to behave in an environment in order to achieve a goal (or to maximise a certain reward) [SB18]. Reinforcement learning algorithms study the environment and learn to optimise their behaviour. This paradigm is based on learning and building knowledge by interacting with the environment.

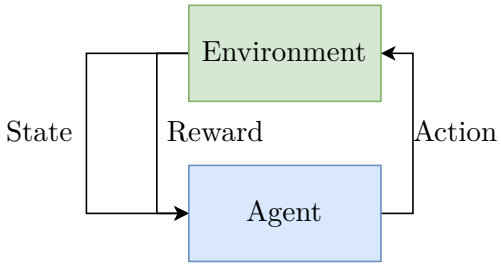


Figure 2.2: The agent-environment interaction in reinforcement learning

Well-known examples are all kinds of games. There is a given environment (e.g. the chessboard and chessmen) and for each chessman you get from the opponent, you get a reward. The main goal is to defeat the opponent. In opposite to supervised machine learning, where the algorithm gets a labelled set and learns based on that set, in reinforcement learning the learner does not know at the beginning which action is the best for a given state [SB18]. In other terms, in reinforcement learning the agent must learn from its own experience. The learner has to explore the environment to get knowledge about which actions will bring the most reward in the long run. Another important aspect to consider is that greedy algorithms only represent a local maximum by always choosing the action with the highest immediate reward. In reinforcement learning, however, the goal is to find the global maximum.

Another fundamental point is that in reinforcement learning, the agent faces the whole problem from the beginning and does not split the problem into sub-problems that may not fit into the "larger picture" and lead to significant limitations [SB18].

2.3.3 Process of Reinforcement Learning

The process of reinforcement learning consists of the following steps and is illustrated in Figure 2.2 [SB18]. The terminology used, is described in Section 2.3.4.

1. Agent (the learner or decision maker) observes the environment (receives a state).
2. Agent decides what action to take based on a strategy.
3. Agent performs the action.
4. Environment receives the action and returns a new state and reward or penalty to the agent.
5. Agent learns from experiences and refines the strategy.
6. Iterate until an optimal strategy is found.

This framework is very abstract and very flexible, resulting in easy application in many problem areas. For example, the iteration step could be any arbitrary time interval, the

actions could be very low-level controls (like increasing the speed of a motor) or high-level decisions like what to have for lunch. Similarly, the environment and thus the states can cover arbitrary fields, ranging from sensing data in a smart factory to symptom descriptions in a smart healthcare setting.

2.3.4 Elements of Reinforcement Learning

In the following, we describe the main elements of a reinforcement learning system [SB18].

- **Agent:** is the entity that interacts with the environment, in concrete performing actions to gain some reward (e.g. a robot).
- **Environment:** the problem situation the agent faces (e.g. a chessboard).
- **Reward:** an immediate return that is given to the agent after performing an action. Negative rewards are also called *penalties*.
- **State:** the current situation the agent faces in the environment. After an action is performed, the environment returns a new state, i.e. a *state transition* takes place after each action.
- **Policy:** is a strategy that defines how the agent should behave at a given time. In other words, it is a mapping between states that faces the agent in the environment and actions they should take in those states. The complexity of this function can vary. In some simple cases, it is a look-up table, while in other cases more sophisticated solutions like a neural network are required.
- **Reward Function:** is a mapping between state-action pairs and a number that represents the reward. This function defines the goal in a reinforcement learning problem, and it defines what is good or bad for an agent given a state-action pair.
- **Value Function:** In contrast to the reward function, which indicates what is good in an immediate sense, the value function specifies what is good in the long term. This means that this function also takes into account the rewards in the future. The goal of an agent is to find actions that give the highest value rather than the highest reward, as thereby the agent can maximise the reward in the long run. It is important to mention that determining a reward is much easier than determining a value. The reason for this is that for a reward, only the current state and the next action are considered, whereas, for values, an estimate must be made from the sequences of observations an agent makes.
- **Model:** mimics the behaviour of the environment. Models can help the agent to determine how the environment will behave. This can be advantageous at the beginning when the agent has no knowledge about the environment without a model. Models can be used for planning, otherwise, the agent acts as a so-called trial-and-error learner.

An essential aspect is that the agent is often a part of the environment. This means that the physical boundary of the agent is not the interface between the agent and the environment. For example, the physical parts of a robot (like the arms, etc.) should also be considered as part of the environment [SB18].

2.3.5 Reinforcement Learning Algorithms

The reinforcement learning algorithms can be classified into model-free vs. model-based, as well as on-policy vs. off-policy algorithms. A brief introduction about the key differences is given in the following points [SB18].

- **Model-free vs. Model-based:** As mentioned above, a model can be used to simulate the environment and thereby plan actions. If no model is available, algorithms are based on trial-and-error to gain knowledge. The advantage of model-free algorithms is that no additional model information must be stored, which can become impractical as the state and actions space grows. On the other hand, model-free algorithms require some training time to explore the environment. The algorithms discussed in the next section fall into the category of model-free algorithms.
- **On-policy vs. Off-policy:** Agents based on an on-policy algorithm use the same policy for learning and performing actions, while agents based on an off-policy algorithm uses different policies. An example of an off-policy method is Q-Learning (see Section 2.4), as actions are selected based on a greedy policy (behaviour policy), but the updated policy (Q-Table) is different from the behaviour policy. An example of an on-policy method would be SARSA (state-action-reward-state-action), which estimates the value of the policy being followed, but for the reasons of space, we do not go into detail here.

2.3.6 Markov Decision Process

In reinforcement learning, the agent performs actions based on the signals of the environments called state. Creating the status information is not trivial, as it is often not clear what information is really necessary. But for the time being, we assume that the creation of the state information is handled by another system for the sake of simplicity. Ideally, we want status information that also compactly represents past impressions without loss of information. Informally said, “A state signal that succeeds in retaining all relevant information is said to be Markov, or to have the Markov property” [SB18]. For example, the current configuration of all chess pieces on a board fulfils the Markov property. The exact information about previous moves has been lost, but is no longer relevant for further play. In other words, the Markov property says that the future is independent of the past given the present.

Now we define the Markov property formally [SB18]. For simplicity, we assume the states and reward values are finite, but this could be extended to continuous states and rewards.

$$P[s_{t+1}|s_t] = P[s_{t+1}|s_1, ..., s_t]$$

In subsequent, we explain the details of the formula. s_t is the current state of the agent and s_{t+1} is the next state. If the equation is given, the system meets the Markov property. This means that the transition from s_t to s_{t+1} is independent of the past, as s_t includes already the needed information of the past states. With the Markov property, we can predict the next state and expected reward given the current state and action.

The approaches described below fulfil the Markov property. In order to understand this better, we first need to introduce Markov decision processes. "A reinforcement learning task that satisfies the Markov property is called a Markov decision process, or MDP" [SB18]. It is a stochastic process with no memory. There is a fixed number of states and the probability to evolve from one state s_1 to another state s_2 is fixed and only depends on the pair (s_1, s_2) [Gér19]. This again means, if the Markov property is given, each state depends solely on the previous state and the transition from that state to the current state. The big benefit is, that no memory is needed, which could become a problem in environments with huge action and state-space. As introduced above, the transition from one state to another by performing an action can result in a reward. The goal of reinforcement learning is to find the optimal state value of any state s, noted as $V^*(s)$, which leads to the Bellman Optimality Equation [SB18]. This applies if an agent acts optimally:

$$V^*(s) = E[R_{t+1} + \gamma V * (s_{t+1}|s_t = s])$$

This is a general equation, and the details of the components are explained in more detail in the concrete example of Q-Learning in the next section.

2.4 Q-Learning

Q-Learning is an off-policy, model-free reinforcement algorithm based on Temporal Differences (TD) [SB18]. Temporal Differences is the approach of learning from an environment through episodes without prior knowledge of the environment. The Q in Q-Learning stands for Quality and is a function. The quality describes how valuable action is in a specific state in order to gain some reward. The Q-Function is based on the Bellman equation and takes two inputs: state (s) and action (a):

$$Q : S \times A \to \mathbb{R}$$

$$Q^\pi(s_t, a_t) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|s_t, a_t]$$

Explanation of the components [SB18]:

- $Q^{\pi}(s_t, a_t)$: Q-Values for the state given a particular state.
- $E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...]$: expected discounted cumulative reward. γ is a configurable discount factor and defines how much rewards in the distant future influence the Q-Value in the immediate future.
- s_t, a_t: given the state and action.

The goal is to maximise the Q-Value, and therefore we define the function for the optimal Q-Value, denoted as Q*:

$$Q^*(s_t, a_t) = E[R_{t+1} + \gamma max Q^*(s_{t+1}, a_{t+1}) | s_t, a_t]$$

2.4.1 Q-Learning Algorithm Process

To store the Q-Values and follow the Q-Learning approach, a Q-Table is used as the data structure. The process described in the following is illustrated in Figure 2.3 [SB18].

- **Initialise the Q-Table:** The Q-Table is initialised with zeros and is a matrix with n columns (where n is the number of actions) and m rows (where m is the number of states)
- **Choose an Action:** The actions are chosen based on the Q-Table. In the beginning, all values are zero, and therefore we need some other concept. This leads us the explore-exploit dilemma, which is described in Section 2.4.2 and one possible concrete concept is the epsilon-greedy strategy.
- **Perform an Action:** After the action is chosen, the agent performs the action and the agent receives a new state from the environment and some reward (or penalty).
- **Measure Reward:** After the action has been performed, the environment returns a reward to the agent, which should be measured.
- **Update Q-Table / Evaluate:** The final step of the iteration is to evaluate the action and update the Q-Table. This is done until the learning/training process is complete and the Q-Table represents a Q-Function that maximises the Q-Value. This update is also called one-step Q-Learning and is defined as followed:

  $$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha * [R_t + \gamma * max Q(s_{t+1,a} - Q(s_t, a_t)]$$

  Explanation of the components:

  - $Q(s_t, a_t)$: old value
  - $[R_t + \gamma * max Q(s_{t+1,a} - Q(s_t, a_t)]$: temporal difference

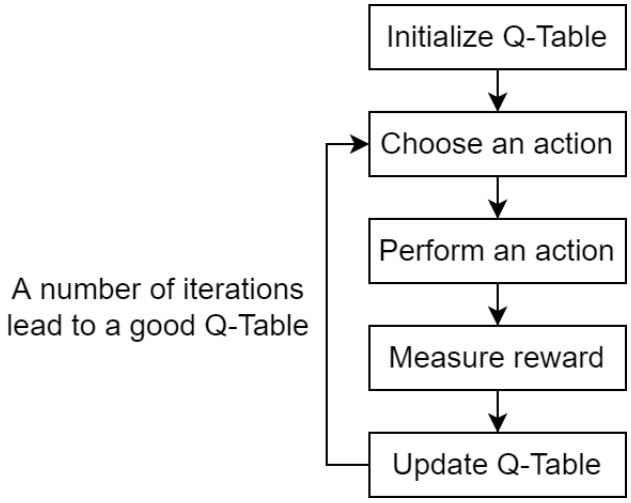


Figure 2.3: Q-Learning algorithm

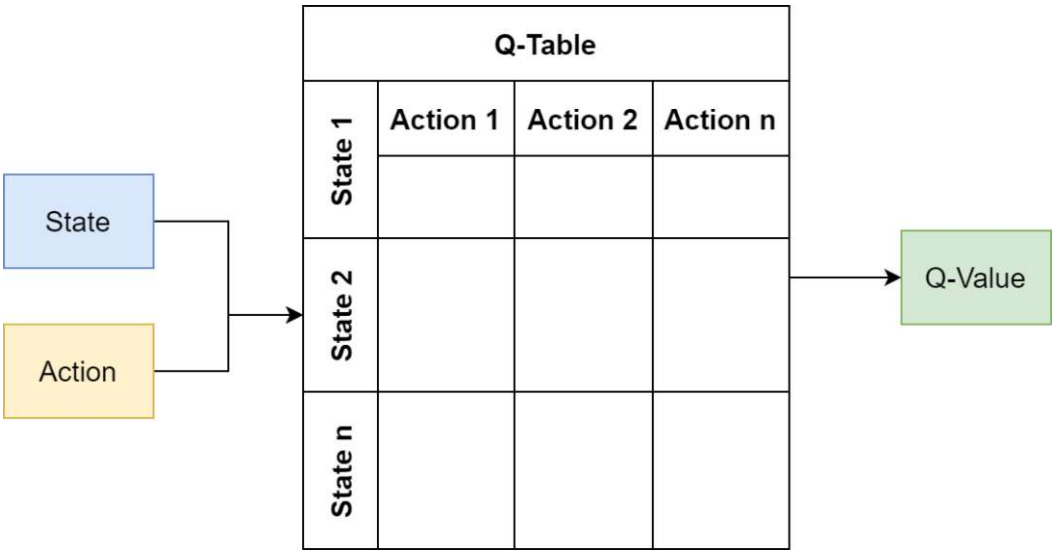


Figure 2.4: Q-Learning - Overview

- $maxQ(s_{t+1,a})$: estimate of optimal future value
- α: learning rate, is a tuning parameter that determines the step size to get the optimal Q-Value

With this approach, we approximate the Q*-function. Figure 2.4 illustrates the structure of a Q-Table. As already mentioned, it is a matrix with rows representing the states and

columns representing the actions. With an input pair (state, action) we get a Q-Value. The idea of the greedy policy here is to select the action with the highest Q-Value.

2.4.2 Exploration vs. Exploitation

In opposition to other kinds of learning, e.g. supervised machine learning, one change of reinforcement learning is the trade-off between exploration and exploitation [SB18].

At the beginning (where all the Q-Values are zero) it is an open question of how the agent should select the actions. There are two different approaches: **Exploration** means to randomly select actions to get more knowledge about the environment and find new information about unexplored states (parts of the environment). On the other hand, **Exploitation** means to select the actions which result in the highest possible outcome based on the current knowledge base. The question now is when an agent should switch from exploration to exploitation, which results in the **explore-exploit dilemma** in reinforcement learning. There are multiple strategies (see [McF18] and [Wenne]) that address the dilemma to get an optimal solution. One is the ϵ-Greedy Strategy, which will be discussed in the next section.

Epsilon-Greedy Strategy

This strategy is one of the most common and simplest algorithms to find a trade-off between exploration and exploitation by choosing between exploring and exploiting randomly. The following pseudocode shows the algorithm:

Algorithm 2.1: Epsilon-Greedy Algorithm

```
p ← random();
if p < ε then
    select random action
else
    select current best action
end
```

That means, with a probability of ϵ any random action is selected (exploration mode) and with a probability of $1 - \epsilon$ the action with the highest reward (or in specific Q-Value) is selected.

Epsilon-Decreasing Strategy

The method is similar to the Epsilon-Greedy Strategy, but in this case the epsilon is decreased over time with a pre-defined decay-rate [Tok21]. Epsilon is calculated as follows:

$$\epsilon_{end} + (\epsilon_{start} - \epsilon_{end}) * e^{-1 * current_{step} * \epsilon_{decay}}$$

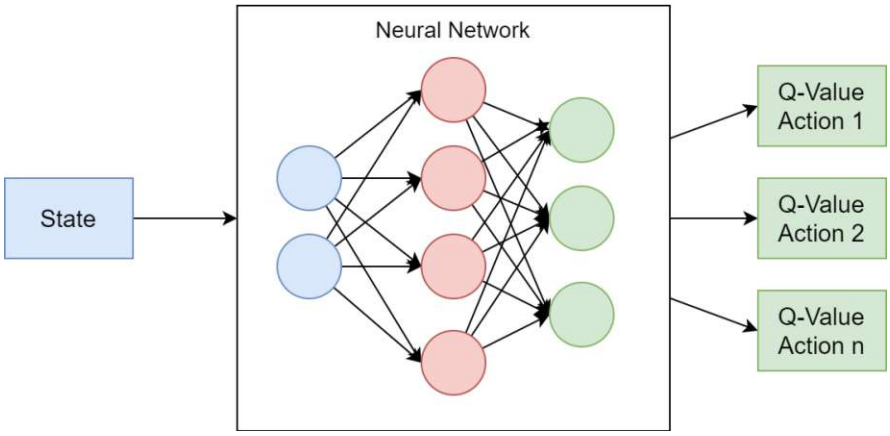


Figure 2.5: DQN - Neural Network Overview

2.5 Deep Q-Learning

The question is what happens when the state and action space becomes so large (e.g. continuous state parameters) that a Q-Table becomes impracticable [Gér19]. For example, we have an environment with 10000 states and 1000 actions per state. This would create a table of 10 million cells. This would quickly get out of hand and memory and time problems would arise. To solve these problems, the idea is to approximate the Q-Values with a deep neural network (DNN), also called Deep Q-Networks (DQN). This method is called Deep Q-Learning [MKS+15].

The whole process is quite similar to the simple Q-Learning. Instead of updating the Q-Table using the Bellman Equation, the network weights in the neural network are updated using the Bellman Equation. To achieve more stability in the learning process, two neural networks are used. In the main network, the weights are updated after each step and after a specified number of steps, the weights are copied from the main network to the target network.

2.5.1 Network Architecture

The input layer always consists of a number of neurons that corresponds to the number of values that represent a state [Lap18]. Then there is an adjustable number of hidden layers with an adjustable number of neurons. This depends on the problem structure. The number of neurons in the output layer is equal to the number of actions, and each neuron defines the Q-Value for one action. The goal of the neural network is to solve a regression problem. Figure 2.5 illustrates the procedure. We have a state as input, which is in this case represented with two neurons, one hidden layer with four neurons and three output neurons representing the different actions. All layers are fully connected. When we set a concrete state as an input, the neural network calculates the different

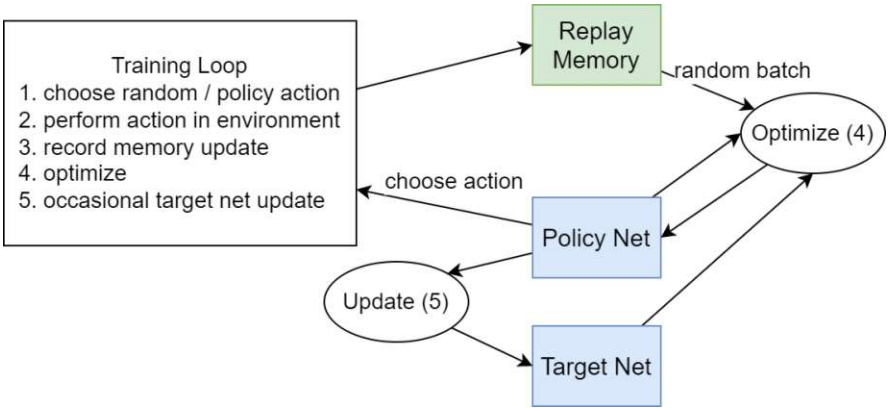


Figure 2.6: DQN - Data Flow [Pas21]

Q-Values for all actions and when the agent is in exploitation mode the action with the highest Q-Value is the best choice.

2.5.2 Experience Replay

The experiences an agent makes are stored $(state, action, reward, state_{next})$. These data are further used in small batches by the agent to learn. Based on this information, we can calculate the loss with functions like the Huber loss function or the Mean Squared Error loss function between the current Q-Values and the target Q-Values with the usage of the Bellman-Equation. The loss functions are described in more detail in Section 2.5.5. After the loss is calculated, the weights are updated with the usage of back-propagation [HN92].

This was only a brief introduction to reinforcement learning. There are multiple other approaches based on the Q-Learning concept like Deep Deterministic Policy Gradient (DDPG) (introduced in [LHP+15]) which combines Q-Learning and policy gradients or, Double DQN (introduced in [vHGS16]) or Dueling DQN (introduced in [WSH+16]).

2.5.3 Training

Figure 2.6 illustrates the data flow in the DQN approach during training. The first step is to choose an action randomly or based on a policy (policy net). After that the action is performed in the (sample) environment resulting in a new state and a reward/penalty. The experience is stored in the replay memory, and the next step is to pick a random batch from the replay memory to perform the training. With this training, the policy net is updated and optimised. The target network is updated occasionally, and this process is done in multiple iterations (episodes) to increase the accuracy of the neural network [Lap18].

In the following sections, some theoretical backgrounds of artificial neural networks is presented, which is important for understanding the design and implementation chapter (see Chapter 4 and Chapter 5).

2.5.4 Activation Functions

The activation function defines for each artificial neuron how the weighted input is transformed into the output. There are several activation functions, the most common being Binary Step, Identity, Sigmoid, TanH, Rectified Linear (ReLU) and Leaky ReLU [BDW18]. Regarding the choice of the most appropriate activation function Di et al. proposed the following:

> In most cases, we should always consider ReLU first. But keep in mind that ReLU should only be applied to hidden layers. If your model suffers from dead neurons, then think about adjusting your learning rate, or try Leaky ReLU or maxout [BDW18].

Accordingly, we will focus on ReLU and Leaky ReLU in the following:

ReLU

The Rectified Linear Unit (ReLU) has the following mathematical definition [BDW18]:

$$f(x) = \begin{cases} max(0, x), & x \geq 0 \\ 0, & x < 0 \end{cases}$$

The computation is much simpler compared to Sigmoid and TanH, which were traditionally used as activation functions [BDW18]. Additionally, ReLU improves convergence by a factor of six times. Therefore, it is very popular and almost all deep learning models nowadays use ReLU. There is a problem called “dying neurons”, which means that the neurons never become active. The reason for this are updates of the weights due to very large gradients. To fix this, Leaky ReLU was introduced.

Leaky ReLU

The difference between Leaky ReLU and ReLU is that Leaky ReLU has a small slope on the negative side instead of a flat slope. The coefficient of this slope is set as a constant and determined before training and not learnt during training [BDW18]. Figure 2.7 illustrates the difference and shows the two different functions as a graph.

2.5.5 Loss Functions

These functions have their origin in statistics and are used to calculate the difference between target and actual values. So the loss function gives a high value if the algorithm

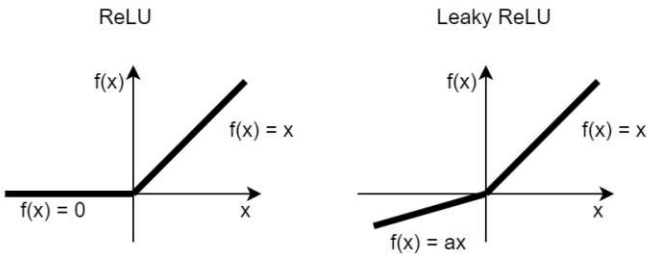


Figure 2.7: ReLU vs. Leaky ReLU

(in our case the neural network) deviates strongly from the actual results. We want to use these functions to find out how good our model is in order to optimise it in a further step. Since our problem in the further work is a regression problem, we focus on the category of regression losses and present the most common loss functions [WMZT22]:

Mean Square Error/Quadratic Loss/L2 Loss

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}$$

The error is measured as the average of the square of the error between the predictions and actual observations. Outliers are more penalised compared to smaller errors.

Mean Absolute Error/L1 Loss

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

In contrast to MSE, only the difference between the predictions and the actual observations (no square) is taken into account here. The big advantage is that it is more robust to outliers, as they are not squared.

Just for the sake of completeness, we mention the Mean Bias Error (MBE), which is the same as the Mean Absolute Error (MAE), with the only difference of that the absolute function is not applied to the values. It should be noted that positive and negative errors can cancel each other out.

In addition, there are other loss functions such as the Mean Squared Logarithmic Error (MSLE) or the Huber Loss function, which tries to combine the robustness of MEA with the stability of MSE [WMZT22]. So when something in the middle (in terms of outliers) is the preferred solution (it depends on the model and the situation), Huber Loss is the way to go.

2.5.6 Optimiser

In the previous section, we presented ways to calculate the error/losses. In this section, we concentrate on ways to minimise the error with optimisation algorithms. The goal of the optimisation functions is to find a global minimum of the loss/error functions. However, the global minimum is not easy to find, but at least a local minimum should be found. In our environment, this can be done by adapting the weights in the neural network. For the sake of simplicity, we highlight the idea of the algorithms and leave aside the detailed mathematical background. First, we introduce one of the most generic optimisation algorithms, called Gradient Descent.

Gradient Descent

To describe this approach, let us use an analogy: Think of a blindfolded man on a mountain who wants to hike down to the bottom of a valley. A good strategy would be to go downhill in the direction of the steepest slope, and that is exactly what gradient descent does. We can think of a gradient as the slope of a function, and gradient descent measures the local gradient of the error function and goes in the direction of descending gradient [Gér19]. The process is done in an iterative way.

One of the most common types of gradient descent is stochastic gradient descent (SGD). In SGD, for each training example within the data set the errors are calculated and the model gets updated.

Adam Optimisation

In the following, we define Adam:

> Adam, which stands for adaptive moment estimation, combines the ideas of momentum optimisation and RMSProp: just like momentum optimisation, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients [Gér19].

Ruder gives a comprehensive overview of gradient descent optimisation algorithms and concludes: “Insofar, Adam might be the best overall choice” [Rud16].

2.5.7 Normalisation

It is a good practice to normalise input data so that each dimension of the input data has the same range (e.g. [0,1]). This has the advantage that the training works faster and the problem of over- or undercompensation of some dimensions is avoided [BDW18].

Batch Normalisation normalises the input for every mini-batch in training, which leads to the possibility of much higher learning rates. The rationale for using this approach is

that “the distribution of each layer’s inputs changes as the parameters of the previous layer change”, which “slows down the training by requiring lower learning rates and careful parameter initialization” [BDW18]

Ioffe et al. also show how the training can be accelerated through batch normalisation [IS15].

There are books on the essentials and fundamental theoretical aspects of artificial neural networks. We have tried to focus on the concepts that will be used and needed in the further work. Further details and information can be found in referenced books and articles.

CHAPTER 3

Related Work

This chapter deals with related work in our field. We will divide this chapter into two sections. Section 3.1 explains the usage of SMT in the area of IoT. The use cases are very diverse and motivate the idea and application of our work. In this section, we will also refer to some papers in the area of robotics where the focus is on motion planning. Section 3.2 shows related work in the area of computational offloading. We profile and present some work on cloud and edge robotics, and further focus on the combination of computational offloading and reinforcement learning.

3.1 SMT within IoT

The usage of SMT is very broad. Basically, the application of SMT is in most cases some kind of verification task. In the following, we focus on the field of IoT and describe various examples in which SMT is used. We will also give examples from the field of robotics and smart factories (Industry 4.0). In the literature, there are many examples that could make use of our idea and PoC system.

3.1.1 Security in IoT

One example comes from Mohsin et al. who present *IoTSAT*, a formal framework for security analysis of IoT [MAH+16]. The attack surface increases considerably as billions of devices based on heterogeneous technologies are connected in different ways. IoT devices often interact with the environment, leading to an expansion of the attack impact from the cyber world to the physical world. In [MAH+16], the authors use a formal modelling approach to create a novel security analysis framework. The system shows attack vectors based on models formulated with SMT. The models are based on two higher-level models, one formalising interactions between IoT entities and the other capturing IoT-specific threat classifications. To detect security issues, the SMT formulae

representing the models are evaluated by invoking an SMT solver. Their concept does not consider limited resources and latency issues, which are essential in the IoT environment. Our concept could use these SMT formulae generated by the core components of *IoTSAT* as workload. The big improvement would be that security threats could be detected more efficiently, as our decision algorithm could be configured to work in a time-optimised way.

3.1.2 IoT and Control

With the advances in IoT, the realisation and deployment of building automation are becoming increasingly popular. In their research article [LBL+16], Liang et al. present a solution for systematically debugging IoT control system correctness for building automation. Building automation is the use of IoT with the aim of increasing user comfort, improving energy efficiency or saving costs, to name a few examples. Such systems could be connected to the heating system, air-conditioning system, lighting, security systems etc. and controlled in a central manner. The control systems often consist of automation rules, also known as IFTTT-style rules (If This Then That), e.g. IF room. temp $<$ 18 THEN room.fireplace = on; [LBL+16]. Besides the automation rules, some policies are defined, and specified as conjunctions of conditions. The main contribution of the work [LBL+16] is to provide a practical way for automated debugging of identified policy violations for users, who are not experts in IoT. To realise this, Liang et al. propose a framework called Salus that transforms rules into SMT formulae. Our solution could improve this work in a similar way as mentioned above.

3.1.3 Motion Planning and Robotics

In the area of robotics, SMT and SAT could be useful for different kinds of problems like scheduling, resource management or motion planning. Nevertheless, robots are often resource-constrained and therefore solving these problems locally could lead to energy or time issues. Therefore, an intelligent mechanism as proposed in our work could improve the solving of the general problems in the field of robotics.

Often the goal for robots is to move around in the environment and reach a goal. In the following paragraphs, we present two works on motion planning.

Hung et al. focus on motion planning with rectangular obstacles using SMT and SMT solvers to find a feasible path from the source to the goal [HST+14]. The robot operates in a three-dimensional space and the random rectangular obstacles are arranged parallel to the X, Y or Z-axis. The problem formulated with SMT is to find a feasible path connecting a starting point S and an end point (goal) G, while keeping a distance d from all obstacles. The robot can only move in straight-line directions, resulting in 90-degree turns. In the paper [HST+14], the planned path is broken down into several connected segments. From the information of the start S, the goal G the obstacles M, the distance d and the space $W \times L \times H$, a set of constraints is obtained, which are mapped to a difference logic satisfiability problem. A further constraint is the minimisation of the path, which would lead to an optimisation problem. To simplify this, an additional

constraint with the path length is added to the SMT problem to limit the travel distance. After specifying the problem, SMT solvers are used to find solutions to these problems and if there is no solution (no path), because not all constraints could be satisfied, SMT solvers can also confirm this.

In their paper, Imeson and Smith. propose a new method for solving multi-robot motion planning problems with complex constraints [IS19b]. In contrast to the work of Hung et al., this paper focuses more on the constraints and the fact that multiple robots coexist in the environment. These planning problems can be very diverse. One example is that multiple robots are distributed over a large and complex physical space and their task is to collect something (e.g. a type of mineral). The tasks would be spatially distributed and interdependent. On the one side, they have the task assignment problem, encoded with SAT and on the other side, they have the path planning problem, encoded as a travelling salesman problem (TSP) (which can be reduced to SAT). These problems are typically coupled, and they propose a framework that can handle additional complex constraints like battery life limitations, robot carrying capacities or robot-task incompatibilities using SMT-based solvers.

Another emerging area is smart factories as the new industrial paradigm. Bit-Monnot et al. present two SMT-based planners for smart factories. As mentioned above, a recurring problem is task planning, and such planning problems can be modelled with SMT [BMLPT19]. Again, the problems could be solved more efficiently with our idea.

3.2 Computational Offloading

Computational offloading is widely used. In this section, we start with computational offloading in the area of robotics, go over to general computational offloading and finish the section with computational offloading with reinforcement learning.

3.2.1 Cloud/Edge Robotics

In this section, we describe the field of robotics invoking cloud or edge resources to gain multiple benefits.

We need to distinguish between networked robotics, a group of robotic devices connected by a communication network [KBS05] and cloud robotics, which is similar to networked robotics, but leverages cloud computing technologies [HTW12]. The aim is to use elastic resources offered by a cloud infrastructure to overcome the resource constraints of robots [HTW12]. By using the cloud and offloading computationally intensive tasks, many new applications for robotics emerge. Hu et al. propose an M2M/M2C communication framework, where the machine-to-machine (M2M) layer is used for communication between machines to form a collaborative computing structure and the machine-to-cloud (M2C) layer is used to utilise a pool of shared computation and storage resources, provided by the cloud. The underlying protocols can be diverse. Another aspect that this paper focuses on is the elastic cloud computing architecture. Three models are

presented: *Peer-Based Model*, where the robots and the cloud instances form a fully distributed computing mesh, *Proxy-Based Model*, where one robot is the "group-leader" communicating with a proxy cloud instance and *Clone-Based Model*, where each robot has a clone in the cloud with which it communicates and to which it offloads tasks. However, the conclusion is that cloud robotics enable "the deployment of inexpensive robots with low computation power and memory requirements by leveraging on the communications network and elastic computing resources offered by the cloud infrastructure" and that many applications can benefit from the cloud robotics approach.

The bandwidth cost and long delays of cloud robotics lead to limitations. Chen et al. propose an edge robotics solution, specifically an industrial robotics system based on edge computing with the deployment of edge nodes near the data sources [CFS18]. The architecture of their robotic system is composed of cloud, edge, and physical resource layers, with the cloud layer forming the core of the system and being responsible for executing the computationally intensive tasks. In addition, the cloud layer is the coordinator and supervisor of the robotic system including several components like service management, path planning etc. The main objective of the edge layer is to filter and pre-process the data, as it is inefficient to transmit all raw data directly to the cloud. The physical resource layer interacts with the real environment and measures data from the environment with sensors and cameras. The results show that Chen et al.'s system "offers better real-time and network transmission performance than a cloud-based approach".

Our work makes use of cloud and edge robotics, but we will use an intelligent mechanism and do not always offload our workload to the cloud or the edge. These aspects are also considered in the next section.

3.2.2 Computational Offloading with specific goals

There is a vast amount of literature on different approaches regarding computational offloading. These are often based on specific goals. In this section, we portray related work with the goal of energy efficiency or latency awareness, as this is also relevant to our work. Zhang et al. provide a solution for energy-efficient offloading for mobile edge computing in 5G heterogeneous networks [ZML+16]. They formulate an optimisation problem to minimise the energy consumption of the offloading system, considering the costs of the task computation and the file transmission. The energy spent on computing mainly depends on the computational capabilities of the mobile edge device and the transmission costs depend on multiple factors like wireless channel state, the size of the computation file and again the mobile device. An additional fact is that interference can reduce transmission rates, which would decrease the energy efficiency [ZML+16]. Zhang et al. propose a complex energy model that takes into account the energy required for CPU cycles and more. Zhao et al. provide another algorithm with a focus on minimising energy consumption and meeting response time requirements [ZZL16]. The following two works concentrate on the time aspect.

Shahhosseini et al. focus on finding a solution for the optimal response time for services in

the IoT paradigm [SAA+22]. They take into account that transmission time leads to an increase in the response time and create a model with a three-layer architecture consisting of a *Sensor Layer*, *Fog Layer* and *Cloud Layer*. In their model they consider the following three questions to optimise the computational offloading problem: where to offload, when to offload, what to offload? Masoudi et al. also address the question of on-device vs. edge computation for mobile services and propose a solution for a delay-aware decision making algorithm to minimise power consumption [MC20].

Our contribution is different to these papers, as the goal can be configured and the underlying model can be freely specified. However, delay awareness and energy efficiency could be two potential goals.

3.2.3 Computational Offloading with Reinforcement Learning

Next to the computational offloading algorithms with specific optimisation goals, there are many examples of computational offloading using reinforcement learning [LGLL18] [CZW+19] [NDW+19b] [NDW+19a]. Below we profile some of them and highlight the differences in our contributions. One difference that applies to all papers is that the workload is not SMT formulae.

Li et al. present a system with the objective of optimising the sum of delay and energy consumption costs [LGLL18]. To achieve this goal, they optimise the offloading decision using an RL-based optimisation framework to consider their time-varying and dynamic environment. Their application domain is Mobile/Multi-access Edge Computing (MEC), which is a viable solution for computation-intensive applications in 5G. We have explained the key concept behind MEC in Section 2.1.4. The core idea of Li et al. is to use Q-Learning utilising a Deep Q Network (DQN) to support a multi-user system environment to reduce delay and energy consumption with different system parameters.

Their concept comprises three different models: *Network Model*, *Task Model* and *Computation Model*. The *Network Model* consists of several user equipment (UE) and one node where a MEC server is deployed. Each UE has a computationally intensive task to perform, which can be offloaded to the MEC or be executed locally. Parameters like the bandwidth of the channel and the upload data rate for each UE are considered in the model. The tasks in the *Task Model* include the computation input data, the total number of CPU cycles required, and the maximum tolerable delay of the task. The *Computation Model* is separated into the *Local Computing Model* and the *Offloading Computing Model*. In these models, the costs of the two approaches are considered, including parameters like required time, energy or transmission delay and this information is combined into an optimisation problem which is "to minimize the sum cost combining execution delay and energy consumption of all users in MEC system" [LGLL18].

The results show that using Q-learning and DQN is much better in terms of the sum of costs than "Full Offload", which means that all UEs offload their tasks to the MEC and "Full Local", which means that all UEs execute their tasks locally.

It is not clear to us where the decision making process takes place. For us, it seems that the decisions are calculated in advance and not at runtime when a new problem occurs. Another limitation is in the decision making, as there is only one MEC server, resulting in a single point of failure. In addition, the tasks are classified in detail, which is also a limitation in dynamic environments.

The limitation that there is only one MEC server does not apply to the work of Chen et al. [CZW+19]. They focus on "designing optimal stochastic computation offloading policies in a sliced radio access network RAN, where a centralised network controller (CNC) is responsible for control-plane decisions on wireless radio resource orchestration over the traditional communication and MEC services". In short, the goal is to optimise computing performance from the mobile users' (MUs) point of view. The decision is influenced by the state of the task queue, the state of the energy queue and the channel qualities between mobile users and base stations. The tasks are modelled similarly to the work of Li et al. [LGLL18], where a computational task is represented by the size of the input data and the number of CPU cycles required. Based on the assumption that the mobile devices of the MUs can be charged wirelessly, they introduced an energy queue in their model where the received energy is stored. To solve the growing state-space problem, they use a DQN [CZW+19]. A major advantage of this solution is that, due to the RL approach, no *a priori* information about dynamics statistics is needed. In their paper, they propose two different concepts. One is called a double deep Q-network (DQN) based reinforcement learning (DARLING) where the idea is similar to our approach, and we discussed the background information in Section 2.5. To the best knowledge of the authors of the paper, they present the first work to combine a Q-function decomposition technique with double DQN, which they call deep state-action-reward-state-action-based reinforcement learning (Deep-SARL) which is the second concept. They use a linear Q-function decomposition technique for simplifying the offloading decision by introducing multiple virtual agents.

A drawback of this solution is that they use a centralised controller and the edge devices cannot decide for themselves. This leads to a single point of failure and the devices are dependent on this controller. Again, the tasks are classified in detail, which is again a limitation in dynamic environments.

In these two papers [NDW+19b] and [NDW+19a], Ning et al. describe the idea of an intelligent offloading framework using Deep Reinforcement Learning in the Internet of Vehicles (IoV) domain. The idea of [NDW+19a] is to use a three-layer offloading framework to minimise the overall energy consumption while meeting the delay constraints of users. The system model consists of three layers: cloudlet, Roadside units (RSU) and fog nodes, where both parked and moving vehicles can be considered as fog nodes. The vehicles can send their tasks to the RSU, which then decides to offload them to the cloudlet or to nearby fog nodes. The main focus of the work is on energy efficiency and the decision on where to offload is always made by the RSUs. Theoretically, tasks can be offloaded to the origin vehicle (where the task was created), as the RSU could decide to offload it to that particular fog node (vehicle). This would lead to unnecessary

transmission overhead, and the issue could be solved with smarter vehicles. However, the results show the effectiveness of their method: “average energy consumption can be decreased by around 60 % compared with the baseline algorithm” [NDW+19a].

To summarise, the existing literature focuses on separate partial problems, like minimising energy consumption, decision making problems regarding computation offloading or system verification on edge devices. There is a lack of research in combining these problems and in solutions that provide system support for the dynamic execution of SMT workloads in a seamless way across the device-to-cloud continuum in a Solver-as-a-Service manner. The focus of this work is to provide a runtime environment to facilitate this.

CHAPTER 4

Architecture Design

In our work, we want to analyse how a software architecture can look like for a system that supports the dynamic and adaptive serving of SMT workloads. To this end, we will first analyse the requirements of our system and then propose a system design that meets these requirements. The design will be provided in a language- and protocol-agnostic manner. The concrete technology used to implement the proposed architecture and design is presented in Chapter 5. The main goal of this chapter is to provide answers to the research questions RQ. 1 and RQ. 2, where RQ. 1 focuses on the architecture aspect and RQ. 2 focuses on the algorithmic aspects of our decision module.

4.1 Requirements

In this work, we follow a component-based software engineering approach. The components, which we call modules in the further work, should be easily interchangeable if we want to use a different strategy or approach in one part of the system.

4.1.1 Resource-constrained devices

Our system should be deployable on different types of edge devices. These edge devices could be very resource-constrained devices, such as IoT devices or single-board computers. To this end, we will focus on a lightweight solution and architecture. In Chapter 6 we will provide a benchmark to show that our system can work properly on resource-constrained devices, such as a Lego Mindstorms EV3.

4.1.2 Multiple Decision Modules

The aim of the work is to find ways to decide whether SMT formulae should be offloaded or solved locally. These decision making algorithms can be based on different methods. Therefore, one requirement is that the system supports multiple decision modules, as the

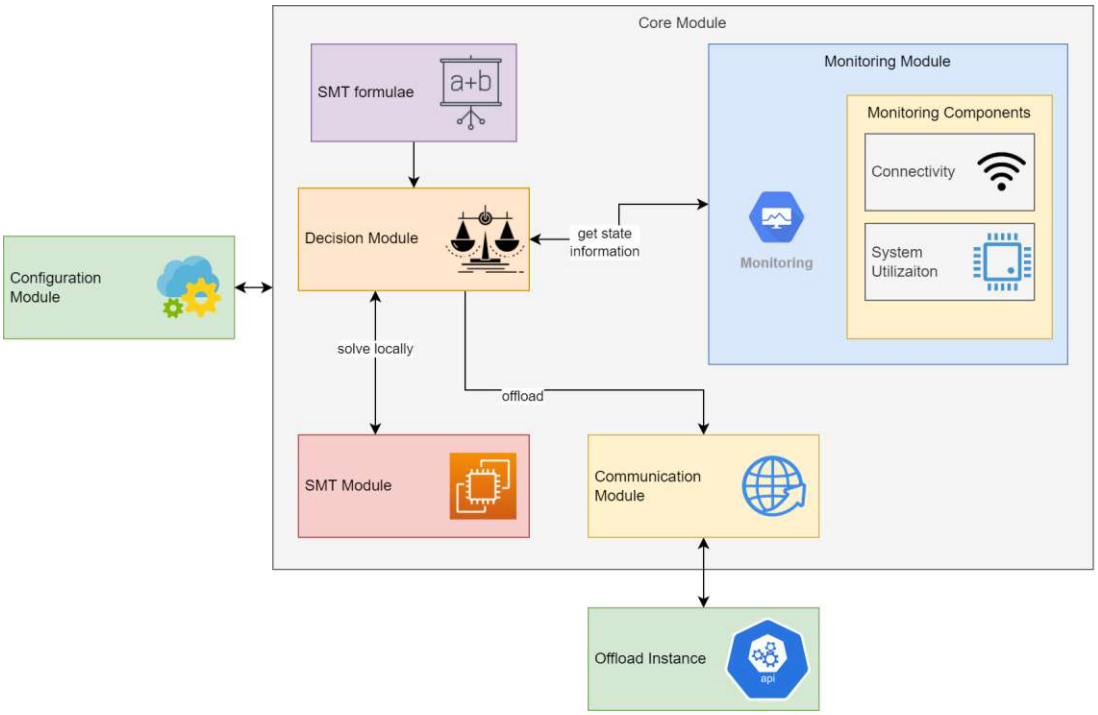


Figure 4.1: Architecture Design

extensibility and interchangeability are important requirements. However, common parts of the software should be extracted, and it should be possible to add another specific approach with very little effort. It is also essential that several decision modules can co-exist, without affecting each other's performance and functionality negatively.

4.1.3 Configuration

Another requirement is that the implementation itself needs to be adapted as little as possible when the environment changes. Therefore, there should be a central way to configure all settings like hyperparameters, offload instances, reward model (all these terms will be clarified in the next sections).

4.2 System Design

Our software is divided into different modules. Figure 4.1 illustrates the different modules and the interactions between them. In the subsequent sections, we will describe these modules in more detail. Figure 4.2 shows the workflow in our architecture in the form of an UML activity diagram. This workflow is described in more detail with the following points:

1. The first step is always the occurrence of a new SMT formula. This can be done at regular intervals or depending on the requirements of the underlying system. The SMT formula can be provided from another system, so the possibilities here are very diverse. Briefly, we just take an SMT formula and do not bother with the creation and so on.

2. The next activity is to get the latest state information, which is relevant for decision making. This results in a call to the monitoring module (see Section 5.5).

3. After determining the state, we call the decision module to make a decision between offloading or solving locally. The decision on offloading could include a decision on which instance to offload, but this information is omitted for simplicity.

4. If we decide to offload, we forward the SMT formula to the communication module, which offloads the SMT formulae to the provided instance.

5. After offloading, we start again at the first activity.

6. If the decision is to solve locally, we pass the SMT formula to the SMT module that solves the problem. After that, we are in a final state because the SMT formula is solved.

7. The last step is that the result is recursively returned to the source instance. Due to the offloading mechanism, several instances could be invoked, but the original instance needs the result.

4.2.1 Configuration Module

This component handles the configuration. All other modules depend on the configuration module. The other way round, the module is loosely coupled and has no dependencies on other modules. Here we want to configure all the other modules and the general software.

In the sections below we present the modules in the core module.

4.2.2 Communication Module

The communication module is used for communication with other devices. If an SMT formulae will be offloaded, this is done via the communication module. Because of the loosely coupled module, the way of communication can easily be replaced, e.g., if we want to change the underlying protocol.

4.2.3 Decision Module(s)

The decision module receives SMT formulae and decides how to proceed with them. Depending on which decision mode is active, the formulae are forwarded to the corresponding submodule. In general, external modules can also be called, to take over further

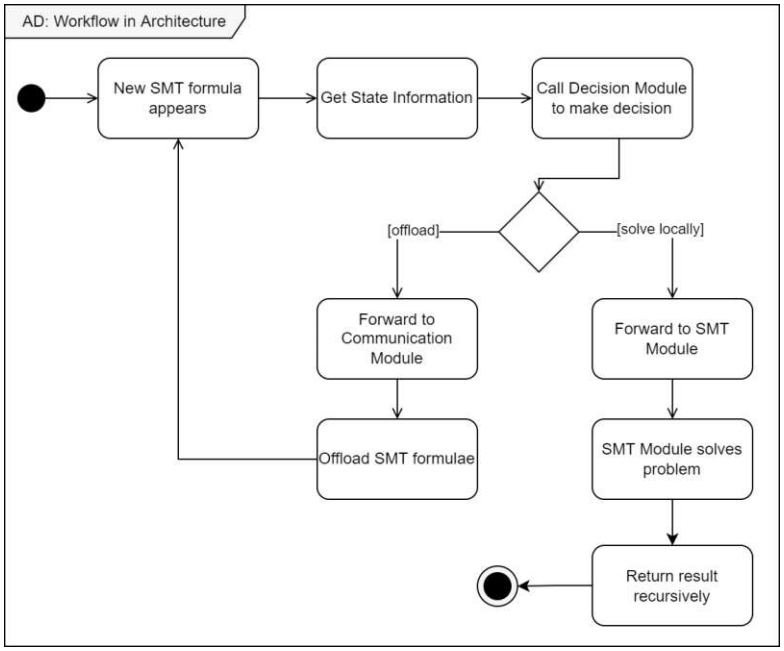


Figure 4.2: Activity Diagram: Workflow in Architecture

processing. If the active decision mode is Q-Learning, the formula is forwarded and further processed by the submodule Q-Learning. The same applies to the decision mode DQN and the submodule DQN. In our architecture, we provide these two approaches. As already mentioned, further approaches could easily be added. Our solution is based on reinforcement learning (see fundamental information in Chapter 2) for the following reasons. The decision module should work in very different and changing system environments, which rules out heuristic approaches. Furthermore, it makes the whole system more flexible and applicable for different applications, as it can be configured to focus on the needs of the supported underlying application. We start with some general details about the design of reinforcement learning.

Reinforcement Learning - General

There exists multiple ways to do reinforcement learning. The parts that are common and necessary for each approach are located in the reinforcement learning module. The specific aspects are located in the associated submodules. In our architecture, there are two common aspects that are needed for both specific submodules:

- **Balance exploration and exploitation**: An essential aspect is to define how the agent should choose the actions at the beginning (see Section 2.4.2). We will use an Epsilon-Greedy Strategy, as this is one of the most common and simplest

methods. All the parameters (exploration rate, decay rate and end of epsilon) are configurable.

- **Environment**: Information about the state and the rewards can be handled by the generalised module. The reward model can be fully configured. In each submodule, there is an environment manager that handles approach-specific aspects.

Below we describe the elements of reinforcement learning in our architecture:

- **Environment/State**: The state information is built from the problem complexity of the SMT formula and the connectivity (more specific details below) of the device.
- **Action**: The actions are in general to solve locally or to offload. If there are multiple instances to offload, we have one action for each instance.
- **Reward/Reward Model**: The reward function could be easily changed. However, we focus on the time-aware mode (see Section 6.3.3). In our case, we define the reward with: configurable constant number of seconds - required time to solve the problem in seconds. That means if, it needs more than the configurable number of seconds, we get a negative reward (= penalty).

One design decision we want to highlight here is the hyperparameter discount factor (γ). The discount factor determines how much the agent cares about rewards in the distant future relative to those in the immediate future. We have a completely myopic agent with a discount factor of 0, due to the following reason. We do not know how the environment changes with our actions, as we do not know the future problem complexity of the SMT formulae and also the connectivity. That means the actions taken now, do not influence the rewards in the future, which lead to the fact that we can not care about rewards in the distant future. Therefore, a myopic and greedy agent is the only sensible design decision and solution.

Q-Learning

For simple Q-Learning, we need a Q-Table. When we speak of Q-Learning in the following work, we mean Q-Learning with a Q-Table. This Q-Table has the following attributes:

We define the following ratings for the states: poor = 0, fair = 1, average = 2, good = 3, excellent = 4

- It has a size of 25 rows and three or five columns (depends on the device).
- The rows represent the state. As we have two dimensions (connectivity + problem complexity) and five rating options, we get $5^2 = 25$ rows. For the sake of simplicity and to keep the Q-Table small and simple, we use only one random connectivity information as the state. Of course, we could expand the state with all connectivity

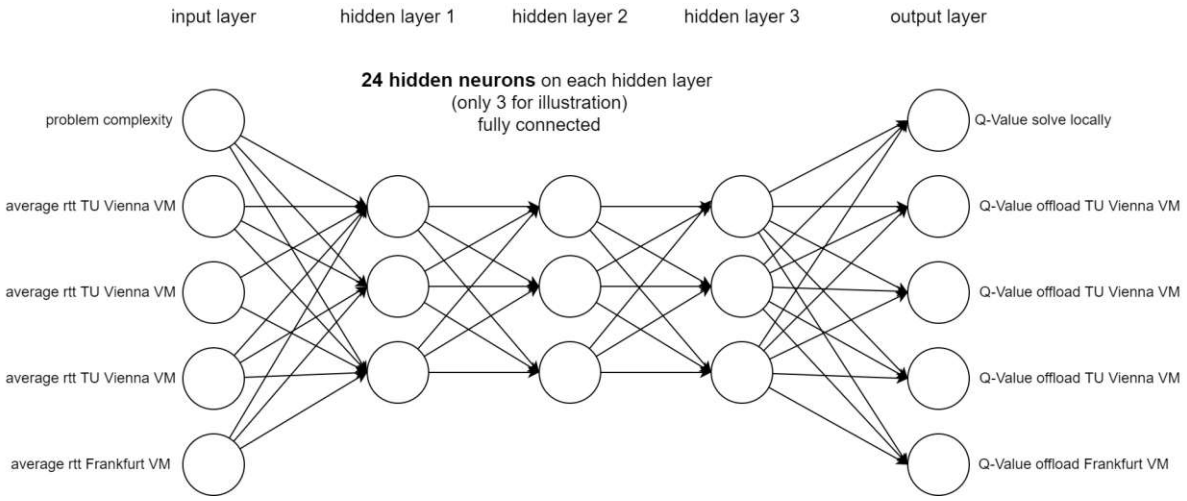


Figure 4.3: Design of Neural Network for DQN

information to all instances, but the Q-Table would become very fast very big, which would lead to performance issues, which is in contradiction of the requirement of a lightweight solution.

- The columns represent the actions. As this depends on the number of possible offload instances, this can vary.

Deep Q-Learning (DQN)

In contrast to the Q-Table, we use here the connectivity information to all instances. This leads to some given facts of the architecture of the neural network, as the number of input neurons must be equal to the number of state dimensions and the number of output neurons must be equal to the number of actions. Figure 4.3 shows a simplified version of the neural network used. To save place, the hidden layers show only three neurons but actually, there are 24 neurons. As described in Section 2.5.1 the number of input neurons is predefined. Since we use the problem complexity and the four different average round trip times of the different cloud instances as state information, we get five input neurons. The different actions are solving locally and offload to one of the four cloud instances. This results in five output neurons, which is also illustrated in Figure 4.3. Then there are three hidden layers with 24 hidden neurons on each hidden layer. Another aspect is the connection of the different layers. We use a fully connected neural network, that results in an architecture where all neurons from one layer are connected to all neurons in the next layer. This architecture is commonly used for regression problems.

Loss Function In our system, we use the MEA (mean absolute error) as the loss function. A brief discussion of the various loss functions can be found in Section 2.5.5. At this point, we would like to emphasise again the advantages of MEA: All errors are weighted

on the same linear scale (we do not put too much weight on outliers) [WMZT22]. Due to the non-deterministic environment, outliers might occur, but we do not want to give them too much weight and therefore MEA is a good choice.

Optimiser To optimise the model, Adam (see Section 2.5.6) is used as the optimisation algorithm in our architecture.

Activation Function As an activation function, we are using Leaky ReLU, because ReLU is the most popular activation function and Leaky ReLU faces the problem of dead neurons (see Section 2.5.4).

4.2.4 Monitoring Module

The monitoring module is used to monitor the environment (the devices on which the system is running). This information is used for decision making. The monitoring component could provide any information which could be useful for the decision making. This could range from battery-level or CPU-usage to connectivity to different hosts in form of average round trip time. The monitoring component is asynchronously executed in a background thread and gets the state information periodically. The time period can be configured. The benefit of doing this in the background and not at each decision is that the cost of the decision itself can be reduced, as the operations for monitoring are more time-intensive. If the device is more powerful, the period could be reduced. This would lead to more up-to-date monitor values for the decision module.

4.2.5 SMT Module

The SMT module consists of two parts. The first one is an interface to the native SMT solver, which can be called by the other modules. The second one provides an endpoint to which other devices can offload their SMT problems. That means that the offload instance in Figure 4.1 could be a complete instance including the core module and configuration module or only an SMT module, which solves the SMT formulae.

CHAPTER 5

Implementation

This chapter presents the implementation details of the proof of concept of the proposed design in Chapter 4. In particular, it starts with some general implementation details, describing the used programming language and packages. Afterwards, a description of the communication mechanism and the different modules in detail follows.

5.1 Service Architecture Implementation

This section presents some general implementation details relevant to the modules described later.

5.1.1 High-Level View

Figure 5.1 illustrates an overview of the architecture, concretising implementation details in contrast to Figure 4.1. As stated in Section 5.1.2, we use Python as the programming language. This is also indicated in the core module and in the SMT solver instance by the Python icon. We use CVC4 as the local native solver and PySMT on the cloud instances (see more details in Section 5.6). When an SMT formula is offloaded to another node via the REST client module, there are two options (conditional control). If the node is a final node, it is offloaded to an SMT solver instance running in a Docker container that solves the problem with PySMT. Otherwise, it is offloaded to an instance, which in turn forwards the problem to the decision module in the core module. The communication is done via REST Endpoints, which are realised with Flask. It is also important that the arrows go in both directions. This means that the results are returned to the original requesting node. This can also happen via several nodes.

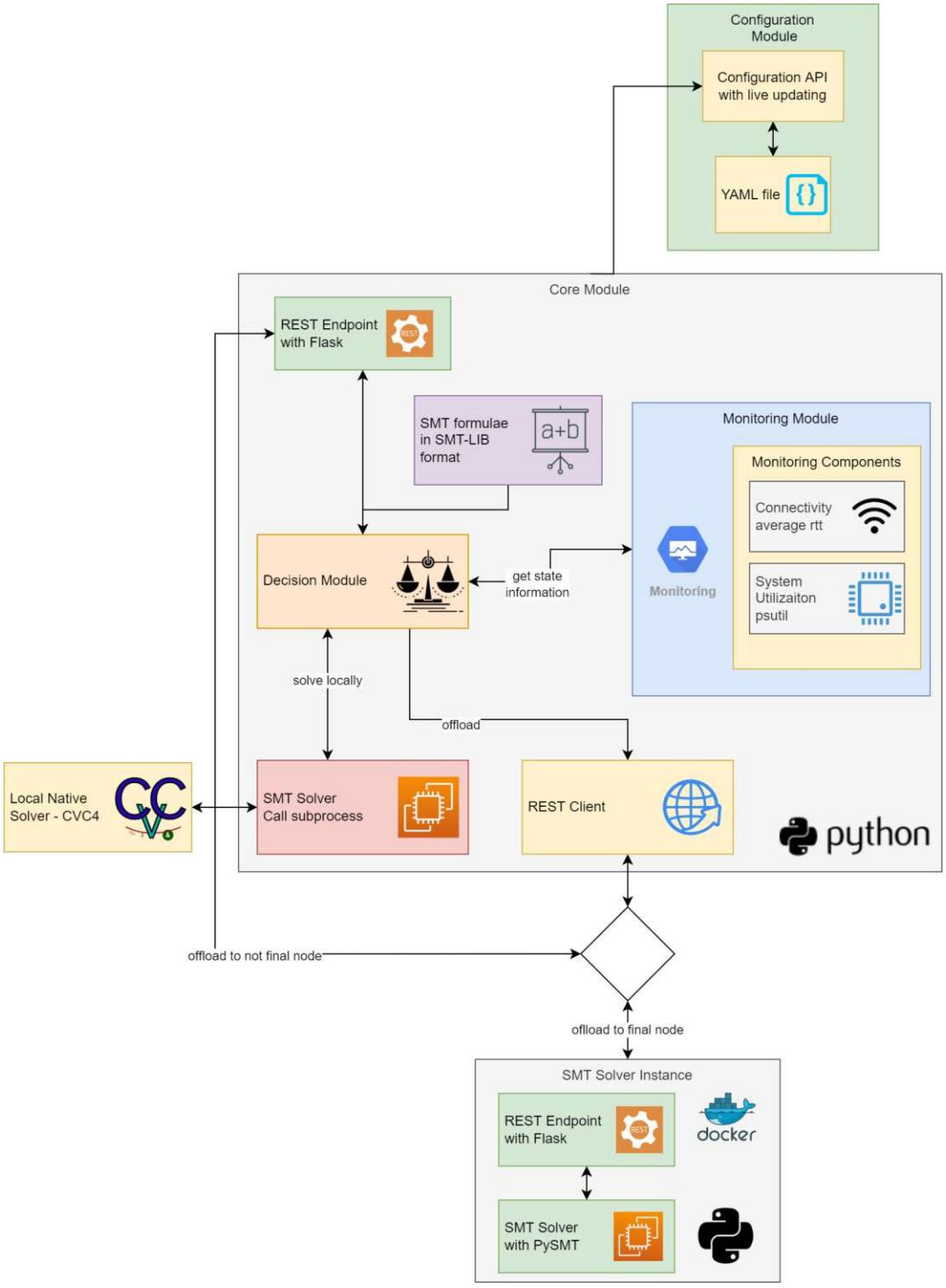


Figure 5.1: Architecture Design with Implementation Details

5.1.2 Development Framework

The entire code base is based on Python. We are using Python 3.5 and higher. On some devices, we have to rely on specific versions, as we need to use dependencies for which no pre-built binaries exist.

The following points should underline why we chose Python:

1. In Chapter 6 we are using a Lego Mindstorms EV3 and ev3dev (see Section 6.2.1). There are multiple libraries that abstract the low-level API in different programming languages like Python, Java, Go etc. However, the Python library is very well documented, has a big community and is easy to use.
2. According to a survey by statista [Vai21], Python is the third most used programming language among developers worldwide in 2021. Only JavaScript and HTML/CSS are used more frequently. There are many open-source frameworks, tools and libraries with a huge community in the background.
3. We use reinforcement learning, which is a subset of machine learning, in the decision module (see Section 4.2.3). Python provides libraries for this area like TensorFlow, Keras, PyTorch (for deep learning) or NumPy (see Section 5.4.2 and Section 5.1.2).
4. It is a high-level programming language and an interpreted language. It allows us to run the same code on multiple platforms without recompilation. Therefore, we can run our system on every Python capable device.

Python Packages

In order not to "reinvent the wheel", we benefit from multiple Python packages. In the following list we highlight the intended purposes and the advantages:

- **psutil**: The package is used to retrieve information about the running system. psutil is an acronym for python system and process utilities and can be used as a cross-platform Python library. It provides an API for several UNIX command-line tools like *ps*, *top* or *nextstat* [psu]. We can use this package to get information about the CPU usage, memory usage and traffic. This information can be used by the decision algorithm. Furthermore, the algorithm can easily be extended with more information about the system using psutil, e.g. disk usage or running processes.
- **pythonping**: The package helps us to obtain information about the connectivity of a system. Specifically, a ping to the instances to offload to is used to achieve this. There exist useful parameters, such as *count* to limit the number of ICMP packets to be sent to reduce the time required to obtain the connectivity information [pytb].
- **python-ev3dev2**: The package provides an interface for ev3dev (more in Section 6.2.1). It is used to determine the battery level of the robot which could be used

for decision making. Furthermore, it provides methods to get sensor data (e.g. from the ultrasonic sensor) or to send data to actuators (e.g. motors connected to wheels) [pyta].

- **requests**: The package is used to offload data and the library makes it very easy to send HTTP requests [Req]. We use the POST method of the HTTP protocol and send the data, in particular the *.smt2 files, which are the textual representation of SMT formulae according to the SMT-LIB specification (see Section 2.2.1) as form data.
- **NumPy**: The package benefits to store and access the Q-Table, which is used in one decision making approach [Numb]. NumPy provides multidimensional array objects and a number of very fast operations on arrays, such as selection etc. In our use case, we want to get the maximum of a row in a two-dimensional array as fast as possible. In addition, NumPy is also used by PyTorch, which is explained in more detail in Section 5.4.2.
- **watchdog**: The package is used to provide the ability to change the configuration at runtime. Watchdog provides a cross-platform API for monitoring file system events [wat].
- **PyYAML**: The package helps us to parse YAML files [pyy]. YAML is a recursive acronym for "YAML Ain't Markup Language" (originally "Yet Another Markup Language") and is a human-readable data serialisation language often used for configuration files, as we do [BKEI09].
- **Flask**: The package is used to provide REST-endpoints that can be called by the instances which want to offload SMT formulae. It is one of the most popular Python web application frameworks that makes it quick and easy to get started [fla].

5.1.3 Deployment Strategy

Some components in our architecture use a container-based cross-platform deployment. The specific technology we use is Docker, which is introduced below, with a focus on the areas of application in our implementation.

> Docker is an open-source engine that automates the deployment of applications into containers. Docker adds an application deployment engine on top of a virtualised container execution environment. It is designed to provide a lightweight and fast environment in which to run your code, as well as an efficient workflow to get that code from your laptop to your test environment and then into production [Tur14].

In the following, we describe the necessary components we need for implementation and deployment/evaluation [Tur14]:

- **Docker Image**: Docker containers are launched from Docker images. Images are the "build" part of the Docker lifecycle. It is a set of instructions to build a Docker container.
- **Docker Container**: These are the running instances of Docker images.
- **Registries**: A registry like Docker Hub[1] is a repository for Docker images. We can download (pull) an image from the registry and start a container from it.
- **Dockerfile**: This is the file, that describes how to create a Docker image.

The SMT solver instance is a containerised application with a simple REST endpoint that solves the provided SMT formulae with PySMT. These instances are deployed in the cloud and do not need to make any further decisions. The big advantage of using Docker here is that we can easily deploy multiple instances on different clouds. This means that instead of installing all the necessary dependencies like Python, PySMT and an underlying SMT solver etc. we only need to install Docker, pull the Docker image (from a registry) and run the image as a container.

We could also use a more sophisticated container orchestration mechanism like Kubernetes[2]. This would bring the advantage of high scalability if there would be many edge devices that want to offload their problems. In our evaluation (see Chapter 6) we have a limited number of devices and therefore Kubernetes would be an overkill solution. Why do we use Docker only on the cloud instances? A containerised application would result in overhead, and also not all edge devices are compatible with Docker. Therefore, our core module and the configuration module are natively installed on the edge devices. However, it would be easy to containerise these modules as well.

The following sections explain the various implementation details of the modules.

5.2 Communication Module

In this module, we primarily use the Python package requests (see Section 5.1.2).

5.2.1 Communication Mechanism

Communication between the components is done via HTTP. When a problem is offloaded, a POST request is created with the SMT formula in the body and the solution result in the answer. However, the implementation allows an easy switch to another communication mechanism like MQTT, as the communication module is loosely coupled. On top of HTTP, we use REST and the following points show why we choose this approach:

[1] https://hub.docker.com/
[2] https://kubernetes.io/

1. It allows us to meet the ETSI MEC specifications (see Section 2.1.4) if it is desirable to deploy our system on top of MEC hosts.
2. REST is more mature/widespread.
3. Our architecture follows a request-response style: We submit SMT formulae and receive the solution as a response. Therefore, REST is more suitable than publish-subscribe approaches like MQTT.

REST API

We only have one endpoint:

POST */formulae*

Request: The type of the request body is form-data. The key is *formula_file* and the corresponding value is the SMT formula as a file in SMT-LIB format (.smt2).

Response: The response content type is *text* and contains the result: *True* if the SMT formula is satisfiable or *False* if not. Of course, it is also possible to return more information, like a model, if the SMT formula is satisfiable. The HTTP status code is in both cases *200 OK*.

5.3 Configuration Module

In this module, we mainly use the Python packages pyyaml and watchdog (see Section 5.1.2). We use a .yaml file for the entire configuration. Table 5.1 shows the most important configuration options.

As already mentioned in Section 5.1.2 with watchdog we offer the possibility to change the configuration at runtime. For example, the update period in the monitoring module could be changed if we find problems with outdated state information.

5.4 Decision Modules

The implementation of the decision modules consists of multiple Python modules. These modules represent the core of the entire work. During the implementation, great care was taken to ensure that the modules can be exchanged as easily as possible. For example, if there is a new idea of decision making that is not already covered, then it is possible to create some new Python modules and reuse existing common code for decision making. The entry point into the whole decision module is the Python module *processing* and *processing_ev3*. Here we decide which specific approach to use for decision making. If we want to add a new approach, we could extend the *DecisionMode* enumeration and add a new *if* branch.

Key/Section	Description
smt.solver-location	defines the installation location of the solver, to call it natively
smt.decision-mode	could be set to q_learning - leads to Q-Learning as decision approach, or deep_q_network - leads to DQN as decision approach, or none - leads to no further decision making
instances	the endpoints for offloading are configured here (.cloud and .edge)
ev3.in-use	set to True if the software is executed on the robot - leads to other behaviour for other configuration calls: e.g. when getting solver instances and this option is set to True, the edge instances are returned, otherwise, the cloud instances
decision.common-hyper -parameters	common hyperparameters like learning rate, number of episodes, etc.
decision.deep-q-network	DQN specific options like batch-size or memory-size
monitoring.update-period	how often (in seconds) the current monitoring status is determined

Table 5.1: Important Configuration Options

Two other common Python modules used by the concrete decision approaches are the *problem_classifier* module and the *state* module. The first module acts as an "oracle" for the complexity of SMT formulae. For simplicity, in our implementation, the classification is hard-coded in the form of a Python directory, where the key is the filename of the SMT formula and the value is the classification as a numerical value. Furthermore, we have two different directories, one for the Raspberry Pi's and one for the robot, as the complexity varies depending on the device. The approach to classification involves the following steps:

1. Execute all problems on the robot and the Raspberry Pi's and record the execution times.
2. Define an upper bound of execution time for the robot and Raspberry Pi's (in our case, it was set to three seconds).
3. Calculate classification value (= problem complexity) with $round(runtime/upper\_bound\_execution\_time * 100)$
4. The higher the value, the harder the problem.

The *state* module represents the state needed for the concrete decision approaches. It calls the *monitoring* module to get information about the connectivity and calls

the *problem_complexity* module presented above to get information about the current problem.

There are three important implementation details to mention. First, the values of the state are normalised with the following simple formula: $normalised\_value = value/MAX\_VALUE$. This has the benefit that the values are then in the range of 0 to 1. The MAX_VALUEs are defined as constants for the different indicators (in the configuration file). Secondly, for the Q-Learning approach (described below, see Section 5.4.1) we need to transform the continuous values into discrete values. This is done with the help of a *Rating* enumeration containing the following values: *poor*, *fair*, *average*, *good*, *excellent*. And the last point is that we need some simulation in our evaluation (see Section 6.3.2). In general, this simulation can influence any value that represents the state. In our case, it is the average round trip time (rtt/latency), but it can easily be extended to simulate other state information. The simulation helps us to test our system under different settings.

5.4.1 Q-Learning

Q-Learning is the first specifically implemented approach for decision making with reinforcement learning. To store the Q-Table, NumPy is used, as already described in Section 5.1.2. The Q-Table has a size of 25 rows and three or five columns (depends on the number of offload options). The state information we use is on the one hand, the problem complexity and on the other hand the connectivity to the other edge devices. This leads to a state dimension of two. Each dimension could have five different values (values of the *Rating* enumeration, introduced above). We use a learning rate of 0.001.

In this section, we also want to describe the transformation from the continuous value range to the concrete value range in more detail. This is implemented with the function shown in Listing 5.1. If no value is given, we return *poor* as the default *Rating*. To be loosely coupled to the number of possible *Rating* values, we store the number of rating classes in a variable. After that, the value (a value between 0 and 1) is multiplied by the number of rating classes and the values after the decimal point are truncated. Due to the constant maximum values, it can happen that the maximum value is exceeded, which would result in a value greater than 1. If the value reaches or exceeds the maximum value, the converted value is reduced by 1. Afterwards, we get the rating by: number of rating classes - 1 - converted value.

Example: value = 0.5, number of rating classes = 5 (e.g. *poor=0*, *fair=1*, *average=2*, *good=3* and *excellent=4*), converted value would then be $0.5 * 5 = 2.5 \rightarrow match.trunc(2.5) = 2$, which would result in $Rating(5 - 1 - 2) \rightarrow Rating(2) \rightarrow Rating.average$. As 0.5 is in the middle of the range, 0 to 1 *average* is the correct rating.

```
def __get_rating(value):
    if value is None:
        return Rating.poor
    number_of_rating_classes = get_number_of_rating_classes()
    converted_value = math.trunc(value * number_of_rating_classes)
    if converted_value >= number_of_rating_classes:
        converted_value = number_of_rating_classes - 1
    return Rating(number_of_rating_classes - 1 - converted_value)
```

Listing 5.1: Transformation of continuous values to discrete values

5.4.2 Deep Q-Learning (DQN)

A slightly more sophisticated implementation is DQN, which is described in this section. To give more details, we also provide two Listings. The first one 5.2 shows how to create a neural network architecture and the second presents how to optimise the model 5.3.

PyTorch

With the increasing trend of deep learning, the number of deep learning frameworks has also increased. Choosing the most suitable library is not easy, as there are many popular ones: PyTorch[3], TensorFlow[4], Keras[5] or Apache MXNet[6].

When developing a deep learning framework, it is difficult to find a good trade-off between usability and speed. However, unlike other popular deep learning frameworks, PyTorch focused on both goals. PyTorch is an open-source machine learning framework developed mainly by developers from the Facebook AI Research Lab [PGM+19].

There are four main design principles [PGM+19]: As developers in the area of machine learning are familiar with Python, the first principle is to *Be Pythonic*. The second principle is *Put researchers first*, since it should be as simple and productive as possible. *Provide pragmatic performance* is the third principle, as the performance must be compelling, but not at the expense of simplicity and ease of use. The last principle is *Worse is better*, which means that a simple but perhaps incomplete solution is better than a comprehensive but complex and difficult to maintain solution.

Why we chose PyTorch:

- The balance between speed and ease of use [PGM+19].
- Our whole project is implemented in Python. Therefore, a library which completely focused on this programming language is a good choice.
- PyTorch is commonly used in research.

[3] https://pytorch.org/
[4] https://www.tensorflow.org/
[5] https://keras.io/
[6] https://mxnet.apache.org/versions/1.9.0/

- PyTorch is faster than Keras [Dee].

Installation

Section 6.2.1 shows the resource limitations of the Lego Mindstorms EV3. We focus on a lightweight solution, but the DQN requires more computing power and is therefore not efficiently feasible on the robot. The main problem here is the available RAM and the main processor, as we would need much more to perform deep learning in a meaningful way. To solve this problem, we have already proposed the Q-Learning approach using the Q-Table, which requires fewer resources (see Section 5.4.1). Beside the resource limitations, we have further difficulties in installing PyTorch on the Lego Mindstorsm EV3, as there are no pre-built packages and compilation has to be done on the robot, which would be very time-consuming, resulting in the need for cross-compilation (see Section 5.6.2). This and the resource-intensity of PyTorch are the reasons why we do not use DQN on the robot. Another fact is that there are only pre-built packages for Python3.7 or lower[7]. Since the default Python version of RaspbianOS is 3.9, we need to install Python3.7 on the Raspberry Pi's.

Model

In this section, we describe the creation of the neural network with PyTorch in detail. The structure and design of the network is explained in Section 4.2.3.

In Listing 5.2 we can see that the first layer (line 6) has a number of input features equal to the number of indicators. In line 15 we see that the number of output features is equal to the number of actions. Between the layers, we use a normalisation technique (see lines 7, 10, 13). The benefits of normalisation are presented in Section 2.5.7. The usage of the activation function Leaky ReLU is shown in the lines 8, 11 and 14 in Listing 5.2. In the PyTorch (torch) environment, the MEA loss function is called *l1_loss* and can be seen in lines 11 and 14 in the Listing 5.3. Listing 5.3 shows the usage of Adam as the optimisation function in line 12 - 17 (with comments included). The default settings are applied, i.e. a learning rate of 0.001 is used (the same learning rate as in the Q-Learning approach (see Section 5.4.1).

```
1  class DQN(nn.Module):
2      def __init__(self, number_of_indicators, number_of_actions):
3          super().__init__()
4
5          self.layers = nn.Sequential(
6              nn.Linear(in_features=number_of_indicators, out_features=24),
7              nn.BatchNorm1d(24),
8              nn.LeakyReLU(),
9              nn.Linear(in_features=24, out_features=24),
10             nn.BatchNorm1d(24),
11             nn.LeakyReLU(),
12             nn.Linear(in_features=24, out_features=24),
```

[7] at the time: 24. March 2022

```
            nn.BatchNorm1d(24),
            nn.LeakyReLU(),
            nn.Linear(24, out_features=number_of_actions)
        )

    def forward(self, t):
        return self.layers(t)
```

Listing 5.2: Neural Network Class

```
def optimize_model():
    if len(memory) < dqn_hyper_parameters['batch-size']:
        return
    experiences = memory.sample(dqn_hyper_parameters['batch-size'])
    states, actions, rewards, next_states = extract_tensors(experiences)

    current_q_values = QValues.get_current(policy_net, states, actions)
    target_q_values = rewards

    # l1_loss = MEA = mean absolute error
    loss = F.l1_loss(current_q_values, target_q_values.unsqueeze(1))
    # gradients of all weights and biases in policy_net are set to zero
    optimizer.zero_grad()
    loss.backward()
    # updates the weights and biases with the gradients
    # that were computed when called backward() on loss
    optimizer.step()
```

Listing 5.3: Function for optimising the model

5.5 Monitoring Module

For this module, the configuration option *monitoring.update-period* sets the period after which the monitoring information is determined. The perfect period requires some fine-tuning and depends on the particular device. For the training process, where the additional overhead does not matter, we fetch the current monitoring status for each SMT formula to always have up-to-date training data. In this module, we mainly use the Python packages psutil, pythonping and python-ev3dev2 (see Section 5.1.2).

5.6 SMT-Solver

The main objective of the work is to solve SMT problems. For this purpose, we use SMT solvers. In our implementation there are two approaches to call them. The first one is with the Python library PySMT, the second one is a native approach. The benefits of PySMT are that the solution is solver independent, and the underlying native solver can be chosen to best fit the environment. The disadvantages are the additional overhead and that some solvers are not easy to integrate. However, according to the documentation of PySMT any SMT-LIB compliant solver could be used [GM15].

5.6.1 PySMT

Here we describe the first approach of calling the SMT solvers. This one is used on the SMT solver instances.

PySMT is a Python library that provides an additional layer between the SMT formulae in the SMT-LIB format (see Section 2.2.1) and the solvers API [GM15]. The big advantage of this additional layer is that the formulae can be defined in a solver independent way, and you could select the best fitting solver for each environment [GM15]. For example, if the application using PySMT is executed on a more resource-rich system, Z3[8] could be used as the underlying SMT-solver. If the system is very constrained or only SAT theories need to be solved, for example, PicoSAT[9] could be used. The PySMT package is divided into three parts: *Formula API*, which includes modules to easily perform quantifier elimination, interpolation, etc. *Solver API* which calls the third part, *Converter*, to convert the general format to the solver specific format.

5.6.2 Native - CVC4

Here we describe the second approach of calling the SMT solvers. We use CVC4 as the native solver. This one is used on the robot and the Raspberry Pi's. The following is a definition and description of CVC4 [BCD+11]:

> CVC4 is an efficient open-source automatic theorem prover for satisfiability modulo theories (SMT) problems. It can be used to prove the validity (or, dually, the satisfiability) of first-order formulae in a large number of built-in logical theories and their combination. CVC4 is the fourth in the Cooperating Validity Checker family of tools (CVC, CVC Lite, CVC3) but does not directly incorporate code from any previous version. A joint project of Stanford University and The University of Iowa, CVC4 aims to support the features of CVC3 and Version 2 of the SMT-LIB standard while optimising the design of the core system architecture and decision procedures to take advantage of recent engineering and algorithmic advances. [BCD+11]

The decision on CVC4 was made due to the following aspects:

1. It is supported by PySMT. So we could use CVC4 natively as well as with PySMT.
2. It is more lightweight than Z3, but supports more theories than Yices[10], Boolector[11] or PicoSAT which are also supported by PySMT.

[8] https://github.com/Z3Prover/z3
[9] http://fmv.jku.at/picosat/
[10] https://yices.csl.sri.com/
[11] https://boolector.github.io/

3. Due to the rare CPU architecture of the device, a compilation by sources approach for installation must be followed. This makes proprietary solvers such as MathSAT[12] out of question.

4. CVC5[13] is very new and therefore there is much less material and guidance on (cross-)compilation via source code compared to CVC4.

Cross-Compilation

There are no pre-compiled CVC4 binaries for the CPU architecture (armv5tejl) of the Lego Mindstorms EV3 (see Section 6.2.1). Therefore, we need to create the binary by compiling the source code. This procedure is not feasible on the Lego Mindstorms EV3 due to its constrained resources. In order to solve this issue, we use the concept of cross-compilation. The normal case is that a compiler runs on one system and produces binaries that run on the same system. Such compilers are also called native compilers [Cro]. A cross-compiler is a compiler that runs on a different system than the produced binary is executed [Cro]. One can distinguish between the host system and the target system: the host system is the system on which the compiler runs and the executable is produced, the target system is the system on which the produced executable is executed [Cro].

The ev3dev (see Section 6.2.1) provides us with a Docker image (see Docker concepts in Section 5.1.3), with the most common developer tools for cross-compiling. From this Docker image, we can create a Docker container that allows us to run something very close to the ev3dev operating system on a more powerful computer. This means that we have the same libraries as the Lego Mindstorms EV3, but compile with the power of a desktop processor. A more detailed tutorial can be found here: [ev3a]. If additional libraries are necessary, it is important to use the target version as well. In our case, we need to add `:armel` when installing packages. CVC4 depends on several other modules. The most complicated module is ANTLR (ANother Tool for Language Recognition)[14], which is a "powerful parser generator for reading, processing, executing, or translating structured text or binary files". For this to work, we must first cross-compile the ANTLR module and use the cross-compiled module in the further cross-compiling procedure of CVC4.

5.7 Deployment

In this section, we briefly describe how we deploy the software on the different components. We can differ between the dockerised software and the modules run natively on. In general, we do not provide an automatic deployment with e.g. Ansible[15]. The first

[12] https://mathsat.fbk.eu/
[13] https://github.com/cvc5/cvc5
[14] https://www.antlr.org/
[15] https://www.ansible.com/

step is always to connect via SSH. We provide a step-by-step tutorial in the next two subsections:

Dockerised SMT Solver Instance

1. Install Docker
2. Open necessary ports to make service reachable remotely (Flask is running on port 5000 by default)
3. Pull the docker image containing the SMT solver instance software from a registry (e.g. DockerHub)
4. Create and run a docker container

Native Other Modules

1. Copy all necessary Python files via ssh
2. Adapt the configuration file: set *smt.solver-location*, *smt.decision-mode*, *ev3.in-use*, *instances*, ...
3. Start the component by running the main module

CHAPTER 6

Evaluation

In this chapter, we will benchmark the different components of our system and compare our approaches with some well-known baseline approaches. It starts with describing the objectives and then moves on to comprehensive technical specifications of our experimental setup, including the hardware on which we deploy our components. The next section looks at the details of the different approaches we use in our experiments and the metrics we measure. We also give an outlook with examples of other possible metrics that could be used, but are not evaluated in our experiments. In the subsequent section, we describe the test data we used. Finally, we present the results of our tests and discuss them in detail. We provide detailed explanations of the performance of the different approaches in the different settings and analyse the causes of the behaviour.

6.1 Objectives

The primary objective is to set up a testbed to show and compare multiple approaches with different settings. These approaches range from simple baseline approaches to more sophisticated solutions. Due to the technical complexity, extensive end-to-end instrumentation is needed to show all the components and modules. We use a robot, as we want to target autonomous mobile robots use cases, and the specific one as a representative of the low end of such techs, given its resource limitations. As resource-constrained edge computing hardware, we use two Raspberry Pi's which are becoming the de facto standard for evaluating edge computing systems for various IoT use cases (see, for example: [CWC+18], [CWC+18] or [HJS+20]). The more powerful server-grade virtual machines in the cloud complete the overall testbed. Also, this testbed is a mix that is a representative of the whole spectrum of hosts that can be found in the device-to-cloud compute continuum. The aim is to connect all the different components and provide a way to easily deploy and configure the whole architecture to compare the different approaches.

Furthermore, this setup should show that our implementation and architecture can be materialised in practice (especially across all supported hardware architectures).

The main goal is to get answers to our research question (RQ. 3) that focuses on a comparison of decision making strategies performances within different edge settings.

Another goal is to show that our solution can be compared in different dimensions with the usage of simulations.

6.2 Experiment Setup

As follows, the technical specifications of the used testbed are described.

6.2.1 Testbed Specifications

Technical Specification for the EV3 Brick

The Lego Mindstorms EV3 is the third generation (the EV3 stands for "evolution") of the Lego Mindstorms line [RL14]. The "brain" of the Lego Mindstorms EV3 robot is the brick, which serves as the control centre and power station. Table 6.1 lists the specifications of the brick. The brick can be connected with multiple sensors (touch, gyro, colour, ultrasonic) and motors (medium and large). Building instructions can be found on the official website.[1] The robot is the most resource-constrained device in our testbed and the SMT formulae originate on the robot when using the full testbed.

ev3dev

Instead of using the pre-installed operating system of the LEGO Mindstorms EV3 we use *ev3dev*. *ev3dev* is a Debian Linux-based operating system [ev3b]. The big benefit of using a Linux-based operating system is the availability of thousands of software packages and the Linux kernel provides the possibility to use USB and Bluetooth devices, like Wi-Fi dongles or keyboards. To connect the robot to the Internet, we use a Wi-Fi USB adapter, specifically the Edimax EW-7811Un, which supports the most common standards and is ideal for robots like this. A further improvement is that *ev3dev* runs from a microSD card and therefore does not touch the installed firmware. By removing the microSD - card and restarting the robot, it can easily be switched back to the original firmware [ev3b].

Technical Specification for the Raspberry Pi's

In contrast to the very resource-constrained robot (see Section 6.2.1) we also use two Raspberry Pi's as more powerful edge devices. In the further course of the work, they

[1] https://education.lego.com/en-us/product-resources/mindstorms-ev3/downloads/building-instructions

Release Date	September 2013
Operating System	Linux (ev3dev)
Display	178 x 128 pixel, Black & White, Monochrome LCD
Main Processor	TI Sitara AM1808 (ARM926EJ-S core) @300 MHz
Flash Memory	16 MB
RAM	64 MB
Memory	Micro SD card slot - supports SDHC, version 2.0, 32 GB
USB Host Port	USB 2.0 Communication to Host PC - up to 480 Mbit/sec
WiFi	Optional dongle via USB port
Bluetooth	Yes
Ports	4 input ports for sensors (1, 2, 3, 4) 4 output ports for motors (A, B, C, D) Mini-USB PC port, to connect the EV3 Brick to a computer
Speaker	for any sound effects
Power	6 AA batteries or rechargeable battery

Table 6.1: Specification of Lego Mindstorms EV3 Brick

Model	Raspberry Pi 4 Model B Rev 1.1
OS	Raspbian GNU/Linux 11 (bullseye)
Processor	Broadcom BCM2711 ARMv7 Processor rev 3 (v7l) 4x Cortex-A72 1.5 GHz
Memory	4GB LPDDR4
Connectivity	2.4 GHz and 5.0 GHz IEEE 802.11b/g/n/ac wireless LAN, Bluetooth 5.0, BLE Gigabit Ethernet 2 x USB 3.0 ports 2 x USB 2.0 ports
Input power	5V DC via USB-C connector (minimum 3A) 5V DC via GPIO header (minimum 3A) Power over Ethernet (PoE)-enabled (requires separate PoE HAT)

Table 6.2: Specification of Raspberry Pi

are also referred to as dedicated edge devices (DEDs). Raspberry Pi is a small single-board computer, of which several series and generations exist. Table 6.2 details the specifications for the Raspberry Pi's. Besides computing power, another difference between the Raspberry Pi's and the Lego Mindstorms EV3 is that the Raspberry Pi's have a continuous power supply. Accordingly, energy aspects, as described in Section 6.3.3, are not relevant on these devices.

OS	Ubuntu 20.04.3 LTS (GNU/Linux 5.4.0-88-generic x86_64)
Processor	Intel Xeon Processor (Skylake) 4 core, 2.2 GHz
Memory	8 GB RAM 48 GB hard disk
Location	Frankfurt, Germany

Table 6.3: Specification of external Cloud VM

OS	CentOS Linux 8 (Linux version 4.18.0-147.el8.x86_64)
Processor	Intel Xeon Processor (Cascadelake) 8 core, 2,1 GHz
Memory	8 GB RAM 22 GB hard disk
Location	TU Vienna

Table 6.4: Specification of TU Vienna Cloud VM 1

OS	CentOS Linux 8 (Core) (Linux version 4.18.0-193.6.3.el8_2.x86_64)
Processor	Intel Xeon Processor (Cascadelake) 4 core, 2,1 GHz
Memory	8 GB RAM 20 GB hard disk
Location	TU Vienna

Table 6.5: Specification of TU Vienna Cloud VM 2

Technical Specification for Cloud VM's

The final part of the testbed is formed by the various cloud instances. There are four different virtual machines, three of which are located in the TU Vienna network and one in Germany. These instances have the most computational power in our testbed. Table 6.3 sums up the detailed specifications of the external cloud VM instance. Table 6.4, Table 6.5 and Table 6.6 show the detailed specifications of the cloud VM instances in the TU Vienna environment.

6.2.2 Dataset

In this chapter, we present the process of selecting the test data. Throughout this work, we use the terms (SMT) formulae, (SMT) problem and test data as synonyms for the workload used in our evaluation. In the appendix A is a detailed listing of the problems used in the various sets and links to the resources. We use problems from the official SMT-LIB-benchmark repository[2]. The selection process consisted of the following steps:

[2] https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks

OS	CentOS Linux 8 (Core) (Linux version 4.18.0-147.el8.x86_64)
Processor	Intel Xeon Processor (Cascadelake) 2 core, 2,1 GHz
Memory	8 GB RAM 20 GB hard disk
Location	TU Vienna

Table 6.6: Specification of TU Vienna Cloud VM 3

Set Name	Number of Problems	Execution Time
Simple	14	[0 - 0.5] seconds
Medium	7	]0.5 - 1] seconds
Hard	9	]1 - 1.5] seconds
Mixed	10	[0 - 1.5] seconds

Table 6.7: Problem Sets

1. Downloading all the problems from the GitLab-repository.

2. Measure the execution times of all the problems with a timeout of 60 seconds. Problems that required more time are filtered out. The classification was done with a local CVC4 solver installed on an Intel(R) Core(TM) i5-6200 CPU @ 2.30GHz, with 8GM RAM Windows 10 machine. We then have a list of problems with a maximum execution time of 60 seconds.

3. Sort the problems by execution time and create three sets/buckets. The first set contains problems with an execution time of 0 - 0.5 (inclusive) seconds. The second set 0.5 (exclusive) - 1 (inclusive) seconds and the third set 1 (exclusive) - 1.5 (inclusive) seconds.

4. Filter out problems that can not be solved on the robot. Multiple problems are not solvable due to the constrained architecture of the robot. Some errors that occurred: *Unimplemented code encounteredConversion is dependent on SymFPU*, *Expected CVC4 to be compiled with SymFPU support* or *Unimplemented code encounteredFloating-point literals not yet implemented.*

5. This results in the problems sets, described in Table 6.7. The mixed set contains problems from the simple, medium and hard set.

6.3 Evaluation Configurations

The following sections describe all the approaches we are comparing, how we simulate latency and the metrics we will use to compare the configurations.

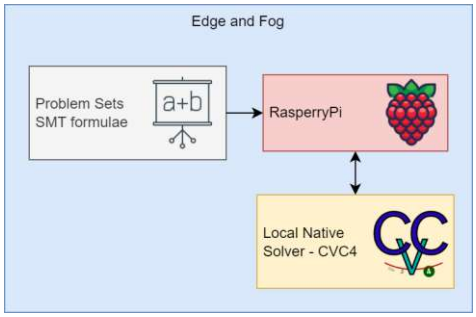


Figure 6.1: DED Only without Robot

6.3.1 Configurations

In this section, we specify the different approaches we deploy and evaluate on our testbed. A distinction can be made between two parts. In the first part, we do not use the robot and only compare the solutions by using the DEDs and the cloud instances. In the second part, we use the entire testbed, from the robot to the DEDs to the cloud instances. The reason we use these two parts is to better show the differences between the various approaches. It could be, for example, that we already have edge devices (robots) that are more resource-rich and comparable to our DEDs. This would be a use case for the first part we mentioned. Also, one approach (DQN) is not feasible on the robot and with this part, we could better show the benefits as the simulation of latency is between the DEDs and the cloud. In the second part, the latency is simulated between the robot and the DEDs. This represents a use case where the edge device is very limited and some more resource-intensive devices (like Raspberry Pi's) are installed in spatial proximity. All the approaches with some intelligent decision making represent use cases where offloading and solving locally are potential options. However, if one option is always the preferred solution, our experiments will show that the algorithms learn this behaviour.

We start with four approaches, where we do not use the robot.

DED Only without Robot

All the SMT formulae are solved on the DED itself. The DED does not make decisions and there is no offloading to other devices. The DED always forwards the formulae to the local native SMT solver (CVC4). This approach should represent a use case where the edge device is more powerful. A scenario, where this approach makes sense, is described in Section 6.3.1.

This results in one option to solve the formulae: on the DED (see Figure 6.1).

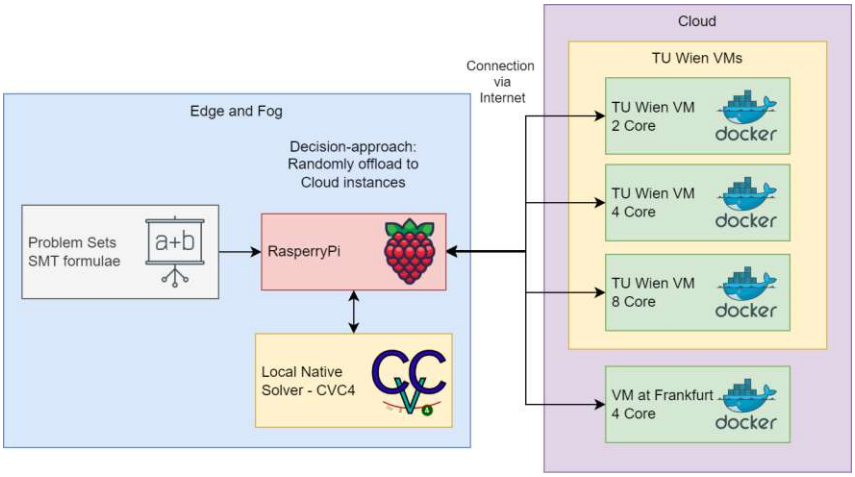


Figure 6.2: Cloud Only without Robot

Cloud Only without Robot

All the SMT formulae are offloaded to cloud instances. The selection is random and on the cloud instances, the requests are processed by a solver instance running as a docker container. The instances provide an endpoint to which the formulae can be sent and solve the formulae using PySMT (see Section 5.6.1). The underlying SMT solver is MathSAT5. This approach represents the traditional cloud offloading approach, or in other words, the computational offloading to the cloud (see Section 2.1.3).

This results in four options to solve the formulae: on the four cloud instances (see Figure 6.2).

Q-Learning on DED + Cloud without Robot

This is the first approach that involves more sophisticated decision making. We use Q-Learning on the DED to decide whether the problem should be offloaded to one of the cloud instances or solved on the DED. On the cloud instances, there is no further decision making.

This results in five options to solve the formulae: on the DED, on the four cloud instances (see Figure 6.3).

DQN on DED + Cloud without Robot

DQN is used on the DED to decide whether the problems should be offloaded to one of the cloud instances or solved on the DED. There is no further decision making on the cloud instances.

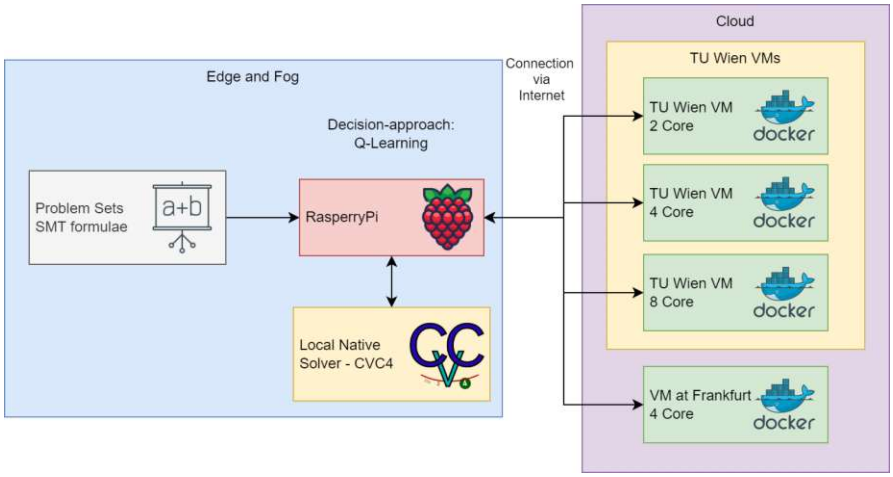


Figure 6.3: Q-Learning on DED + Cloud without Robot

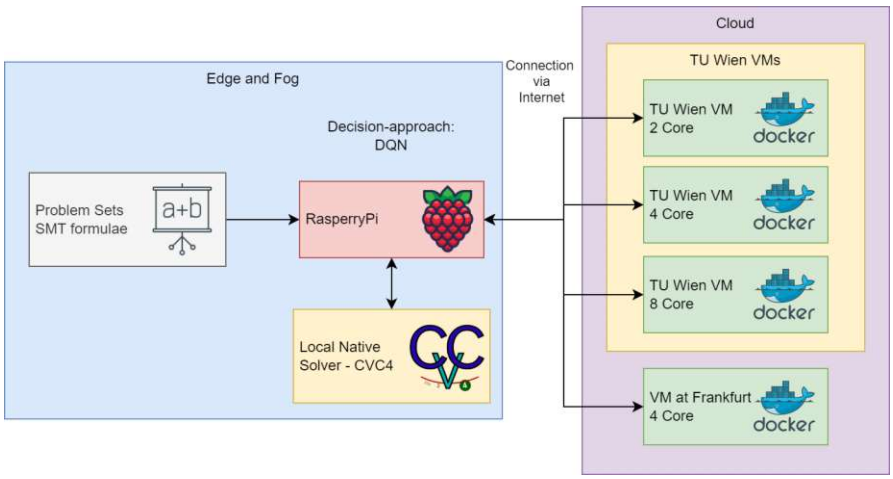


Figure 6.4: DQN on DED + Cloud without Robot

This results in five options to solve the formulae: on the DED, on the four cloud instances (see Figure 6.4).

In the following, we present approaches, where the problems originate at the robot.

Robot Only

All the SMT formulae are solved on the robot itself. The robot does not make decisions and there is no offloading to other devices. The robot always forwards the formulae to the local native SMT solver (CVC4). These scenarios serve to demonstrate the limitations of executing only on-device computations and motivate the need for offloading. It

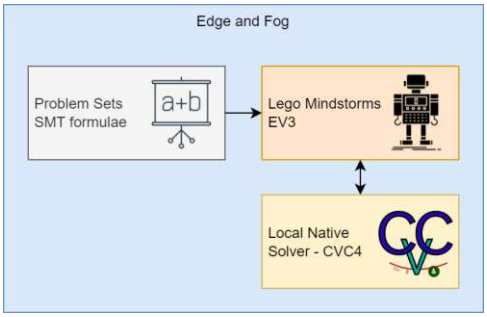


Figure 6.5: Robot Only

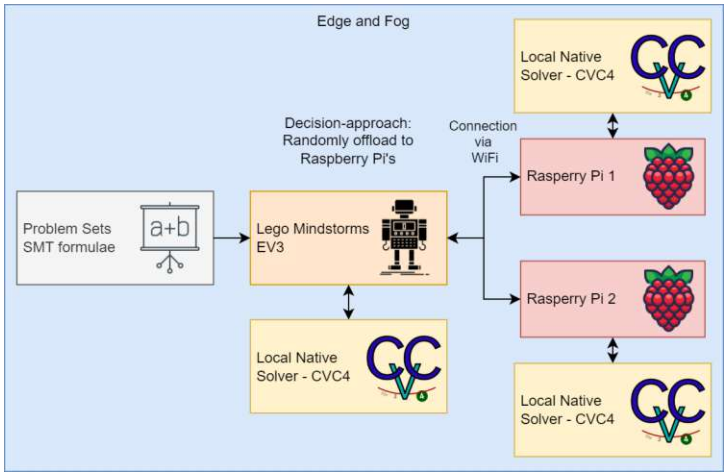


Figure 6.6: DEDs Only

should be noted that this setup can be considered characteristic of scenarios that feature disconnected and fully autonomous robot operation, where physical limitations such as the lack of network infrastructure (e.g. some disaster management situations [MTK16]) mandate that the execution of service logic takes place on the robots themselves, without edge or cloud assistance and control.

This results in one option to solve the formulae: on the robot (see Figure 6.5).

DEDs Only

All the SMT formulae are offloaded to the DEDs. The selection is random and the DEDs forward the formulae to the local native SMT solver (CVC4).

This results in two options to solve the formulae: on the two DEDs (see Figure 6.6).

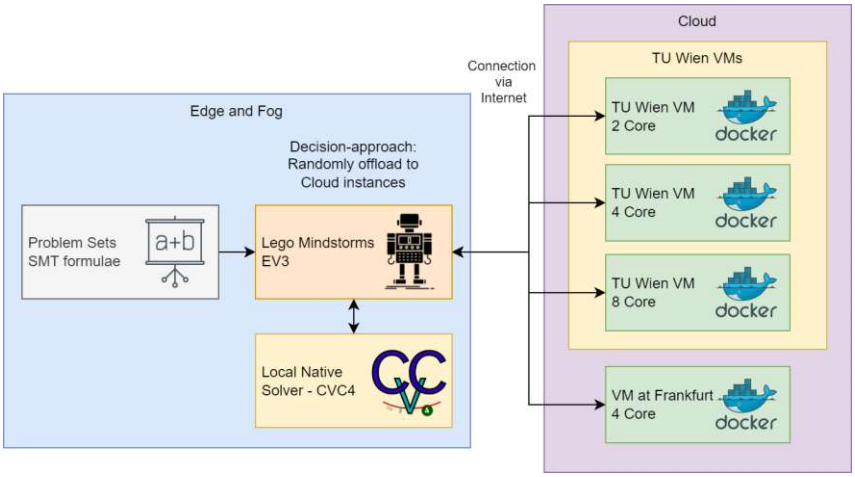


Figure 6.7: Cloud Only

Cloud Only

Basically identical to the approach described in Section 6.3.1. The difference to Section 6.3.1 is that the problems are originated at the robot and not on the DEDs.

This results in four options to solve the formulae: on the four cloud instances (see Figure 6.7).

Q-Learning on Robot + DEDs

We use Q-Learning on the robot to decide whether the problem should be offloaded to one of the DEDs or solved on the robot. On the DEDs there is no further decision making. They forward the formulae to the local native SMT solver (CVC4). In this approach, the additional latency represents the additional latency between the robot and the DEDs.

This results in three options to solve the formulae: on the robot, on the two DEDs (see Figure 6.8).

Q-Learning on Robot + Q-Learning on DEDs + Cloud

We use Q-Learning on the robot to decide whether the problem should be offloaded to one of the DEDs or solved on the robot. Q-Learning is also used on the DEDs to decide whether the problems should be offloaded to one of the cloud instances or solved on the DEDs. There is no further decision making on the cloud instances. As in the previous approach, here the additional latency represents the additional latency between the robot and the DEDs.

This results in seven options to solve the formulae: on the robot, on the two DEDs, on the four cloud instances (see Figure 6.9).

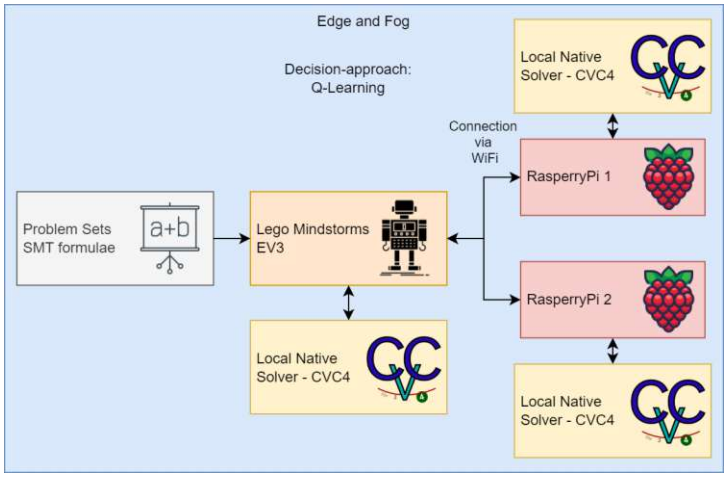


Figure 6.8: Q-Learning on Robot + DEDs

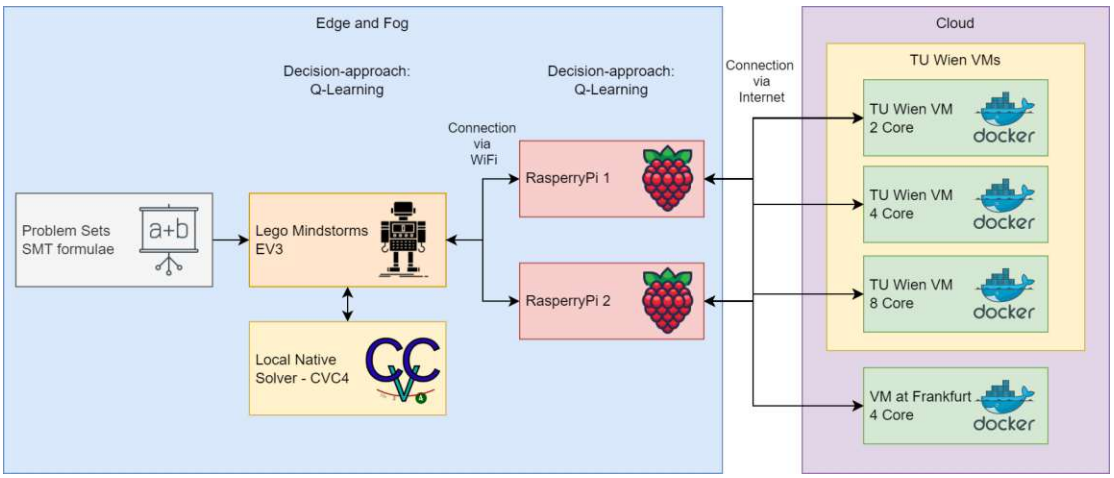


Figure 6.9: Q-Learning on Robot + Q-Learning on DEDs + Cloud

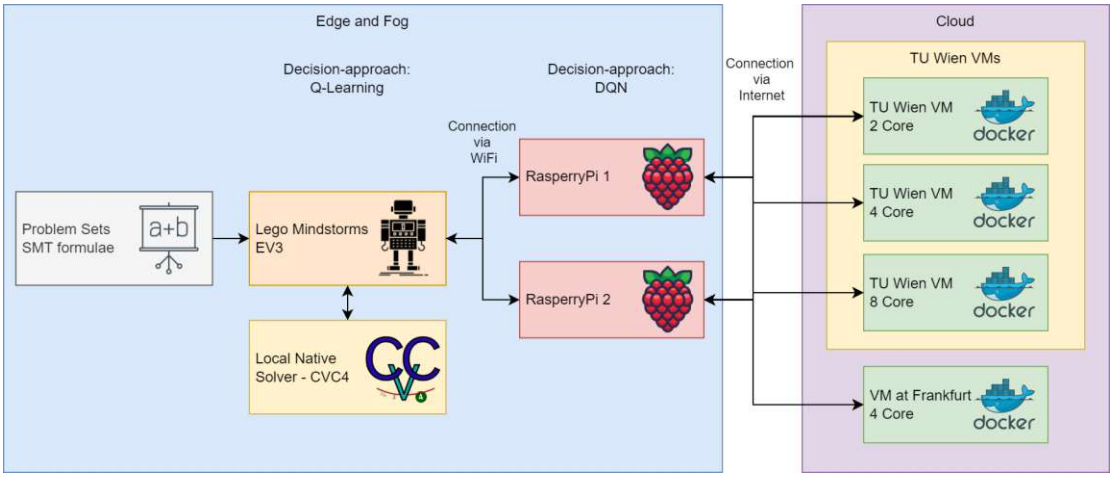


Figure 6.10: Q-Learning on Robot + DQN on DEDs + Cloud

Q-Learning on Robot + DQN on DEDs + Cloud

We use Q-Learning on the robot to decide whether the problem should be offloaded to one of the DEDs or solved on the robot. DQN is used on the DEDs to decide whether the problems should be offloaded to one of the cloud instances or solved on the DEDs. There is no further decision making on the cloud instances. In this approach, the additional latency represents the additional latency between the robot and the DEDs.

This results in seven options to solve the formulae: on the robot, on the two DEDs, on the four cloud instances (see Figure 6.10).

6.3.2 Latency Simulation

In order to compare our approaches with different latency settings, we simulate the latency at the application level. We use 5 different setups. In the first case, no artificial latency is added. We only have the natural latency between the components. In the other four setups, a latency of 100 to 400 ms is artificially added. This results in some additional time being needed when offloading problems. This additional time (additional waiting time for response) is calculated with the following simple formula:

$$\text{additional response time} = \text{additional latency} * 0.005$$

The constant of 0.005 is set based on the analysis of the changes in response time when simulating latency at the network level. To simplify the evaluation, we use this constant value and a simple calculation. However, a more sophisticated simulation could also be used, but this work does not consider this aspect. For example, the simulated latency could be different between the instances and the calculation of the additional response

time could follow a more advanced scheme/algorithm. The latency and additional waiting time is added between the DEDs and the cloud instances for the approaches without the robot and between the robot and the DEDs for the approaches with the robot. With this simulation, we are able to show how the approaches behave in different environments with additional latency between different components of our testbed.

6.3.3 Metrics

Below, we describe how we will compare the different approaches. We concentrate on the time metric. The energy-aware mode is explained in detail and is evaluated in a limited and simplified version, and other modes are only explained theoretically.

Time - Time-Aware Mode

One obvious metric is time. The question is how much time the different approaches need to solve a specific number of problems of varying complexity.

Energy - Energy-Aware Mode

Another potential metric is energy. The question is how the different approaches drain the battery. Energy consumption when solving SMT problems locally could be higher or lower than when offloading the problems. For example, for very simple problems, the energy consumption for offloading (transmission costs etc.) could be higher than the local execution. On the other hand, offloading can be more energy-saving if the problems are very complex and require long execution times. As a consequence, various indicators such as problem complexity, current battery level, connectivity etc. could further influence the energy consumption. In our testbed, the Lego Mindstorms EV3 only provides us with information on the open load voltage. The measurement of the battery level from the open load voltage is very unreliable. Therefore, an accurate comparison between the different approaches is not feasible. We have done some experiments with the battery level on the robot, but a good comparison is not possible. In different runs with similar problems and similar execution time, the battery level does not drop in a traceable and deterministic way. Because of this, we will only evaluate our solution with a simplified energy model in the energy-aware mode at the end of this chapter.

The model we use in our evaluation for energy consumption is specified as follows, where $e(p)$ is the function with the parameter p as the provided SMT problem.

$$e(p) = \begin{cases} \text{normalised size of } p, & \text{if } p \text{ is offloaded} \\ \text{normalised complexity of } p, & \text{if } p \text{ is executed locally} \end{cases}$$

We are aware that this model may have two limitations. First, we use our sophisticated reinforcement learning approach for a simple minimum condition. However, that shows that our proposed system can also be used in environments with very simple reward

functions. Second, the model is an abstract model that aims to approximate actual energy consumption costs in practice, and that (i) loss of accuracy is unavoidable, (ii) adaptions of the model may be required to account for the particularities of different connectivity technologies.

Traffic - Time/Energy and Traffic-Aware Mode

Our implementation makes it possible to combine multiple metrics. Offloading is not free of charge. There are costs for the bandwidth and also costs for hosting and calling the services. The cloud service, for example, could operate on a "charge per request" model. More specifically, the execution of SMT requests could be offered as a function-as-a-service (FaaS) using serverless computing [BCC+17]. Accordingly, a traffic-aware mode makes sense where the goal is to reduce the costs by reducing the volume of traffic. However, only a traffic-aware mode is not really useful, as the solution would be to never offload and thus reduce the traffic to zero. But it would make sense to combine the traffic-aware mode with the time-aware ore energy-aware mode presented above. So on the one hand we want to minimise the energy consumption or the time needed and on the other hand we want to reduce the generated traffic. Our algorithm would have to find some trade-offs, as the objectives conflict with each other, and it is not possible to optimise for one without compromising the other.

6.4 Experiment Results

In this section, we present the results of the different approaches. We want to stress a few points in advance that are relevant in the results. The execution time of the SMT formulae is not deterministic. For multiple runs, we get small differences. Therefore, we take the average of several problems over several runs. As our testbed is installed in a real environment, the latencies between the different runs may vary slightly.

6.4.1 Analysis of Comparison without Robot

In this section we focus on the results without the robot: Figure 6.11 compares all the different approaches with different latencies and Figure 6.12 illustrates the average results across all latencies.

In the following, we point out some details. The start and the end of problem-solving is on a DED, i.e. the results are based on the start and the end timestamps taken on the DED. The simulated latency is added between the DED and all the cloud instances. Figure 6.11 illustrates that the cloud only approach always increases linearly with additional latency. The results point out that the performance of the DED only approach is always the same and is independent of the additional latency.

Simple Problem Set: The focus is on the graph at the top left in Figure 6.11. Offloading makes no sense, so the best solution is always to solve on the DED. From the graph we can see that results of Q-Learning, DQN and DED only are almost the same.

It can be seen in Figure 6.12 that the DQN approach performs slightly worse on average than the DED Only and Q-Learning approaches. The reason for this is the overhead due to the use of the neural network, which is not compensated in this set.

Medium Problem Set: The focus is on the graph at the top right in Figure 6.11. If the latency is very low (0ms additional latency), offloading make sense. In the other cases, it is the better choice to solve the problems on the DED. The DQN approach can better classify the complexity of the problems, and therefore the results for 0ms additional latency are better than the Q-Learning approach. Also on average across all medium problems, the DQN approach is the best solution.

Hard Problem Set: The focus is on the graph at the bottom left in Figure 6.11. For additional latency up to 200ms, offloading is the better choice. For more than 200ms, solving on the DED is the better choice. The cause for that is that the problems of the hard problem set are more complex and need more computational power, which is provided better by the cloud instances compared to the Raspberry Pi. However, if the latency and thus the additional response time is higher, solving on the DED is the better choice. With 100ms additional latency, the DQN approach is worse than the cloud only and Q-Learning approaches. The reason for this could be, on the one hand, that the latencies in these runs differed a bit (see general remarks above). On the other hand, due to the non-deterministic execution time of the problems, the accuracy of the model used might not be good enough to choose the best options in this scenario. But for other settings (e.g. 0ms) the DQN approach is better than the Q-Learning approach, but moderately worse than the cloud only approach due to the additional overhead.

Mixed Problem Set: The focus is on the graph at the bottom right in Figure 6.11. We can see that the results are quite similar to the medium problem set, which is reasonable as it contains problems of the simple, the medium and the hard problem sets.

In general (Figure 6.12), we see that our approaches (including some intelligence like Q-Learning or DQN) always choose the optimal decision between offloading and solving on the DED. We also see that the DQN approach is the best solution.

6.4.2 Analysis of Comparison with Robot

In this section we focus on the results with the robot: Figure 6.13 compares all the different approaches with different latencies and Figure 6.14 illustrates the average results across all latencies.

In the following, we point out some details. The start and the end of problem-solving is on the robot, i.e. the results are based on the start and the end timestamps taken on the robot. The simulated latency is added between the robot and all DEDs. Figure 6.13 show that the cloud only and DED only approach always increases linearly with additional latency. The results point out that the performance of the robot only approach is always the same and is independent of the additional latency. An important piece of information is that we omit the results of the robot only approach for the sets, as the average time

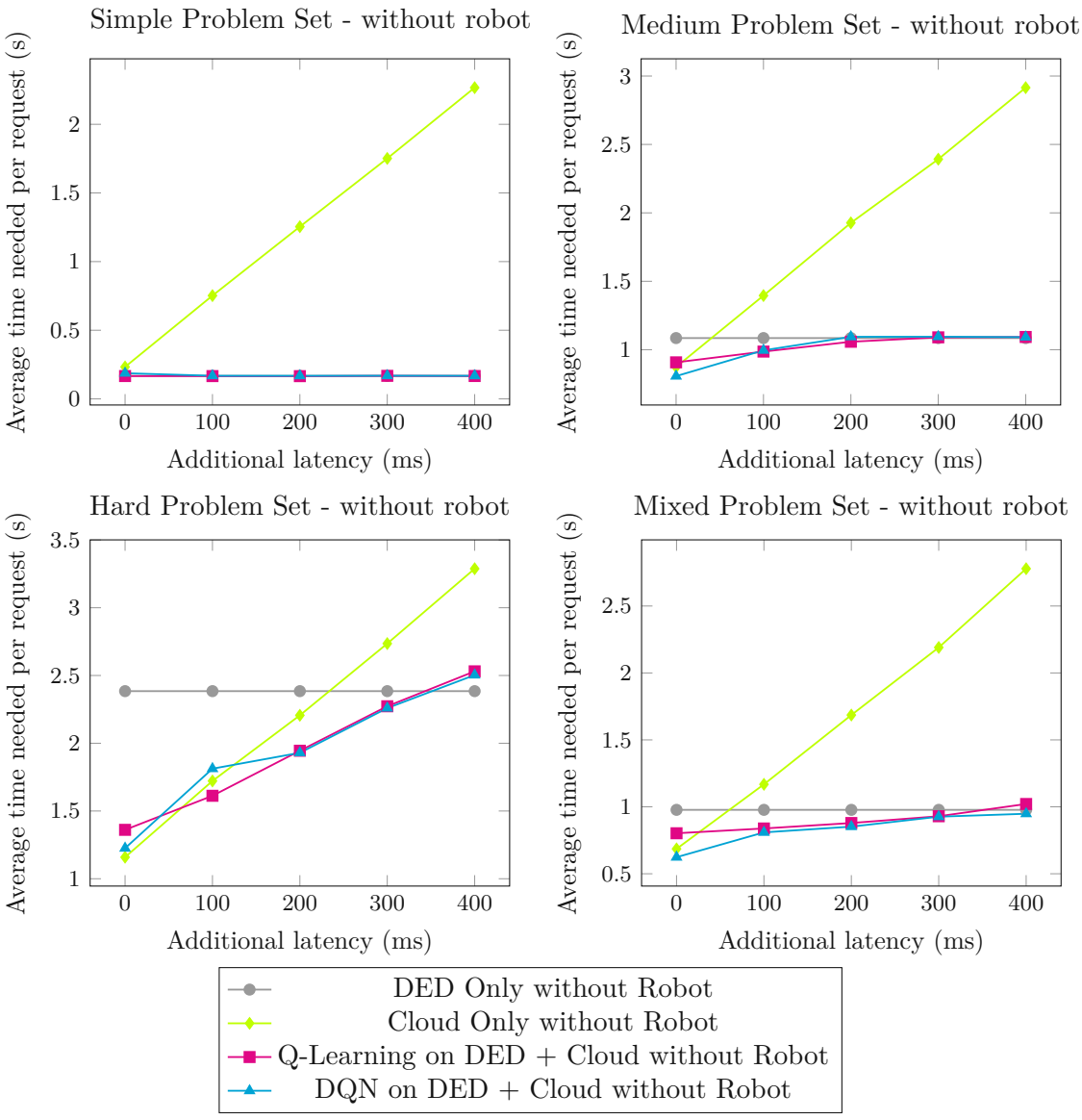


Figure 6.11: Comparison without Robot

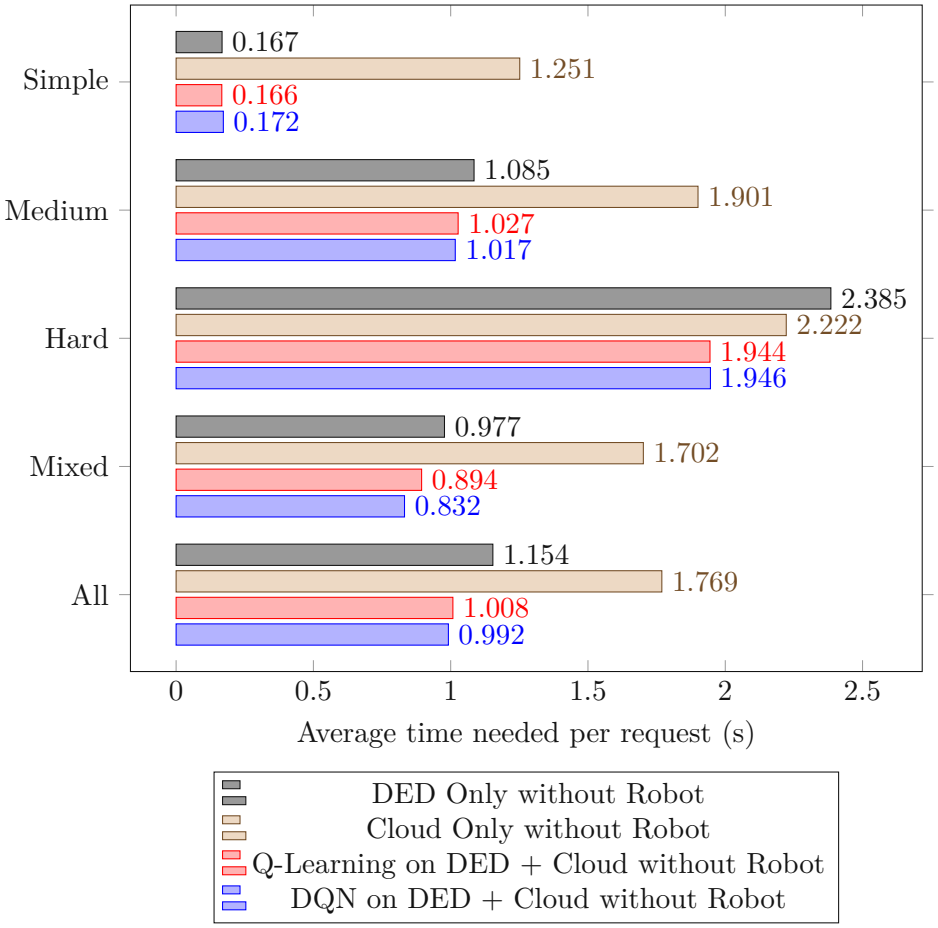


Figure 6.12: Comparison of all latencies (on average) - without robot

needed per request is much higher, which would negatively affect the visualisation of the diagram. In particular, the robot only approach achieves 5.058, 46.254, 104.176 and 46.045 seconds in average per request for the simple, medium, hard and mixed problem sets, respectively. Mainly, there is not much difference between DQN and Q-Learning approach on the DEDs, as the latency only differs between the robot and the DEDs and not between the DEDs and the cloud. The differences here are illustrated and described above.

Simple Problem (refcount38.smt2): The focus is on the graph at the top centre in Figure 6.13. This problem has very low complexity and only in this diagram we add the execution time on the robot. This figure illustrates an example where it is better to solve the problems on the robot than to offload them. The performance of the DED only and the cloud only approach is very similar, as the problem is very simple.

Simple Problem Set: The focus is on the graph at the top left in Figure 6.13. The simple set contains some very simple problems (like refcount38.smt2, described above).

From about 100ms additional latency, it is, therefore, better for some problems to solve them on the robot. All of our approaches learn this behaviour and make the optimised decision. In this case, cloud only approach is slightly worse than DEDs only for the following reasons: 1. the problems are very simple and can be solved very fast on the DEDs, 2. the natural latency between the robot and the cloud is worse than the natural latency between the robot and the DEDs. In general, Q-Learning and DQN on the DEDs lead to additional overhead that can not be compensated for problems from the simple problem set. This causes a bit worse results when there is an additional decision making approach on the DEDs.

Medium Problem Set: The focus is on the graph at the top right in Figure 6.13. Offloading from the robot is always the best option. The problems are more complex (compared to the simple problem set), therefore cloud only is better than DED only. In contrast to the simple problem set the additional decision making on the DEDs pays off and therefore these approaches perform better than only Q-Learning on the robot.

Hard Problem Set: The focus is on the graph at the bottom left in Figure 6.13. Offloading from the robot is always the best option. The problems are more complex (compared to the simple problem set) therefore cloud only is better than DED only. The problems are in most cases offloaded to the cloud, so cloud only and decision making on the DEDs are the best options and very similar.

Mixed Problem Set: The focus is on the graph at the bottom right in Figure 6.13. In most cases, offloading from the robot is the best option (except for some very simple problems like refcount38.smt2). For some problems the best option is to solve on the robot, for some problems to solve on the DEDs and for some on the cloud instances. Therefore, Q-Learning on the robot + decision making on the DEDs performs best.

In general, all the diagrams show that our approaches always make the optimal decision between offloading and solving on the DED or on the robot. We also see that DQN and Q-Learning on the DEDs are principally very similar, as the latency is stable, and we cannot take advantage of the decision making with DQN as shown above.

6.4.3 Analysis of Energy-Aware Mode Results

Since, the energy-aware mode only makes sense on the robot, as the other devices in our experiment setup have a continuous power supply, we will only compare the following three configurations: Robot Only, DEDs Only, Q-Learning on Robot + DEDs. For the reason of simplification, we do not consider energy costs on the devices with a continuous power supply, although our model would easily allow that. We also omit the configuration Cloud Only, as this is identical to DEDs Only from a device energy consumption perspective. Figure 6.15 shows the comparison of the three configurations mentioned above. The energy needed is defined with the model we introduced in 6.3.3. The bars show the sum of all problems in the simple, medium, hard and mixed problem sets. When using our system, the decision making module opts for a minimum of needed energy from the local execution and offloading, resulting in lower overall energy consumption. We also see

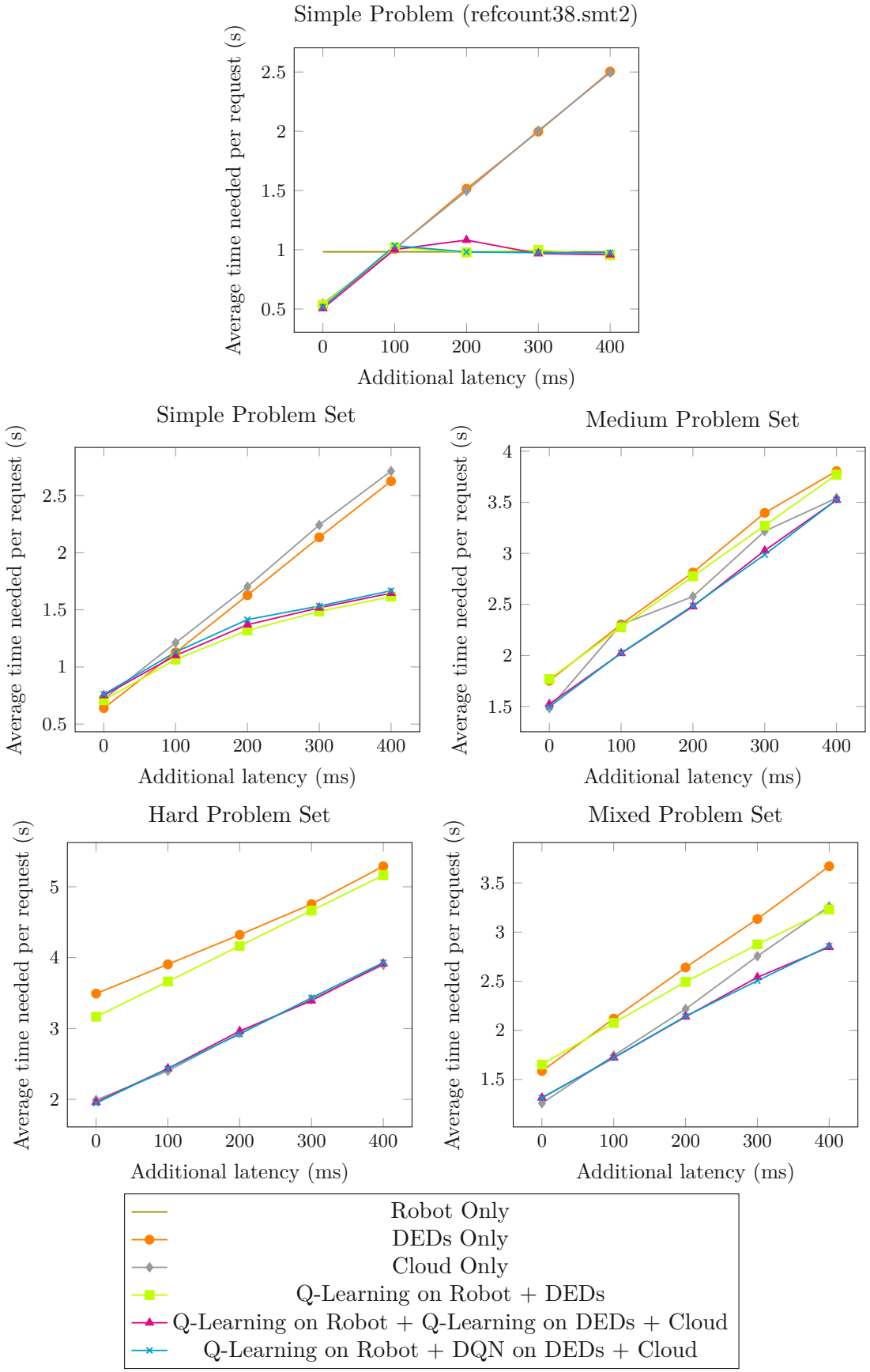


Figure 6.13: Comparison with robot

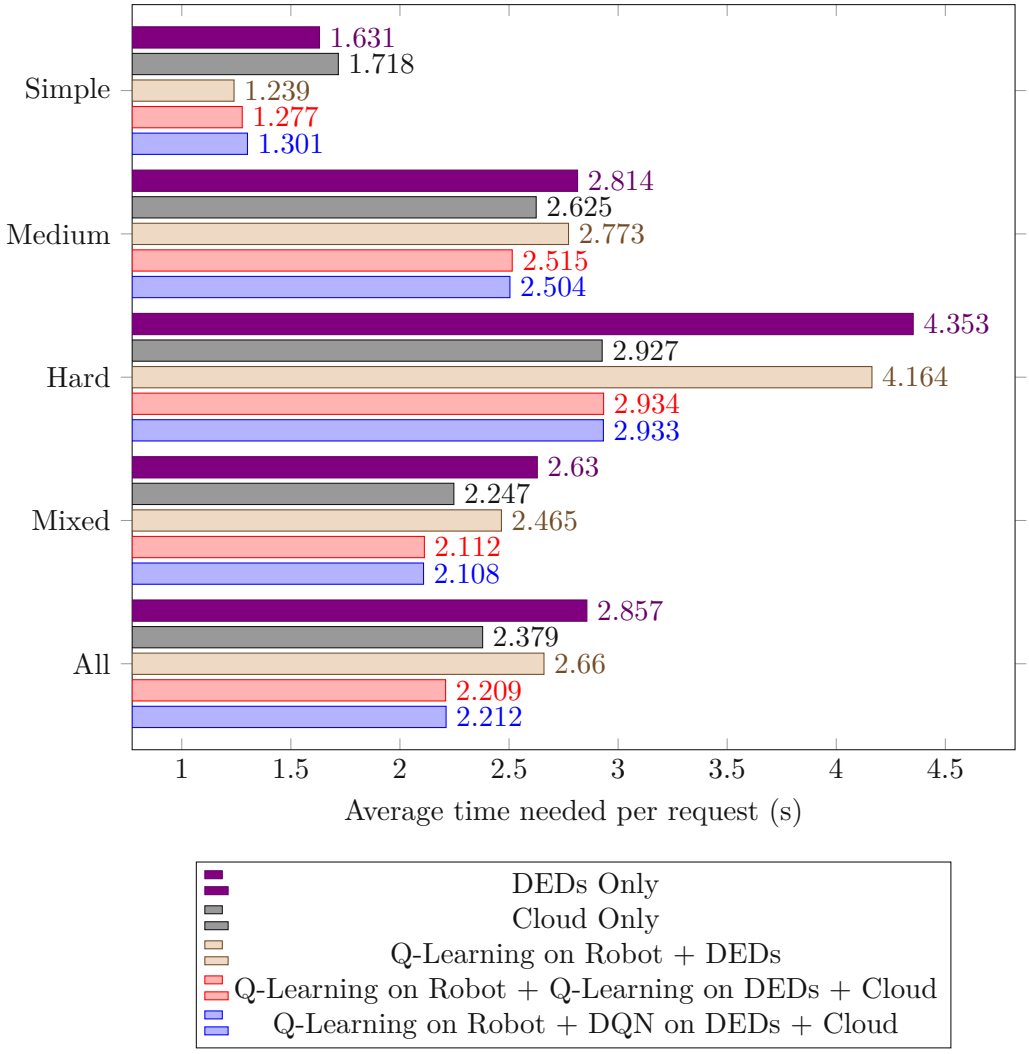


Figure 6.14: Comparison of all latencies (on average) - with robot

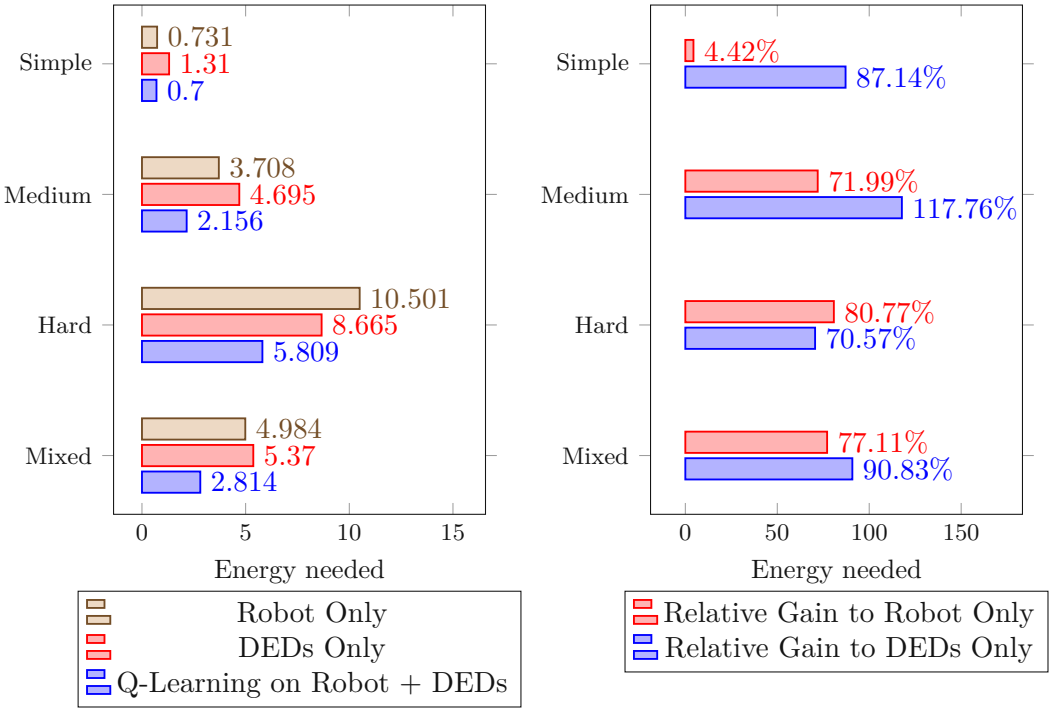


Figure 6.15: Comparison with energy-aware mode

that for the simple and the medium problem set, it is better to solve locally, and for hard problems, it is better to offload. In most cases, the size of the problems and the complexity correlates, but there are also exceptions where the description (the file size) is large, but the problem is not that complex and vice versa. On the left, the increase in energy cost as a percentage of the cost of the Q-Learning mechanism are shown. We see that the gains are significant, and e.g. for the medium set we can reduce the needed energy by more than half.

Figure 6.16 shows a comparison of the energy needed for all problems of all sets with the different approaches. The total is the sum of the energy for all problems from all sets with the three different approaches, i.e. it is the energy needed for three times the amount of problems. The figure presents the ratios of the energy consumed by each approach to the total energy consumed across all experiments.

6.5 Use Case: Path Planning for Fog-Supported Robots

In this section, we present a real use case in the field of fog robotics and show a 360-degree end-to-end view of our system, where a problem is translated into an SMT formula, processed by our system and the result is utilized by a robot. We create a planning problem based on a Hamiltonian path, which is a well-known problem in the mathematical field of graph theory.

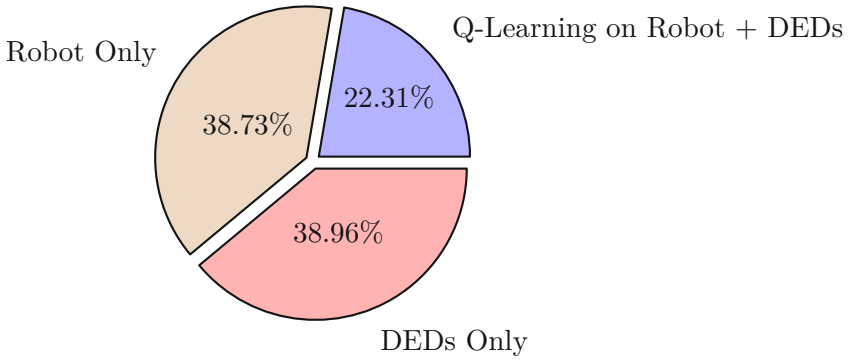


Figure 6.16: Energy needed per approach

> A Hamiltonian path, also called a Hamilton path, is a graph path between two vertices of a graph that visits each vertex exactly once. If a Hamiltonian path exists whose endpoints are adjacent, then the resulting graph cycle is called a Hamiltonian cycle (or Hamiltonian cycle) [Wei03].

The Hamiltonian path problem is NP-complete, and it follows that the problem can be reduced in polynomial time to a satisfiability problem which can be solved by an SMT solver efficiently.

The search for a Hamiltonian cycle may appear in multiple real applications; consider a contemporary warehouse where a robot operates. A simple application scenario is a warehouse with a robot. The robot receives a continuous stream of commands, each containing a set of items (e.g., construction materials) to pick up and carry them to its base. Each material/item is placed at a fixed location. These materials represent the vertices, and the ways between them represent the edges. The robot should not pick up materials twice, and therefore every vertex should be visited only once, leading to a Hamiltonian path problem. This scenario could be extended with more constraints like different edges having different costs and so on, but for the sake of simplicity we present the simplest form. The robot is required to end its route with the starting point again, and therefore we formulate a Hamiltonian cycle problem dynamically. Each command is translated to an SMT formula, whose satisfiability checking will result in a solution to the specific Hamiltonian cycle problem instance. In turn, this solution is translated to a sequence of low-level instructions for the robot, so that the latter implements the derived route.The stream of commands therefore represents our SMT workload, which is possible to execute on the robot, on top of dedicated edge devices, or in the cloud, according to a decision made dynamically – on a per-command basis – by our SMT-as-a-Service framework.

6.5.1 Hamiltonian Cycle in SMT-LIB Format

Since our system requires the input in SMT-LIB format, we need to develop a mechanism to translate a Hamiltonian cycle problem into an SMT formula in SMT-LIB format. It is essential that the procedure is independent of the environment and the robot is able to find Hamiltonian cycles for different graphs.

Listing 6.1 shows how to populate a file in SMT-LIB format that encodes a Hamiltonian cycle problem based on a graph. The input graph has the format of a Python dictionary, where each key is a vertex in the graph and the value is a list of adjacent vertices, e.g. `0:[1,2], 1:[2], 2:[1,0]`, which means that there are three vertices 0, 1, 2 and e.g. 0 has 1 and 2 as neighbours. In line 2 we set the option to produce models when solving the formulae, as we use the model to move the robot. Line 3 shows the used logic, namely QF_LIA which stands for quantifier-free linear integer arithmetic.

The logic involved is as follows. We generate a list of *Int* constants and assert that the first vertex (v0) is 0. Then we pick a vertex i, and attempt to number all vertices reachable from i in such a way that they have a number that is one higher (modulo the number of vertices) than the one assigned to the vertex i (we use an OR constraint for this). A resulting constraint could look like the example below, where 8 is the number of vertices and v1 and v3 are adjacent vertices of v0.

```
((assert (or(= v1 (mod (+ v0 1) 8))(= v3 (mod (+ v0 1) 8))))
```

The last two lines show that we are checking for satisfiability, and if this is the case, we want to obtain a model that satisfies all assertions. Two complete examples based on the use case below can be found in the Appendix B.

```
1  def fill_temporary_file(graph, temp_file):
2      temp_file.writelines("(set-option :produce-models true)\n")
3      temp_file.writelines("(set-logic QF_LIA)\n")
4      number_of_vertices = len(graph)
5      for i in range(number_of_vertices):
6          declaration = "(declare-const v" + str(i) + " Int)\n"
7          temp_file.writelines(declaration)
8      temp_file.write("(assert (= v0 0))\n")
9      for i in range(number_of_vertices):
10         or_conditions = ""
11         for j in graph.get(i):
12             or_conditions = or_conditions + "(= v" + str(j) +
13             " (mod (+ v" + str(i) + " 1) " + str(
14                 number_of_vertices) + "))"
15         if or_conditions != "":
16             or_assert = "(assert (or" + or_conditions + "))\n"
17             temp_file.writelines(or_assert)
18     temp_file.writelines("(check-sat)\n")
19     temp_file.writelines("(get-model)\n")
```

Listing 6.1: Fill SMT-LIB file for Hamiltonian Cycle Problem

6.5.2 Moving on a Grid

For the sake of simplicity, we let the robot move only in four directions: (i) right, (ii) left, (iii) up and (iv) down. The robot knows a grid structure with coordinates and is able to move to specific coordinates. In Listing 6.2 we show how the robot moves to a destination represented with coordinates. The solution (the model) from the SMT solver is parsed to get a sequence of vertex IDs (the path) which is translated to actual grid elements. Then, for each stop (defined by a starting point and a destination) in the path, the function `move_to` presented in Listing 6.2 is called so that the robot moves from the starting node to the destination. The variable `coordinates` is a dictionary where the key is the number of the vertex and the value is a tuple representing the coordinates of the vertex (the grid elements). The invoked functions call the robot API and turn on the motor for a specific number of seconds to move forward/backward or to change the direction of the robot (see details in Listing 6.3). The robot only knows two "view directions" and for moving left and up the speed is negative, leading the robot to move backwards.

```
1  def move_to(starting_point, destination, view_direction):
2      destination_coordinates = coordinates.get(destination)
3      starting_point_coordinates = coordinates.get(starting_point)
4      move_action = (destination_coordinates[0]
5                     - starting_point_coordinates[0],
6                     destination_coordinates[1]
7                     - starting_point_coordinates[1])
8      if move_action[0] != 0:
9          if view_direction == 'right':
10             change_direction_to_down()
11         move(move_action[0])
12         view_direction = 'down'
13     if move_action[1] != 0:
14         if view_direction == 'down':
15             change_direction_to_right()
16         move(move_action[1])
17         view_direction = 'right'
18     return view_direction
```

Listing 6.2: The move to logic of the robot

In our use case, we assume a fixed grid and choose a constant speed. Based on the speed and the distances between the stops, we determine how long the motor must be active (see Listing 6.3). In a real-world deployment scenario, it is also realistic to start from a fixed terrain and create an initial configuration on which to base the robot's movements. Some robots (also Lego Mindstorms EV3) offer a more high-level API, that only needs the distance and the speed or the rotation angle as parameters, which has the advantage that the duration of how long the motor has to be active does not have to be specified. However, in this use case, we use the low-level interfaces. For simplicity, however, we focus on simple grid structures, as very general and complex floor plans would go beyond the scope.

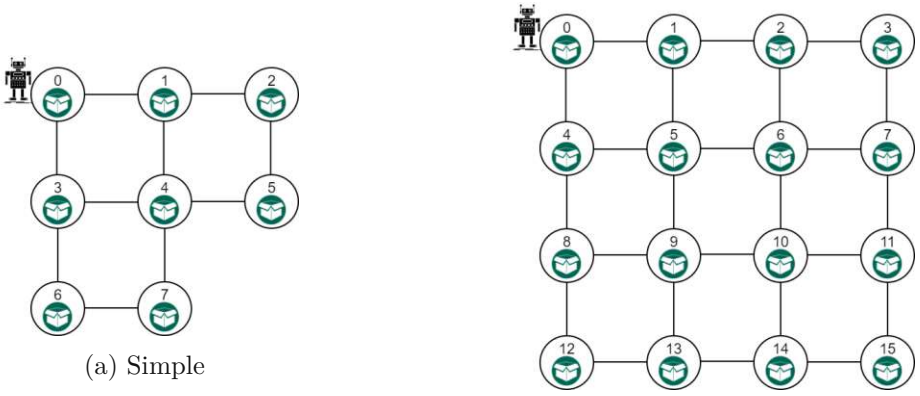
```
TURN_SPEED = 10
MOVE_SPEED = 40

def move(value):
    forward = 1
    if value < 0:
        forward = -1
    motor.on_for_seconds(left_speed=MOVE_SPEED * forward,
        right_speed=MOVE_SPEED * forward, seconds=abs(value))

def change_direction_to_right():
    motor.on_for_seconds(left_speed=TURN_SPEED * -1,
        right_speed=TURN_SPEED * -1, seconds=1.5)
    motor.on_for_seconds(left_speed=0,
        right_speed=TURN_SPEED, seconds=3)
    motor.on_for_seconds(left_speed=TURN_SPEED * -1,
        right_speed=0, seconds=0.35)

def change_direction_to_down():
    motor.on_for_seconds(left_speed=TURN_SPEED * -1,
        right_speed=TURN_SPEED * -1, seconds=1.5)
    motor.on_for_seconds(left_speed=TURN_SPEED,
        right_speed=0, seconds=3)
    motor.on_for_seconds(left_speed=0,
        right_speed=TURN_SPEED * -1, seconds=0.35)
```

Listing 6.3: Move logic of robot

6.5.3 Experiment

In this section, we feature two experiment setups representing the general use case. In Section 6.5.1 we presented how to translate the path problem into an SMT formula and in Section 6.5.2 we showed how to make the robot move. Figure 6.17 shows an abstract version of our use case. In Figure 6.17a a simple version with 8 vertices and in Figure 6.17b a more complex version with 16 vertices is illustrated. The robot starts at vertex 0 and the underlying idea is that each vertex has some materials placed that need to be collected.

Figure 6.18 shows the real Lego Mindstorms EV3 in a miniaturised warehouse terrain grid. Again, on the left side, Figure 6.18a illustrates the simple version and Figure 6.18b the complex one. Here we need to distinguish a bit between the structures. To reuse the tape on the floor, in the simple version the squares represent the vertices (e.g. the robot is placed in a square representing a vertex in the figure) and in the complex version, the crosses of the tape represent the vertices (e.g. the robot is placed on a cross representing a vertex in the figure). We set the graphs from Figure 6.17 as the input graphs, and the SMT problem is generated by the robot with the function presented in Section 6.5.1. The input graph is arranged as a grid in both cases. After that, the problem is passed to our proposed system and processed properly. Since there exists a solution for the Hamiltonian cycle problem, a model (a path) is returned, which is then further processed

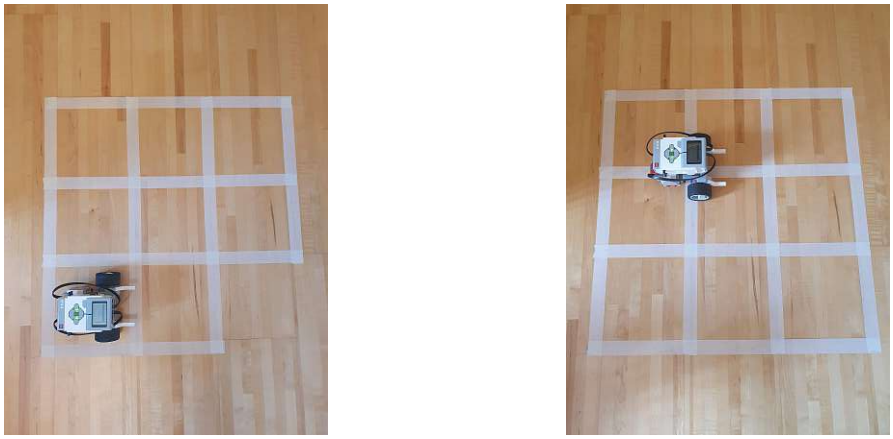


(a) Simple

(b) Complex

Figure 6.17: Use case - Abstract

by the robot's motion module. This scenario could be further extended with additional constraints or further SMT problems during the robot's motion, but for simplicity, we stick to these problems.



(a) Simple

(b) Complex

Figure 6.18: Use case - Real World

In these experiments, we do not simulate latency and compare only Robot Only, DEDs Only, and Q-Learning on Robot + DEDs, as we focus on the use case rather than the benchmarks previously presented.

Figure 6.19 and Figure 6.20 illustrate the results of our experiments. On the left, we compare the three approaches Robot Only, DEDs Only and Q-Learning on Robot + DEDs including the generation of the SMT problem, the processing and the movement of the robot based on the result of the processing. The experiments on the right show the average of five Hamiltonian cycle problems without problem generation and movement

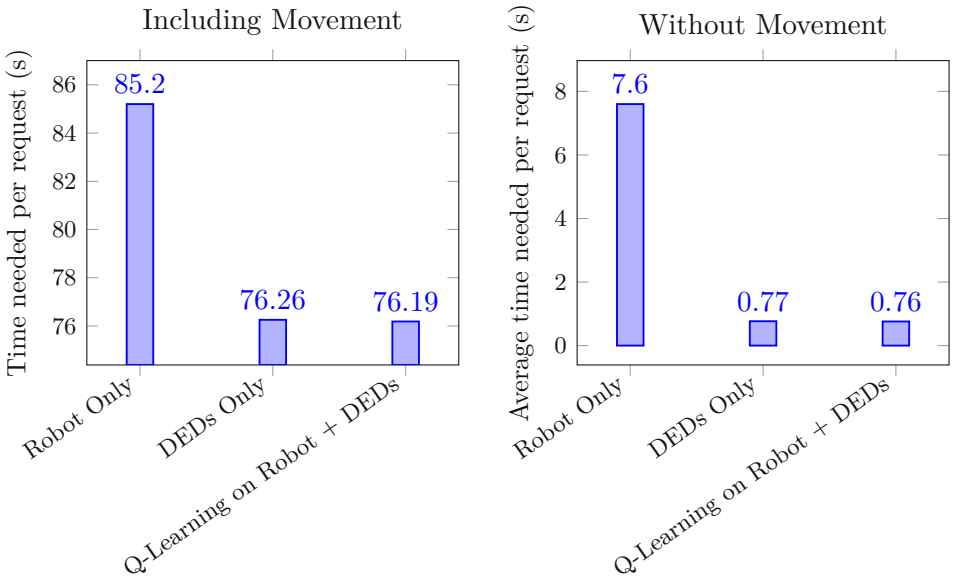


Figure 6.19: Use Case Simple - Experiment Results

of the robot. We see that offloading to the DED is the best option for both the simple and the complex problem, and therefore our approach achieves similar results as DEDs Only. Our mechanism thus is shown to select the best of the available strategies. In more general and complex settings, as we have demonstrated in Section 6.4, it is reasonable to expect that our RL-based mechanism would effectively balance between on-device, edge, and cloud workload execution.

6.6 Summary

The results of all our experiments have further strengthened our hypothesis of the necessity and the performance enhancement of the proposed system that processes SMT workload in a dynamic and adaptive way. The experiments show that solving problems always on the robot is only a good choice for very simple problems. When we do not include the robot in our configurations, we can see from Figure 6.12 that the Q-Learning approach used on the DED improves the performance by 12.65% compared to the DED only approach and could save 43.02% of the required time compared to the cloud only approach. Our experiments confirm that the DQN approach performs slightly better compared to the Q-Learning approach by a factor of 1.02. It must be pointed out that these decision strategies also cause an additional overhead, which is amortised in our experiments. The results shown in Figures 6.15 and 6.16 support that our system also improves the performance based on other goals and in particular energy costs by around 42% compared to robot only or DEDs only. Another finding is that the resources of the devices and the complexity of the problems are the driving indicators for decision making.

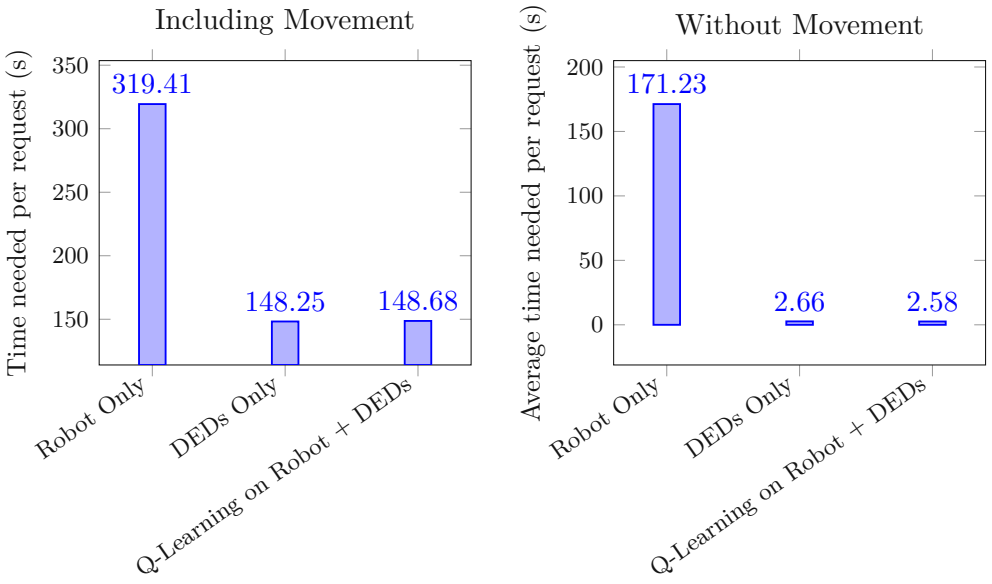


Figure 6.20: Use Case Complex - Experiment Results

"Path Planning for Fog-Supported Robots" underlines the advantages, and shows that our system is able to select the optimal solution for specific use cases.

CHAPTER 7

Conclusion

This thesis has given an account of the fundamentals of computational offloading, reinforcement learning and SMT. The overarching theme of the present work was to combine these areas. Additionally, in the literature review summarised in Chapter 3, we found several systems that could benefit from the approach presented. In this work, we have shown a proof of concept implementation of an architecture for dynamic and adaptive SMT workload serving. We have furthermore outlined that our system can be used in multiple environments and can be extended to support other reward modes. Besides the adaptability using reinforcement learning, our system is also flexible enough to support other decision making mechanisms. Additionally, our design, implementation and evaluation results fulfil the requirements we specified in Section 4.1. Furthermore, we have shown multiple deployment scenarios for our system, spanning over the entire device-to-cloud continuum. Additionally, we presented a concrete use case, in the form of path planning for fog-supported robots. Finally, we provided an analysis of the different scenarios and the corresponding experiment results.

7.1 Adherence to Design Requirements

When considering the requirements we specified in Section 4.1 and the final result achieved, we can observe the following

- **Resource-constrained devices:** The benchmarks in Chapter 6 show deployment scenarios using resource-constrained devices, such as Lego Mindstorms EV3 and single-board computer like Raspberry Pi's. This acts as a proof that our system is also usable in such constrained environments.
- **Multiple Decision Modules:** In our PoC system (see Chapter 5) and the evaluation (see Chapter 6), we show the possibility of different decision approaches

on different devices. Therefore, the best fitting engine, depending on the resources, can be deployed on each device. Moreover, our modularised design (see Chapter 5) allows the integration of further decision making mechanisms in the future, without significant engineering effort for the integration with the rest of the framework.

- **Configuration:** Our PoC implementation shows that very little configuration is needed. Moreover, some parts of the configuration can be reused across all instances and only some small parts must be adapted for each device.

7.2 Revisiting Research Questions

In this section, we will revisit the three research questions we presented in the introduction (see Chapter 1.2). We attempt to answer them directly or refer to the parts of our work where we answer them and highlight the results here again.

RQ. 1: How to architecturally support SMT workloads in the device-to-cloud continuum?

In Chapter 4 we have devised a system architecture for dynamic and adaptive SMT workload serving in the device-to-cloud continuum. We later implement the software architecture in Chapter 5. It is important to create an architecture that is as independent as possible from the concrete underlying setting. Therefore, we use state-of-the-art technologies in the implementation, i.e. HTTP for communication between distributed instances of the system. Where the system allows, we also rely on containerisation, implying an easier and more independent deployment. As we are looking for an architecture in the device-to-cloud continuum, we created a distributed system where the instances of the architecture are stretching over the entire compute continuum. That means the SMT workload can be handled on each layer, i.e. there are instances on the edge, the fog and the cloud. The open question is how to choose the best option to execute the workload resulting in the transition to the next research question.

RQ. 2: How to provide offloading decision support for the evaluation of SMT formulae in specific deployment setups and concrete goals?

The support depends on the goal that is being pursued. Our solution is modular and configurable, i.e. the goals can be configured depending on the deployment setup and the requirements of the operator. Based on that, the indicators for offloading decision support are defined. This flexible way is enabled by the usage of reinforcement learning, where the actor (in our case the decision maker) is able to learn based on the environment (in our case the deployment setup) and rewards (in our case the concrete goals). We found during the experiments that the problem complexity is a very significant indicator of decision making, applying to all the goals we outlined in our work. As we focus on the goal of reducing response time in our experiments, besides the problem complexity, latency is another important input needed for the reinforcement learning actor. We

answer why we use reinforcement learning as the strategy for decision making in Section 4.2.3 and present the details of the concrete implementation in Section 5.4. Additionally, we provide a comprehensive summary of the reinforcement learning fundamentals in Section 2.3.2.

RQ. 3: How can different decision making strategies improve the performance of SMT formulae evaluation within edge settings?

Our findings in Section 6.4 show the improvements in the performance of our decision making strategies using reinforcement learning compared to baseline approaches like a cloud only one. The different edge settings are given by simulating latency and evaluating multiple problems with different complexities. The most remarkable result to emerge from the experiments is that comparing across all problem sets the decision making strategies using reinforcement learning can improve the results by 43.02% comparing Q-Learning and cloud only and 43.92% comparing DQN and cloud only (see Figure 6.12).

7.2.1 Key Takeaways

We have contributed a framework and mechanisms to support the execution of SMT workloads as a service at the edge and beyond.

1. Reinforcement learning provides flexible and adaptive support for decision making in computational offloading.

2. The problem complexity of SMT formulae is a very important indicator for decision making, as this influences the runtime massively.

3. The cost of invoking SMT solvers on very resource-constrained devices is lower only for high latencies to offloading instances and simple problems (in respect to complexity) with the goal of time efficiency.

4. The decision making approach creates additional overhead, but this definitely amortises in changing edge environments.

7.3 Limitations and Future Work

There are a number of parts in our system which could be extended and improved in future work. One area is the deployment strategy. Of course, this depends on the environment in which our proposed system is installed. If the number of devices where the problems are created is increasing, more focus must be set on scalability. As we provided dockerised parts of our system, this could be realised with a container orchestration technology like Kubernetes. Furthermore, the deployment could be more automated to reduce the number of manual steps with tools like Ansible.

In our evaluation and PoC we concentrate on time efficiency (and simplified energy efficiency) of solving problems. Of course, other metrics like traffic-costs, more realistic energy consumption models etc. could be useful, which would lead to the need for a new reward/penalty model for our decision making strategy. This would also result in other input parameters and state information. However, our solution is implemented in a way, that such an extension should not be a big deal.

Another limitation is that we use an "oracle" for the complexity of SMT formulae. This "oracle" is based on preconfigured constant values. For new problems, we always need to add constants and classify the new problems. A more flexible solution or a solution that is more independent of the complexity of the problems is conceivable here.

The communication mechanism is based on REST, which results in synchronous communication as the sender waits for the response. There is the question of what happens to the REST API when a request takes a long time, which is a valid concern for computation-heavy SMT formulae. This problem can be solved in a number of ways, which is potentially future work. One could be to provide an additional endpoint where the status of the evaluation can be queried instead of maintaining the connection open. This means we post the problem to an endpoint that provides a callback where we can get the solution (or the status information that the problem has not been evaluated) instead of waiting for the result. We could also switch to a completely different mechanism like publish-subscribe, however, the most appropriate approach can be application-specific.

APPENDIX A

Data Sets

In the following we list the different data sets and the files used in the present work. They have the form (name: link).

A.1 Simple Data Set

1. 1_1.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_LRA/-/blob/master/2019-ezsmt/robotics/1_1.smt2
2. b161test0003.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_ABV/-/blob/master/bench_ab/b161test0003.smt2
3. cpachecker-induction.32_7a_cilled_true-unreach-call_linux-3.8-rc1-drivers–regulator–isl6271a-regulator.ko-main.cil.out.c.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_UFLRA/-/blob/master/cpachecker-induction-svcomp14/cpachecker-induction.32_7a_cilled_true-unreach-call_linux-3.8-rc1-drivers--regulator--isl6271a-regulator.ko-main.cil.out.c.smt2
4. double_req_bl_0330a_true-unreach-call.c_2.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_FP/-/blob/master/20190429-UltimateAutomizerSvcomp2019/double_req_bl_0240a_true-unreach-call.c_5.smt2
5. storeinv_invalid_t3_pp_sf_ai_00004_001.cvc.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_AUFLIA/-/blob/master/storeinv/storeinv_invalid_t3_pp_sf_ai_00004_001.cvc.smt2
6. RND_3_5.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/LRA/-/blob/master/scholl-smt08/RND/RND_3_5.smt2

7. swap_invalid_t3_np_nf_ai_00010_007.cvc.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_AX/-/blob/master/swap/swap_invalid_t1_np_nf_ai_00004_002.cvc.smt2

8. refcount38.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/AUFDTLIA/-/blob/master/20172804-Barrett/fmf-cav2013/refcount/refcount38.smt2

9. slent_kaluza_201_sink.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_SLIA/-/blob/master/2019-Jiang/slent/slent_kaluza_201_sink.smt2

10. float_req_bl_0877_true-unreach-call.c_0.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_FP/-/blob/master/20190429-UltimateAutomizerSvcomp2019/float_req_bl_0877_true-unreach-call.c_0.smt2

11. javafe.ast.FieldDecl.149.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/UFLIA/-/blob/master/simplify/javafe.ast.FieldDecl.149.smt2

12. swap_invalid_t1_np_nf_ai_00004_002.cvc.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_AX/-/blob/master/swap/swap_invalid_t1_np_nf_ai_00004_002.cvc.smt2

13. contraposition-1.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/UF/-/blob/master/20201221-induction-by-reflection-schoisswohl/reflectiveConjecture/contraposition-1.smt2

14. R509-011___higher_order_proof___why_c5e835_sparkmnhigher_ordermnfold-T-defqtvc___00.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/ufdtlira/-/blob/master/20200306-Kanig/spark2014bench/R509-011__higher_order_proof__why_896899_sparkmnhigher_ordermnfold-T-defqtvc__00.smt2

A.2 Medium Problem Set

1. RND_6_39.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/LRA/-/blob/master/scholl-smt08/RND/RND_6_39.smt2

2. mutex1.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_LIA/-/blob/master/2019-cmodelsdiff/mutualExclusion/mutex1.smt2

3. cruise-control.nosummaries.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_LIRA/-/blob/master/LCTES/cruise-control.nosummaries.smt2
4. qlock.induction.7.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_ALIA/-/blob/master/qlock2/qlock.induction.7.smt2
5. javafe.ast.OnDemandImportDecl.275.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/UFLIA/-/blob/master/simplify/javafe.ast.OnDemandImportDecl.275.smt2
6. 099-incremental_scheduling-15631-0.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_LIA/-/blob/master/2019-ezsmt/incrementalScheduling/099-incremental_scheduling-15631-0.smt2
7. 182.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/BV/-/blob/master/2017-Preiner-psyco/182.smt2

A.3 Hard Problem Set

1. 183.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_SLIA/-/blob/master/2019-full_str_int/py-conbyte_cvc4/leetcode_int-addStrings/183.smt2
2. splice_true-unreach-call_false-valid-memtrack.i_10.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/abv/-/blob/master/20190429-UltimateAutomizerSvcomp2019/splice_true-unreach-call_false-valid-memtrack.i_10.smt2
3. bresenham-ll_valuebound2-O0.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NIA/-/blob/master/20210219-Dartagnan/ReachSafety-Loops/bresenham-ll_valuebound2-O0.smt2
4. s3_srvr.blast.01_false-unreach-call.i.cil.c_0.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_AUFNIA/-/blob/master/UltimateAutomizer/s3_srvr.blast.01_false-unreach-call.i.cil.c_0.smt2
5. 151_gcc.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/aufbv/-/blob/master/20210301-Alive2/gcc/151_gcc.smt2
6. egcd3-ll_valuebound50-O0.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NIA/-/blob/master/20210219-Dartagnan/ReachSafety-Loops/egcd3-ll_valuebound50-O0.smt2

7. digital-stopwatch.locals.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NIA/-/blob/master/LCTES/digital-stopwatch.locals.smt2

8. orb05_700.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_RDL/-/blob/master/scheduling/orb05_700.smt2

9. javafe.ast.ArrayRefExpr.40.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/UFLIA/-/blob/master/simplify/javafe.ast.ArrayRefExpr.40.smt2

A.4 Mixed Problem Set

1. swap_invalid_t3_np_nf_ai_00010_007.cvc.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_AX/-/blob/master/swap/swap_invalid_t3_np_nf_ai_00010_007.cvc.smt2

2. refcount38.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/AUFDTLIA/-/blob/master/20172804-Barrett/fmf-cav2013/refcount/refcount38.smt2

3. contraposition-1.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/UF/-/blob/master/20201221-induction-by-reflection-schoisswohl/reflectiveConjecture/contraposition-1.smt2

4. double_req_bl_0240a_true-unreach-call.c_5.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_FP/-/blob/master/20190429-UltimateAutomizerSvcomp2019/double_req_bl_0240a_true-unreach-call.c_5.smt2

5. cpachecker-induction.32_7a_cilled_true-unreach-call_linux-3.8-rc1-drivers–regulator–isl6271a-regulator.ko-main.cil.out.c.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_UFLRA/-/blob/master/cpachecker-induction-svcomp14/cpachecker-induction.32_7a_cilled_true-unreach-call_linux-3.8-rc1-drivers--regulator--isl6271a-regulator.ko-main.cil.out.c.smt2

6. javafe.ast.FieldDecl.149.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/UFLIA/-/blob/master/simplify/javafe.ast.FieldDecl.149.smt2

7. R509-011___higher_order_proof___why_896899_sparkmnhigher_ordermnfold-T-defqtvc___00.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/ufdtlira/-/blob/master/20200306-Kanig/spark201

4bench/R509-011__higher_order_proof__why_896899_sparkmnhigher_ordermnfold-T-defqtvc__00.smt2

8. storeinv_invalid_t3_pp_sf_ai_00004_001.cvc.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_AUFLIA/-/blob/master/storeinv/storeinv_invalid_t3_pp_sf_ai_00004_001.cvc.smt2
9. 182.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/BV/-/blob/master/2017-Preiner-psyco/182.smt2
10. orb05_700.smt2: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_RDL/-/blob/master/scheduling/orb05_700.smt2

APPENDIX B

Use Case SMT-LIB Encoding

In the following, we list the encoding of two use cases presented in 6.5 in the SMT-LIB format.

B.1 Simple Use Case

```
(set-option :produce-models true)
(set-logic QF_LIA)
(declare-const v0 Int)
(declare-const v1 Int)
(declare-const v2 Int)
(declare-const v3 Int)
(declare-const v4 Int)
(declare-const v5 Int)
(declare-const v6 Int)
(declare-const v7 Int)
(assert (= v0 0))
(assert (or(= v1 (mod (+ v0 1) 8))(= v3 (mod (+ v0 1) 8))))
(assert (or(= v0 (mod (+ v1 1) 8))(= v2 (mod (+ v1 1) 8))(= v4 (mod (+ v1 1) 8))))
(assert (or(= v1 (mod (+ v2 1) 8))(= v5 (mod (+ v2 1) 8))))
(assert (or(= v0 (mod (+ v3 1) 8))(= v4 (mod (+ v3 1) 8))(= v6 (mod (+ v3 1) 8))))
(assert (or(= v1 (mod (+ v4 1) 8))(= v3 (mod (+ v4 1) 8))(= v5 (mod (+ v4 1) 8))(= v7 (mod (+ v4 1) 8))))
(assert (or(= v2 (mod (+ v5 1) 8))(= v4 (mod (+ v5 1) 8))))
(assert (or(= v3 (mod (+ v6 1) 8))(= v7 (mod (+ v6 1) 8))))
(assert (or(= v4 (mod (+ v7 1) 8))(= v6 (mod (+ v7 1) 8))))
(check-sat)
(get-model)
```

Listing B.1: Encoding of Simple Use Case in SMT-LIB Format

B.2 Complex Use Case

```
(set-option :produce-models true)
(set-logic QF_LIA)
(declare-const v0 Int)
(declare-const v1 Int)
(declare-const v2 Int)
(declare-const v3 Int)
(declare-const v4 Int)
(declare-const v5 Int)
(declare-const v6 Int)
(declare-const v7 Int)
(declare-const v8 Int)
(declare-const v9 Int)
(declare-const v10 Int)
(declare-const v11 Int)
(declare-const v12 Int)
(declare-const v13 Int)
(declare-const v14 Int)
(declare-const v15 Int)
(assert (= v0 0))
(assert (or(= v1 (mod (+ v0 1) 16))(= v4 (mod (+ v0 1) 16))))
(assert (or(= v0 (mod (+ v1 1) 16))(= v2 (mod (+ v1 1) 16))(= v5 (mod (+ v1
    1) 16))))
(assert (or(= v1 (mod (+ v2 1) 16))(= v3 (mod (+ v2 1) 16))(= v6 (mod (+ v2
    1) 16))))
(assert (or(= v2 (mod (+ v3 1) 16))(= v7 (mod (+ v3 1) 16))))
(assert (or(= v0 (mod (+ v4 1) 16))(= v5 (mod (+ v4 1) 16))(= v8 (mod (+ v4
    1) 16))))
(assert (or(= v1 (mod (+ v5 1) 16))(= v4 (mod (+ v5 1) 16))(= v6 (mod (+ v5
    1) 16))(= v9 (mod (+ v5 1) 16))))
(assert (or(= v2 (mod (+ v6 1) 16))(= v5 (mod (+ v6 1) 16))(= v7 (mod (+ v6
    1) 16))(= v10 (mod (+ v6 1) 16))))
(assert (or(= v3 (mod (+ v7 1) 16))(= v6 (mod (+ v7 1) 16))(= v11 (mod (+ v7
    1) 16))))
(assert (or(= v4 (mod (+ v8 1) 16))(= v9 (mod (+ v8 1) 16))(= v12 (mod (+ v8
    1) 16))))
(assert (or(= v5 (mod (+ v9 1) 16))(= v8 (mod (+ v9 1) 16))(= v10 (mod (+ v9
    1) 16))(= v13 (mod (+ v9 1) 16))))
(assert (or(= v6 (mod (+ v10 1) 16))(= v9 (mod (+ v10 1) 16))(= v11 (mod (+
    v10 1) 16))(= v14 (mod (+ v10 1) 16))))
(assert (or(= v7 (mod (+ v11 1) 16))(= v10 (mod (+ v11 1) 16))(= v15 (mod (+
    v11 1) 16))))
(assert (or(= v8 (mod (+ v12 1) 16))(= v13 (mod (+ v12 1) 16))))
(assert (or(= v9 (mod (+ v13 1) 16))(= v12 (mod (+ v13 1) 16))(= v14 (mod (+
    v13 1) 16))))
(assert (or(= v10 (mod (+ v14 1) 16))(= v13 (mod (+ v14 1) 16))(= v15 (mod (+
     v14 1) 16))))
(assert (or(= v11 (mod (+ v15 1) 16))(= v14 (mod (+ v15 1) 16))))
(check-sat)
(get-model)
```

Listing B.2: Encoding of Complex Use Case in SMT-LIB Format

List of Figures

1.1 Use Case Architecture 3

2.1 Edge Computing Paradigm [SCZ+16] 9
2.2 The agent-environment interaction in reinforcement learning 14
2.3 Q-Learning algorithm 19
2.4 Q-Learning - Overview 19
2.5 DQN - Neural Network Overview 21
2.6 DQN - Data Flow [Pas21] 22
2.7 ReLU vs. Leaky ReLU 24

4.1 Architecture Design 36
4.2 Activity Diagram: Workflow in Architecture 38
4.3 Design of Neural Network for DQN 40

5.1 Architecture Design with Implementation Details 44

6.1 DED Only without Robot 62
6.2 Cloud Only without Robot 63
6.3 Q-Learning on DED + Cloud without Robot 64
6.4 DQN on DED + Cloud without Robot 64
6.5 Robot Only 65
6.6 DEDs Only 65
6.7 Cloud Only 66
6.8 Q-Learning on Robot + DEDs 67
6.9 Q-Learning on Robot + Q-Learning on DEDs + Cloud 67
6.10 Q-Learning on Robot + DQN on DEDs + Cloud 68
6.11 Comparison without Robot 72
6.12 Comparison of all latencies (on average) - without robot 73
6.13 Comparison with robot 75
6.14 Comparison of all latencies (on average) - with robot 76
6.15 Comparison with energy-aware mode 77
6.16 Energy needed per approach 78
6.17 Use case - Abstract 82
6.18 Use case - Real World 82
6.19 Use Case Simple - Experiment Results 83

6.20 Use Case Complex - Experiment Results 84

List of Tables

2.1 Comparison of SMT solvers 12
2.2 Description of theories [C+11] 13

5.1 Important Configuration Options 49

6.1 Specification of Lego Mindstorms EV3 Brick 59
6.2 Specification of Raspberry Pi 59
6.3 Specification of external Cloud VM 60
6.4 Specification of TU Vienna Cloud VM 1 60
6.5 Specification of TU Vienna Cloud VM 2 60
6.6 Specification of TU Vienna Cloud VM 3 61
6.7 Problem Sets 61

List of Algorithms

2.1 Epsilon-Greedy Algorithm 20

Listings

5.1 Transformation of continuous values to discrete values 51
5.2 Neural Network Class 52
5.3 Function for optimising the model 53
6.1 Fill SMT-LIB file for Hamiltonian Cycle Problem 79
6.2 The move to logic of the robot 80
6.3 Move logic of robot 80
B.1 Encoding of Simple Use Case in SMT-LIB Format 95
B.2 Encoding of Complex Use Case in SMT-LIB Format 96

Bibliography

[AJ09] Fatima M. Albar and Antonie J. Jetter. Heuristics in decision making. In *PICMET '09 - 2009 Portland International Conference on Management of Engineering Technology*, pages 578–584, 2009.

[BCC+17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.

[BCD+11] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[BDW18] Anurag Bhardwaj, Wei Di, and Jianing Wei. *Deep Learning Essentials: Your hands-on guide to the fundamentals of deep learning and neural network modeling*. Packt Publishing Ltd, 2018.

[BKEI09] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008-05*, 11, 2009.

[BKM14] Clark Barrett, Daniel Kroening, and Tom Melham. Problem solving for the 21st century, 2014.

[BMLPT19] Arthur Bit-Monnot, Francesco Leofante, Luca Pulina, and Armando Tacchella. SMT-based Planning for Robots in Smart Factories. In Franz Wotawa, Gerhard Friedrich, Ingo Pill, Roxane Koitz-Hristov, and Moonis Ali, editors, *Advances and Trends in Artificial Intelligence. From Theory to Practice*, pages 674–686, Cham, 2019. Springer International Publishing.

[BST+10] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, England)*, volume 13, page 14, 2010.

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343, Cham, 2018. Springer International Publishing.

[C+11] David R Cok et al. The smt-libv2 language and tools: A tutorial. *Language c*, pages 2010–2011, 2011.

[CFS18] Youdong Chen, Qiangguo Feng, and Weisong Shi. An industrial robot system based on edge computing: An early experience. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[CJS13] Miguel Castro, Antonio J Jara, and Antonio FG Skarmeta. Smart lighting solutions for smart cities. In *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pages 1374–1379. IEEE, 2013.

[CLJC21] Patrik Cervall, Anette Lundvall, Peter Jonsson, and Stephen Carson. Ericsson Mobility Report November 2021. `https://www.ericsson.com/en/reports-and-papers/mobility-report/reports/november-2021`, November 2021. (Accessed on 05/12/2022).

[Cro] The GNU configure and build system - Cross Compilation Tools. `https://www.airs.com/ian/configure/configure_5.html`. (Accessed on 01/13/2022).

[CWC+18] Baotong Chen, Jiafu Wan, Antonio Celesti, Di Li, Haider Abbas, and Qin Zhang. Edge computing in iot-based manufacturing. *IEEE Communications Magazine*, 56(9):103–109, 2018.

[CZW+19] Xianfu Chen, Honggang Zhang, Celimuge Wu, Shiwen Mao, Yusheng Ji, and Medhi Bennis. Optimized Computation Offloading Performance in Virtual Edge Computing Systems Via Deep Reinforcement Learning. *IEEE Internet of Things Journal*, 6(3):4005–4018, 2019.

[Dee] Deep Learning Frameworks Speed Comparison - Deeply Thought. `https://wrosinski.github.io/deep-learning-frameworks/`. (Accessed on 03/24/2022).

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[DMB11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[DRK14] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering*, pages 610–614, 2014.

[EA+18] Joan Espasa Arxer et al. Smt techniques for planning problems, 2018.

[EKJ96] Bernard Espiau, Konstantinos Kapellos, and Muriel Jourdan. Formal verification in robotics: Why and how? In *Robotics Research*, pages 225–236. Springer, 1996.

[ETS] ETSI - Multi-access Edge Computing - Standards for MEC. `https://www.etsi.org/technologies/multi-access-edge-computing`. (Accessed on 03/21/2022).

[ev3a] Using docker to cross-compile. `https://www.ev3dev.org/docs/tutorials/using-docker-to-cross-compile/`. (Accessed on 01/13/2022).

[ev3b] ev3dev is your EV3 re-imagined. `https://www.ev3dev.org/`. (Accessed on 04/07/2022).

[Eva11] Dave Evans. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*, 1(2011):1–11, 2011.

[fla] Welcome to flask — flask documentation (2.1.x). `https://flask.palletsprojects.com/en/2.1.x/`. (Accessed on 01/13/2022).

[Gér19] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2019.

[GM15] Marco Gario and Andrea Micheli. Pysmt: a solver-agnostic library for fast prototyping of smt-based algorithms. In *SMT Workshop 2015*, 2015.

[goMaECM] ETSI Our group on Multi-access Edge Computing (MEC). ETSI - Our group on Multi-access Edge Computing (MEC). `https://www.etsi.org/committee/1425-mec`. (Accessed on 03/21/2022).

[GR17] Bob Gill and Santhosh Rao. Technology insight: Edge computing in support of the internet of things. Technical report, Gartner Research Report, 2017.

[Gra21] Edge computing market share & trends report, 2021-2028. `https://www.grandviewresearch.com/industry-analysis/edge-computing-market`, May 2021. (Accessed on 05/12/2022).

[HJS+20] Chaitra Hegde, Zifan Jiang, Pradyumna Byappanahalli Suresha, Jacob Zelko, Salman Seyedi, Monique A Smith, David W Wright, Rishikesan Kamaleswaran, Matt A Reyna, and Gari D Clifford. Autotriage-an open

source edge computing raspberry pi-based clinical screening system. *medrxiv*, 2020.

[HN92] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[HST+14] William N. N. Hung, Xiaoyu Song, Jindong Tan, Xiaojuan Li, Jie Zhang, Rui Wang, and Peng Gao. Motion planning with Satisfiability Modulo Theories. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 113–118, 2014.

[HTW12] Guoqiang Hu, Wee Peng Tay, and Yonggang Wen. Cloud robotics: architecture, challenges and applications. *IEEE network*, 26(3):21–28, 2012.

[Hö14] Andrea Höfler. SMT Solver Comparison. `https://spreadsheets.ist.tugraz.at/wp-content/uploads/sites/3/2015/06/DS_Hoefler.pdf`, July 2014. (Accessed on 02/04/2022).

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[IS19a] Frank Imeson and Stephen L Smith. An smt-based approach to motion planning for multiple robots with complex constraints. *IEEE Transactions on Robotics*, 35(3):669–684, 2019.

[IS19b] Frank Imeson and Stephen L. Smith. An smt-based approach to motion planning for multiple robots with complex constraints. *IEEE Transactions on Robotics*, 35(3):669–684, 2019.

[KBS05] Vijay Kumar, George Bekey, and Arthur Sanderson. Networked robots. *WTEC Panel on INTERNATIONAL ASSESSMENT OF RESEARCH AND DEVELOPMENT IN ROBOTICS DRAFT REPORT*, pages 57–72, 2005.

[Lap18] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[LBL+16] Chieh-Jan Mike Liang, Lei Bu, Zhao Li, Junbei Zhang, Shi Han, Börje F. Karlsson, Dongmei Zhang, and Feng Zhao. Systematically Debugging IoT Control System Correctness for Building Automation. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*, BuildSys '16, page 133–142, New York, NY, USA, 2016. Association for Computing Machinery.

[LGLL18] Ji Li, Hui Gao, Tiejun Lv, and Yueming Lu. Deep reinforcement learning based computation offloading and resource allocation for MEC. In

2018 IEEE Wireless Communications and Networking Conference (WCNC), pages 1–6, 2018.

[LHP+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[MAH+16] Mujahid Mohsin, Zahid Anwar, Ghaith Husari, Ehab Al-Shaer, and Mohammad Ashiqur Rahman. IoTSAT: A formal framework for security analysis of the internet of things (IoT). In *2016 IEEE Conference on Communications and Network Security (CNS)*, pages 180–188, 2016.

[MC20] Meysam Masoudi and Cicek Cavdar. Device vs Edge Computing for Mobile Services: Delay-Aware Decision Making to Minimize Power Consumption. *IEEE Transactions on Mobile Computing*, 06 2020.

[McF18] Roger McFarlane. A survey of exploration strategies in reinforcement learning. *McGill University*, 2018.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[MTK16] Robin R Murphy, Satoshi Tadokoro, and Alexander Kleiner. Disaster robotics. In *Springer handbook of robotics*, pages 1577–1604. Springer, 2016.

[NDW+19a] Zhaolong Ning, Peiran Dong, Xiaojie Wang, Lei Guo, Joel J. P. C. Rodrigues, Xiangjie Kong, Jun Huang, and Ricky Y. K. Kwok. Deep Reinforcement Learning for Intelligent Internet of Vehicles: An Energy-Efficient Computational Offloading Scheme. *IEEE Transactions on Cognitive Communications and Networking*, 5(4):1060–1072, 2019.

[NDW+19b] Zhaolong Ning, Peiran Dong, Xiaojie Wang, Joel J. P. C. Rodrigues, and Feng Xia. Deep Reinforcement Learning for Vehicular Edge Computing: An Intelligent Offloading System. *ACM Trans. Intell. Syst. Technol.*, 10(6), oct 2019.

[Numa] Number of Internet of Things (IoT) connected devices worldwide in 2018, 2025 and 2030. `https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/`. (Accessed: 02/04/2022).

[Numb] Numpy documentation — numpy v1.22 manual. https://numpy.org/doc/stable/. (Accessed on 01/13/2022).

[OM06] James A O'brien and George M Marakas. *Management information systems*, volume 6. McGraw-Hill Irwin, 2006.

[Pas21] Adam Paszke. Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 1.11.0+cu102 documentation. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html, 2021. (Accessed on 03/22/2022).

[PD21] Victor Casamayor Pujol and Schahram Dustdar. Fog robotics—understanding the research challenges. *IEEE Internet Computing*, 25(5):10–17, 2021.

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

[PM10] David L Poole and Alan K Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[psu] psutil documentation — psutil 5.9.1 documentation. https://psutil.readthedocs.io/en/latest/. (Accessed on 01/13/2022).

[PVB21] Martin Pech, Jaroslav Vrchota, and Jiří Bednář. Predictive maintenance and intelligent sensors in smart factory. *Sensors*, 21(4):1470, 2021.

[pyta] Api reference — python-ev3dev 2.1.0.post1 documentation. https://ev3dev-lang.readthedocs.io/projects/python-ev3dev/en/stable/spec.html. (Accessed on 01/13/2022).

[pytb] pythonping · pypi. https://pypi.org/project/pythonping/. (Accessed on 01/13/2022).

[pyy] Pyyaml documentation. https://pyyaml.org/wiki/PyYAMLDocumentation. (Accessed on 01/13/2022).

[Req] Requests: Http for humans™ — requests 2.27.1 documentation. https://docs.python-requests.org/en/latest/. (Accessed on 01/13/2022).

[RL14] Mark Rollins and Mannie Lowe. *Beginning Lego Mindstorms Ev3*, volume 253. Springer, 2014.

[Rud16] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[SAA+22] Sina Shahhosseini, Arman Anzanpour, Iman Azimi, Sina Labbaf, DongJoo Seo, Sung-Soo Lim, Pasi Liljeberg, Nikil Dutt, and Amir M. Rahmani. Exploring computation offloading in IoT systems. *Information Systems*, 107:101860, 2022.

[Sat17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[SB18] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[SCZ+16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[SMT] SMT-LIB The Satisfiability Modulo Theories Library. `https://smtlib.cs.uiowa.edu/`. (Accessed on 03/21/2022).

[SOM14] F. Shrouf, J. Ordieres, and G. Miragliotta. Smart factories in industry 4.0: A review of the concept and of energy management approached in production based on the internet of things paradigm. In *2014 IEEE International Conference on Industrial Engineering and Engineering Management*, pages 697–701, 2014.

[SPKS12] Anuj Sehgal, Vladislav Perelman, Siarhei Kuryla, and Jurgen Schonwalder. Management of resource constrained devices in the internet of things. *IEEE Communications Magazine*, 50(12):144–149, 2012.

[SY15] Yan-Yan Song and LU Ying. Decision tree methods: applications for classification and prediction. *Shanghai archives of psychiatry*, 27(2):130, 2015.

[Tok21] A. Aylin Tokuç. Solving the K-Armed Bandit Problem | Baeldung on Computer Science. `https://www.baeldung.com/cs/k-armed-bandit-problem`, February 2021. (Accessed on 03/23/2022).

[TSM+17] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. *IEEE Communications Surveys Tutorials*, 19(3):1657–1681, 2017.

[Tur14] James Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[Vai21] Lionel Sujay Vailshery. Most used languages among software developers globally 2021 | statista. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/, August 2021. (Accessed on 01/16/2022).

[vHGS16] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1), Mar. 2016.

[wat] Watchdog — watchdog 2.1.5 documentation. https://python-watchdog.readthedocs.io/en/stable/. (Accessed on 01/13/2022).

[Wei03] Eric W Weisstein. Hamiltonian path. *https://mathworld. wolfram. com/*, 2003.

[Wenne] Lilian Weng. Exploration Strategies in Deep Reinforcement Learning | Lil'Log. https://lilianweng.github.io/posts/2020-06-07-exploration-drl/, 2020 June. (Accessed on 03/30/2022).

[WMZT22] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, 9(2):187–212, Apr 2022.

[WSH+16] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[ZML+16] Ke Zhang, Yuming Mao, Supeng Leng, Quanxin Zhao, Longjiang Li, Xin Peng, Li Pan, Sabita Maharjan, and Yan Zhang. Energy-efficient offloading for mobile edge computing in 5g heterogeneous networks. *IEEE Access*, 4:5896–5907, 2016.

[ZZL16] Xiaohui Zhao, Liqiang Zhao, and Kai Liang. An energy consumption oriented offloading algorithm for fog computing. In *International conference on heterogeneous networking for quality, reliability, security and robustness*, pages 293–301. Springer, 2016.