UNIVERSITY OF CALIFORNIA,
IRVINE

Resilient Computation Offloading for Real-Time Mobile Autonomous Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Davide Callegaro

2021

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

My gratitude goes first to **Professor Marco Levorato**, who guided me through my doctorate. It has been amazing to work on problems together, bringing ideas into reality, always with passion and kindness. The long days in the laboratory to put the finishing touches on a submission have been inspirational. Thank you Marco for pushing me beyond my comfort zone. It has been remarkable to have your support as an academic advisor, and I am grateful to have you as one of my mentors.

A great deal of gratitude goes to a former lab-mate, now **Prof. Sabur Baidya**, who made research fun and who has always been up to the challenge! Prof. Baidya is an amazing researcher, and I wish him the very best in his future!

I really appreciated the time and energy that **Prof. Francesco Restuccia** put in our work. Thank you for all the technical guidance, presentation advice and powerful encouragements!

I want to thank all the colleagues I had the pleasure of working with. It has always been a fantastic experience to spend time with each and every one of you: Ali, Anas, Delaram, Igor, Peyman, Sharon, Yoshitomo.

I would like to thank the other members of my dissertation committee, **Professor Nikil Dutt** and **Professor Nalini Venkatasubramanian** for their valuable feedback and suggestions. I can honestly say I enjoyed my final examination, and I have to thank you for making it an interesting conversation!

My experience at UC Irvine would not have been the same without all the students who helped me mature and understand how to enable them to progress in their work. Special thanks to all the teams, for also the joy and fun you brought to work!

Thank you to the external collaborators **Gowri Ramachandran** and **Prof. Bhaskar Krishnamachari**, part of the amazing Deep Edge team, which was the start of my main line of research.

Big thanks to my Internships' advisors: **Akhilesh Yoshi** and **Sandeep Reddy Goli**, **Marcellino Gemelli**, who helped me see real problems in the industry and bring some of the tools back to academia.

I want to acknowledge that my research was supported in part by: Prof. Marco Levorato, Donald Bren School of Information and Computer Sciences at UCI, Intel Corporation and NSF grants IIS-1724331 and MLWiNS-2003237 and DARPA under grant HR00111910001. I will be always thankful to the people and organizations that helped me achieve what is my best work yet!

Finally I would like to thank my family and friends for their support. I could have not started a doctorate program without the encouragement and support of my parents, **Massimo** and **Margerita**. I hope this achievement makes them and my brother **Andrea** proud!

The COVID-19 pandemic hit during 2020, and since the beginning of that year, I spent most of my time working from home with my girlfriend **Reebbhaa**. She made challenges look easy and always facing them with a smile. Her love, trust and support helped me focus on the important things and I am proud of all of our achievements.

VITA

Davide Callegaro

EDUCATION

Doctor of Philosophy in Computer Science	**2021**
University of California, Irvine	*Irvine, California*
Laurea Magistrale in Computer Engineering	**2016**
Universita' degli Studi di Padova	*Padua, Italy*
Laurea Triennale in Information Engineering	**2013**
Universita' degli Studi di Padova	*Padua, Italy*

PROFESSIONAL EXPERIENCE

Research Intern	**Jul.2017–Sep.2017**
Robert Bosch Corporation	*Palo Alto, California*
Research Intern	**Jun.2018–Sep.2018**
Nutanix	*San Jose, California*

ACADEMIC RESEARCH EXPERIENCE

Intelligent & Autonomous Systems Laboratory	**2016–2021**
University of California, Irvine	*Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant	**2016–2018**
University of California, Irvine	*Irvine, California*

REFEREED JOURNAL PUBLICATIONS

Optimal Edge Computing for Infrastructure-Assisted UAV Systems **2021**
IEEE Transactions on Vehicular Technology

Head Network Distillation: Splitting Distilled Deep Neural Networks for Resource-Constrained Edge Computing Systems **2020**
IEEE Access

Cloud-Backed Mobile Cognition Power-Efficient Deep Learning in the Autonomous Vehicle Era **2021**
Springer Computing

REFEREED CONFERENCE PUBLICATIONS

SeReMAS: Self-Resilient Mobile Autonomous Systems Through Predictive Edge Computing **Jul.2021**
18th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)

Optimal Task Allocation for Time-Varying Edge Computing Systems with Split DNNs **Dec.2020**
IEEE Global Communications Conference (GLOBECOM)

Dynamic Distributed Computing for Infrastructure-Assisted Autonomous UAVs **Jun.2020**
Proceedings of the IEEE International Conference on Communications (ICC)

Information Autonomy: Self-Adaptive Information Management for Edge-Assisted Autonomous UAV Systems **Nov.2019**
Proceedings of the IEEE Military Communication Conference (MILCOM)

Distilled Split Deep Neural Networks for Edge-Assisted Real-Time Systems **Oct.2019**
Proceedings of the ACM International Conference on Mobile Computing and Networking (MobiCom)

A Measurement Study on Edge Computing for Autonomous UAVs **Aug.2019**
Proceedings of the Workshop on Mobile Air-Ground Edge Computing, Systems, Networks, and Applications (MAGESys)

Optimal Computation Offloading in Edge-Assisted UAV Systems **Dec.2018**
IEEE Global Communications Conference (GLOBECOM)

Intelligent Data Filtering in Constrained IoT Systems **Nov.2017**
Proceedings of the Asilomar Conference on Signals, Systems, and Computers (ACSSC)

ABSTRACT OF THE DISSERTATION

Resilient Computation Offloading for Real-Time Mobile Autonomous Systems

By

Davide Callegaro

Doctor of Philosophy in Computer Science

University of California, Irvine, 2021

Associate Professor Marco Levorato, Chair

The operations of Mobile Autonomous Systems (MAS) rely on real-time data analysis. For instance, autonomous vehicles' navigation requires the low-latency analysis of high resolution images to detect, and avoid, objects. Unfortunately, many Mobile Autonomous Systems (MASs) have constrained computing and energy resources, and the continuous execution of state-of-the-art algorithms is out of their reach. By offloading the processing load to compute-capable device located at the network edge, the edge computing paradigm can mitigate this issue. However in practical real-world settings, the wireless channel connecting the mobile devices to the edge server often presents erratic capacity patterns due to mobility. As a result, the overall delay perceived by the mobile application may be affected by large variations, which in turn harm control.

This thesis explores solutions to the problem described above. To this aim, in addition to new concepts and edge offloading strategies, a complete real-world platform – the HyDRA platform, was developed to support design and evaluation, as well as dataset collection. HyDRA is a fully open source software and hardware platform realizing flexible machine learning-empowered computing for MAS. From a hardware perspective, HyDRA is composed of several Unmanned Aerial Vehicles (UAVs) and ground devices collaboratively performing

data analysis to accomplish system-wide goals. From a software perspective, the HyDRA middleware enables real-time control of data and task routing within the system, organized as a distributed set of modules transforming the data captured by the MAS into actuable control.

Taking HyDRA as a starting point, this thesis makes the following conceptual contributions:

• the end-to-end delay in remote computing settings for MAS – and specifically autonomous quad-copters – were characterized by means of real-world experiments that produced a comprehensive dataset focused on object detection from images. The study considered both Wi-Fi and Long-Term Evolution connectivity, and several embedded computing platforms. The results demonstrates the instability of application level delay even in line-of-sight settings and relatively slow vehicle motion.

• A framework for the dynamic control of task offloading in MASs with extreme temporal variations is developed. The frameworks is based on a preliminary experimental analysis, which indicates that there is no dominant feature, including obvious features such as channel quality, and that prediction necessitates an ensemble of *weaker* features. We first mathematically formulate a Redundant Task Offloading Problem. Then, we create predictors that can help manage the resource usage/performance trade-off. Specifically, we propose a myopic predictor as baseline and a DRL-based approach, which operates on a set of features from application, network and device-level components. To the best of our knowledge, this is the first framework addressing the problem of redundant task offloading in MAS with a data-driven approach which efficacy is verified in a real-world testbed and with replicable dataset-based experiments.

• A modeling and optimization framework based on Markov Decision Processes (MDP) was developed to analyze the structural properties of dynamic control strategies determining where information is processed in collaborative computing scenarios for MAS. In this section of the thesis, the focus of control is primarily between local and remote analysis. Using recent split Deep Neural Networks deep neural network (DNN) techniques, the framework also controls at a fine-grain how the analysis task (the DNN in this case) is divided between the mobile device and the edge server based on current system parameters.

Chapter 1

Introduction

1.1 Copyright Notice

Some material contained withing this dissertation has been previously published and is used with permission.

- Copyright © 2018 IEEE. Reprinted with permission, from Davide Callegaro, and Marco Levorato, "Optimal Computation Offloading in Edge-Assisted UAV Systems" In Proceedings of the IEEE Global Communications Conference (GLOBECOM), Abu Dhabi, United Arab Emirates, Dec. 2018.
- Copyright © 2019 IEEE. Reprinted with permission, from Davide Callegaro, Sabur Baidya, Gowri S. Ramachandran, Bhaskar Krishnamachari, and Marco Levorato, "Information Autonomy: Self-Adaptive Information Management for Edge-Assisted Autonomous UAV Systems" In IEEE MILCOM 2019-IEEE Military Communications Conference (MILCOM).
- Copyright © 2020 IEEE. Reprinted with permission, from Davide Callegaro, Yoshitomo

Matsubara, and Marco Levorato, "Optimal Task Allocation for Time-Varying Edge Computing Systems with Split DNNs" In IEEE .

- Copyright © 2021 IEEE. Reprinted with permission, from Davide Callegaro, Marco Levorato, and Francesco Restuccia, "SeReMAS: Self-Resilient Mobile Autonomous Systems Through Predictive Edge Computing" In Proceedings of 18th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON), 2021.
- Copyright © 2021 IEEE. Reprinted with permission, from Davide Callegaro, and Marco Levorato, "Optimal Edge Computing for Infrastructure-Assisted UAV Systems" In IEEE IEEE Transactions on Vehicular Technology, vol. 70, no. 2, pp. 1782–1792, 2021.

1.2 Motivation: Challenges in Highly Mobile Computing

Mobile Autonomous Computing includes devices running a plethora of different applications: from autonomous driving, to industry 4.0 or UAVs. All these applications require a stream of high dimensional input data to be processed in real-time to allow data driven control to take place. In the last decade this type of processing has been increasingly using DNNs to analyze high dimensionality data, such as video, lidar, etc. For instance, analyzing images using DNNs requires a large execution time, due to the large amount of operations they involve. In order to reduce their computational delay, Graphics Processing Units, Tensor Processing units or Field Programmable Gate Arrays are used.

However, continuous heavy data processing can be problematic for small embedded devices that are computationally and energy constrained. In this category fall UAVs, which thanks to advancements in embedded computers can carry processing units capable of such computational power, but would in such case still have to manage their energy budget. In fact, the computational energy required by these applications can reach 10-15% of the power required to hoover, forcing the device to offload some of its computation to extend its mission time.

1.2.1 Responsiveness

One metric that is central in MASs is their responsiveness to stimuli. In our research, we measure this responsiveness as the capture-to-control delay (also referred here as the task delay), defined as the interval of time between information acquisition and the corresponding action being taken. For instance, if the UAV collects a images at time t, analyzes it locally in δ_cs and updates its mission plan in δ_cs more, the task delay will be equal to the sum of such components. If the analysis of the image is offloaded to a server, we will add the communications components to the task delay: one to transport the image from the device to the server, and one transporting the result from the server to the client.

However, note as the connection between the mobile device and the edge server is necessarily wireless, communication delays are inherently erratic, and dependent on several time-varying variables and parameters. Moreover, the – time varying – state of the overall system also influences the task delay. For example, if the server has higher than usual load, the task will incur queueing delays at the server side. Similarly, if several devices are accessing the wireless channel, the communication delay will increase due to congestion. Moreover, as shown in Chapter 4, mobility among other factors influences the instantaneous channel performance, requiring approaches to mitigate such uncertainty when dealing with mission critical applications.

1.3 Dissertation Contributions & Overview

In this thesis, we make a series of conceptual and system-level contributions, which are presented and discussed in the different chapters:

- in Chapter 3, we introduce the HyDRA framework and we develop an optimization problem used to balance energy usage and network resources. We also present a heuristic algorithm, whose performance we corroborate with experimental results.

- in Chapter 4, we explore a predictive approach to the problem of task offloading. We collect information from different sources (networking, application and board) and create two parametric algorithm to trade-off computational resources, information, networking and server resources.

- in Chapter 5, we take a theoretical approach to the problem of task offloading, and we solve the decision problem of whether to compute on the embedded device or offload to the edge server. We balance task delay and energy consumption at the device, and find the optimal policy using a Markov Decision Process (MDP).

- in Chapter 6, we solve a similar problem, trading-off energy consumption at the device and task delay, but we do so using the innovative split DNNs. When modeling such scenario, we encounter a MDP where tasks are heterogenous (full offloading or split offloading) and which requires us to solve a Linear Fractional Program. We study how the instantaneous throughput influences the optimal policy and show our results over-perform a fixed policy up to 30%.

Chapter 2

Related work

2.1 Real-time Applications in Mobile Internet of Things

2.1.1 Edge Computing in Mobile Autonomous Systems

Edge computing can significantly improve reliability and performance in mobile applications [7]. Different frameworks perform a multi-layer optimization to exploit the full potential of edge computing [61, 19]. To fully exploit the edge servers, the user equipment needs to periodically make a decision on whether to process tasks locally, or to offload. In the latter case, there might be multiple technologies or networks available, e.g. [10], and a link must be chosen for each transmission.

2.1.2 Static Approaches

Convex optimization has been proven to be ineffective due to the presence of complex factors such as user's mobility [13]. Classic approaches are shown to perform better with coarser granularity settings and when considerable prior knowledge is available. For example, in [44] the authors develop an online multi-decision making scheme, solving a task offloading problem while jointly optimizing caching, communication and computation resources in the Internet of Vehicles, exploiting the proximity of users to roadside units.

2.1.3 Data Driven Approaches to Real-Time Scheduling

Fast-changing mobile networks usually employ data-driven approaches, using Markov Decision Processes (MDP), Q-Learning or DRL. MDPs achieve a good tradeoff between the flexibility of learning and the data efficiency of a model-based solution [12, 58]. However, MDP-based solutions often lead to exceedingly large state spaces, and require vast amounts of data to find the correct transitions for each state-action pair during training. Finally, they are very memory intensive both in training and execution time. For these reasons, DRL approaches have been proposed. Cao et. al [13] present a general framework for intelligent offloading in multi-access edge computing composed by observation tier, analysis tier, prediction tier and policy tier. In this chapter, we consider a much more complicated problem where the trade-off is beyond power efficiency and link performance.

2.1.4 Simulation Environments

Recently, researchers have worked towards simulation environments for drones, for example, OpenUAV [50] and FlyNetSim [3]. However, neither of the two environments can capture the interactions between mobility and application delay that are key in this chapter. Thus, we are sharing our dataset with the community to further allow research that can explain and exploit these interactions.

2.2 Theoretical Solutions and Heterogeneous Metrics

Most recent contributions on edge computing for UAVs focus on planning aspects, and mostly from an purely abstract perspective [40, 14, 39, 11] or center their attention on UAV-assisted edge computing and cloudlets [28, 61, 20, 56, 43].

Other contributions, such as, [2], focus, instead, on monetary metrics (for instance associated with communications) to guide the optimization process. In [2] a game theoretic approach is proposed, with which the authors are able to reduce the communication cost, considering not only multiple servers, but also the possibility for some of them to offload the computation further. Our work centers on short-term metrics, which allow a fine degree of control when optimizing the offloading process. We remark how this is a marked difference with respect to most existing literature.

Very recent contributions, such as [16], focus on the optimization of specific scenarios, assuming the apriori knowledge of a prediction model for the channel state, which leads to sophisticated decision making algorithms to determine which part of the task can be offloaded. This interesting class of approaches imposes stronger limitations on the type of analysis task. [30] presents an approach based on fuzzy logic to face the high uncertainty induced by these applications.

In [35], B.Liu et al. propose to offload computationally intensive tasks from UAVs to edge and cloud servers. They formulate a joint computation and routing optimization by defining a three-layered computational model on which they design a polynomial near-optimal approximation algorithm. The authors use a Markov approximation technique described in [17], which is useful when solving network combinatorial optimization problems. However, they do not consider energy-related metrics and control.

Energy expense is considered in the work by Zhu et al. [62], where cooperative approach to computation offloading for UAVs is presented that aims to improve the inefficiency of naive local computing solutions. The authors explore an urban environment for UAV operations, and use simulated annealing to minimize the energy consumption while satisfying a delay constraint. Our solution jointly optimizes energy and delay, offering more flexibility to the application.

Part I

Experimental Study of Task Offloading for MASs

Chapter 3

Dynamic Distributed Scheduling for Infrastructure Autonomous UAVs

As previously discussed, the ability to observe and analyze the surrounding environment to inform decision making is the key to autonomy. In physical systems, state information is extracted by acquiring and processing endogenous and exogenous signals in real-time. Despite the important advances both in algorithms and embedded platforms of the recent years, the execution of sensing-processing-control pipelines in lightweight airborne platforms such as commercial Unmanned Aerial Vehicles (UAV) is a non-trivial problem. Rather intuitively, there exists an inherent tradeoff between three key metrics: accuracy, decision delay, and energy consumption. Improving accuracy of analysis often requires increasing complexity, which comes at the price of a larger execution time, and thus decision delay, or a larger weight and energy consumption.

Herein we focus on video analysis, and specifically on object detection, an important component of most advanced autonomous systems. Modern object detection algorithms take the form of Deep Neural Networks (DNN). The most performing DNNs are extremely complex,

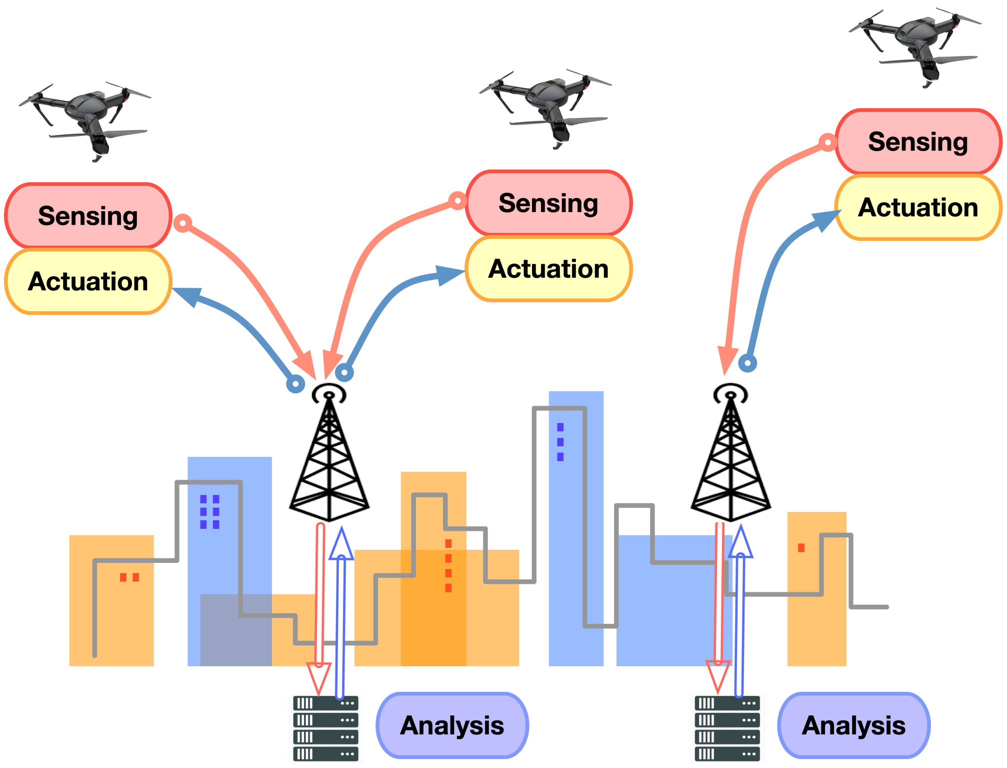


Figure 3.1: Edge computing scenario considered in this chapter: UAVs offload analysis modules to ground edge servers. The architecture we developed enables the seamless distribution of modules across devices, as well as to re-route data analysis in real-time to improve performance.

and their execution requires powerful computing platforms. Two key recent advancements make object detection possible in constrained mobile devices:

- The development of techniques such as distillation, pruning and quantization led to the construction of effective simplified DNN models with a significantly reduced complexity compared to the full models.

- The development of powerful embedded computers equipped with accelerators and GPUs.

Intuitively, despite the clever optimized design of simplified models, aggressive complexity reduction results in a perceivable degradation of accuracy. We remark that lower complexity also means a shorter execution time, that is, a smaller time between information acquisition and control – here referred to as *capture-to-control time* – a critical parameter for an effective control. The modern GPU-equipped embedded boards mentioned above allow fast execution of fairly complex DNN models. However, the use of GPUs to speed up execution significantly increases energy consumption, another crucial metric, especially when considering airborne systems.

This chapter seeks insights on edge computing *for* UAVs from a real-world deployment and real-world testing, especially to characterize and counteract temporal variations in capture-to-control time shaped by variations in the channel conditions, including gain and congestion level. Specifically, we make the following contributions:

(*i*) We develop an experimental platform realizing an infrastructure assisted UAV system. We focus on navigation tasks based on object detection via DNN models, and equip the UAV with one of the most powerful embedded computers for machine learning to enable a fair comparison with offloading to edge servers.

(*ii*) We develop *Hydra*[8], a middleware architecture enabling the adaptive distribution of computation tasks within infrastructure assisted UAV systems. The modular architecture grants significant flexibility in deploying sensing, analysis, and control pipelines, and includes a logic to dynamically activate-deactivate pipelines in response to changes in the state of their components or environment.

(*iii*) We test the architecture to illustrate its performance against variations of channel gain and contention/interference from other mobile devices.

Our experimental results indicate that offloading complex analysis tasks to edge servers grants a significant reduction in the overall energy intake of operating the UAV, thus prolonging mission lifetime. The proposed Hydra architecture is shown to provide reliable control against fluctuations of the capture-to-control time at different temporal scales based on a tunable tradeoff between energy and delay performance.

The rest of the chapter is organized as follows. Section 3.1 introduces the experimental platform and provides a preliminary discussion of the problem at hand. In Section 3.2, we present and discuss the architecture and logics of Hydra. Section 3.3 presents and discusses the experimental results.

3.1 Problem Setup and Preliminary Results

First, we describe the task and experimental platform, and make some preliminary considerations on metrics of interest.

3.1.1 Computation Task

We consider the sensing-analysis-control pipeline illustrated in Fig. 3.2: the onboard camera acquires images that are analyzed using an object detection algorithm, whose output is a series of labeled bounding boxes. The control module selects a bounding box with a predefined label, and produces steering commands. The objective of steering is to center the bounding box with respect of the vision range of the UAV and match it to a predefined size. Note that our objective is to analyze communication-computation aspects of this class of problems, so we focus on the accuracy, delay and energy consumption associated with the pipeline, rather than on the specific output control.

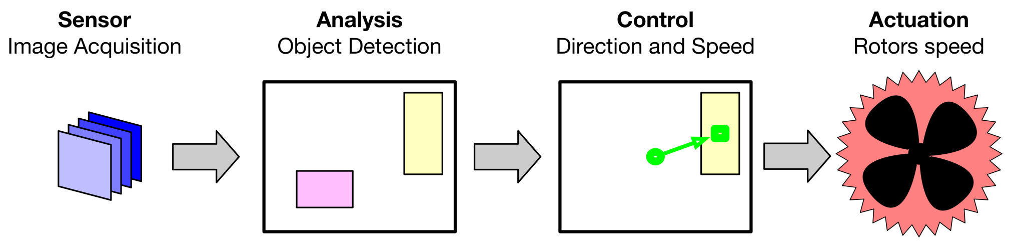


Figure 3.2: Sensing, analysis and control pipeline where object detection performed on images acquired by the UAV is used to control navigation.

We use ssdlite mobilenet v2 [47] trained on the Coco dataset [34] and floating Point 32 bits precision for object detection, a highly optimized model designed for mobile devices. Performance in terms of accuracy is expressed in terms of mean Average Precision (mAP), which measures a combination of precision, recall and bounding box intersection with ground truth. The model we use achieves a mAP equal to 22 [1], compared to the $37 - 43$ of full sized models which are out of reach of most embedded computers.

3.1.2 Experimental Platform

Experiments are performed using a 3DR solo quadcopter customized to mount on an attached plate an additional embedded computer and battery, a GoPro Hero 4 camera, and a Magewell HDMI to USB converter. Specifically, we use the Nvidia Jetson Nano with 4GB RAM, Quad-core ARM Cortex-A57 MPCore processor and 128-core Nvidia Maxwell GPU. We use as edge servers Nvidia Jetson TX2 boards with 8 GB RAM, hex-core ARMv8 64-bit CPU and an integrated 256-core Nvidia Pascal GPU.

The UAV connects to the edge servers using WiFi communications, and specifically IEEE 802.11n, which offers higher data rates (upto 130 Mbps application throughput) compared to IEEE 802.11 a/b/g and can operate both on 2.4 GHz and 5 GHz band. We configure the access points to operate in the 2.4 GHz band on non-overlapping channels.

3.1.3 Preliminary Considerations

The Nvidia Nano is a recently released extremely powerful embedded computer, specifically designed to provide state of the art performance in executing machine learning algorithms. The average time to execute the bare object detection task is 87 ± 6 ms, which almost equal to the 75 ± 8 ms achieved by the Jetson TX2 board. We report that the execution of other

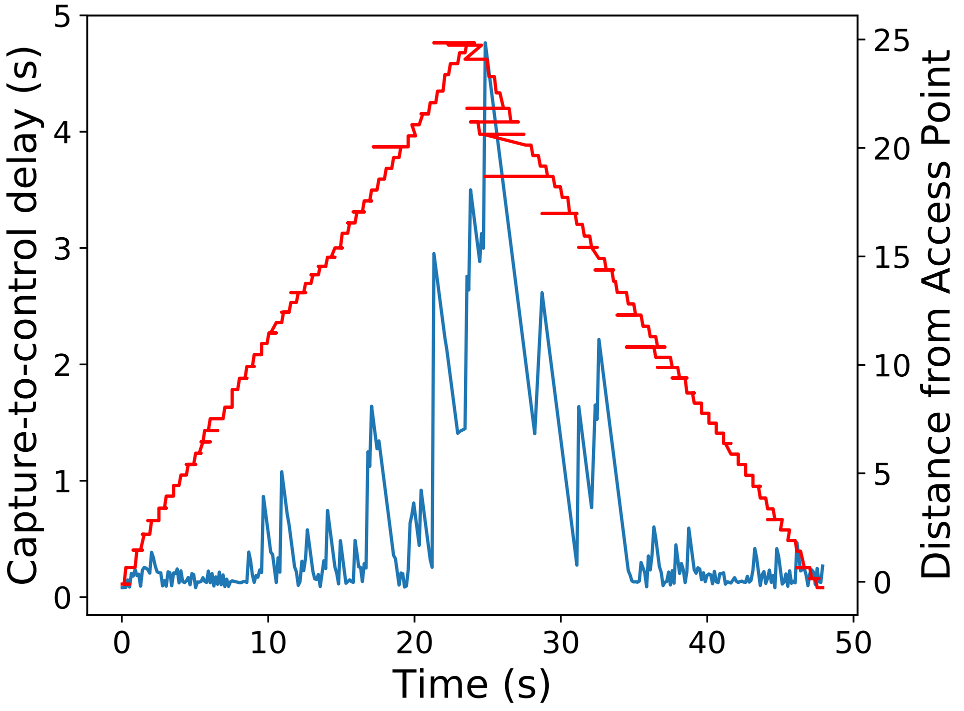


Figure 3.3: Temporal variation of capture-to-output delay and distance from a reference edge server as the UAV moves away from and toward it.

models, such as ssd mobilenet v1, took almost the same time on the Nano and half of the time on the TX2 likely due to specific architectural characteristics. We choose v2 to benchmark our system due to its slightly higher accuracy and to provide the most advantageous conditions to the local analysis option. We remark that a more powerful edge server would obviously advantage remote analysis pipelines, and that any other task using the GPU at the UAV could significantly impact the performance of local onboard analysis loops.

However, the extreme performance of the Nano comes at the price of a high energy consumption. Continuous local computing requires 4 ± 0.5 W (measured during flight using the Jetson Nano utilities), which is more that 10% of the 38 W required by the UAV to hover or navigate. This is mostly connected to the use of the GPU, however, our extensive testing on other embedded computers resulted in execution times above half a second.

Continuous offloading to 1 or 2 edge servers requires 2 ± 0.2 W and 2.055 ± 0.1 W, respectively.

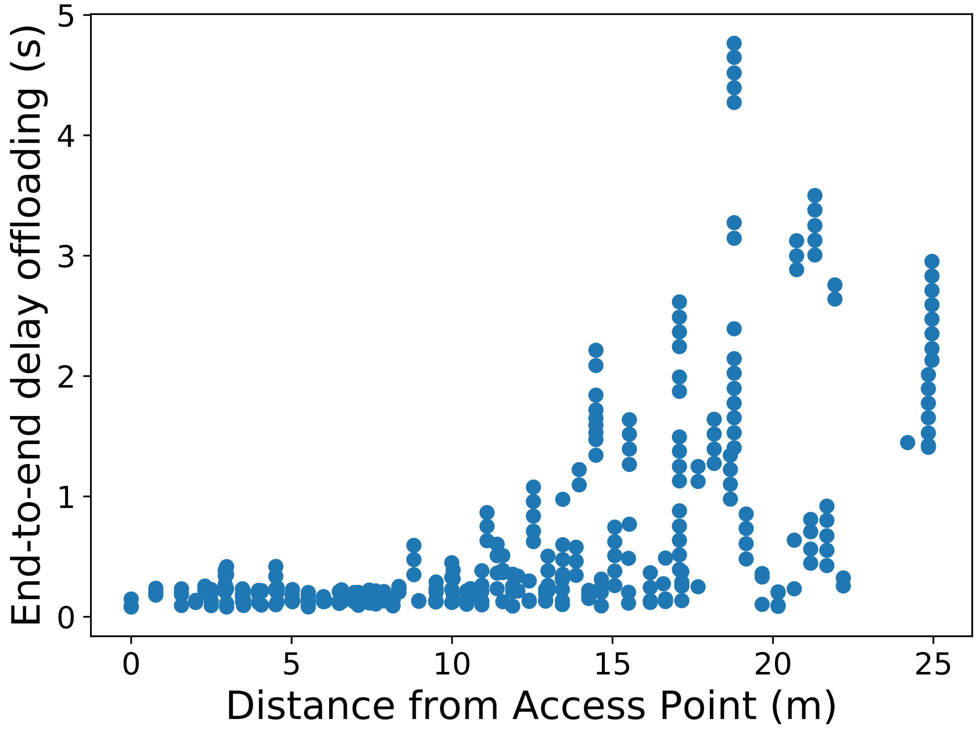


Figure 3.4: Capture-to-output time as a function of the distance between UAV and the edge server.

Therefore, offloading dramatically reduces energy intake, and would prolong mission time. However, the capture-to-control time of pipelines through available edge servers has additional components corresponding to the transfer of the image from the UAV to the server and of steering commands on the reverse path. Importantly, those components are heavily influenced by latent variables such as path loss and channel load, and present large fluctuations at fine-time scale due to fading, as well as channel access and transport protocols' parameters.

Fig. 3.3 shows an experimental trace obtained as the UAV is flying away from the edge server – a detailed description of the software and experimental parameters used to obtain these results is provided later. Micro and macro scale variations are observed on a general trend of degradation due to path loss. Distance spikes, automatically corrected, are due to wind. Fig. 3.4 reinforces the notion that distance and connection to edge servers are a rather poor predictor of the delay, with large variations and clustering effect.

As shown in the experimental results section, the onset of heavy-duty data streams has a more abrupt impact on the capture-to-delay time, whose trajectory presents sudden spikes and a more erratic behavior.

The need for a flexible and adaptive strategy is apparent. Importantly, the reaction of the system cannot be exclusively driven by macro-scale parameters such as distance due to sharp delay spikes triggered by micro-scale effects such as the dynamics of the inner state of communication and networking protocols. Moreover, hardly observable variables such as channel load and exogenous traffic emission have a considerable impact on the delay and its dynamics.

3.2 HYDRA

Herein, we describe the structure of Hydra and its embedded logics, which enables the activation/deactivation of pipelines, both local in-device and through edge servers.

3.2.1 Hydra Architecture

As shown earlier, autonomy pipelines are composed of three main logical blocks, namely sensing, data analysis and control, which transform environmental or internal signals into control. HYDRA allows the construction of flexible pipelines, where the flexibility is both in where the blocks are executed and which blocks are executed. To this aim, the open-source architecture we developed [8] is modular, where the module abstraction corresponds to an encapsulation of data transformation functions. The high level schematics of Hydra is depicted in Fig. 3.5.

The core of Hydra is a reliable threading of the modules over a distributed system, where the

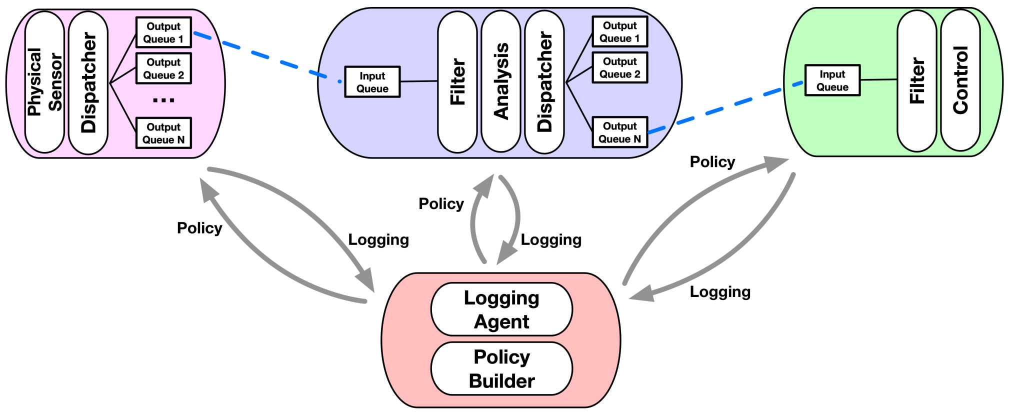


Figure 3.5: High-level schematics of the modular structure of Hydra.

threading itself is controlled by a logic. Every module is characterized by a core function, and is equipped with an input and one or many output queues. The queues are the interfaces between a specific module and all the other – local or remote – modules. Thus, the flow of information, as well as the deactivation of pipelines, is controlled by the routing strategy implemented by the modules. For instance, the deactivation of a pipeline will be realized by disabling an output queue, thus avoiding the summoning of the modules following it.

Some input queues implement filtering to remove replicas of data structures transiting the modules. In our implementation, the input queue of the actuation module at the UAV detects and filters out replica outputs of data analysis modules – that is, outputs corresponding to the same initial data – to avoid the implementation of duplicate steering actions. Moreover, some queues log the activity to monitor the performance of the associated pipelines. In our implementation, the logging driving pipeline selection is delegated to the input queue of the final actuation module at the UAV. The input queue of this module will log the capture-to-control time of active pipelines by tracking the delivery time of control outputs with respect to the generation time of their corresponding frame. Note that both times are generated at the same device.

A module outside those realizing the transformation of the data implements the high-level control logic of HYDRA. Specifically, we concentrate all intelligence in the *logics* module. The module collects the logs from target input queues and determines the routing policies of selected output queues. In this specific implementation, the logics module collects the temporal patterns of frame emission and output reception from the onboard sensing and control modules, respectively. The sequences of delays are, then, used to compute the average driving pipeline activation/deactivation and selection.

The logics module can also control data capture parameters to optimize the flow of information through the pipelines and maximize performance. Herein, we set the image capture rate to match the performance of the fastest pipeline over a moving window. Intuitively, an exceeding capture frequency will overload the active pipelines and create undesirable queueing effects, which may have a substantial impact on the capture-to-control delay. In the current implementation, we enforce a strict queueing policy, where we store only the most recent data structure received from the previous modules. This simple strategy avoids delay accumulation, and results in an "effective" capture rate where all dropped samples are not accounted for.

3.2.2 Hydra Logics

Intuitively, the macro-scale parameters governing temporally local capture-to-output delays are hardly observable. Even if some network interfaces can report the value of some relevant variables, such as channel gain and modulation, many others remain inaccessible unless a complex, and possibly resource consuming, information exchange is established. The most eminent examples include channel load and server load, which would require a direct exchange of information with local access points and available servers. Remarkably, even once these macro features of the environment are known, as shown in the previous section the capture-to-control time still presents complex patterns inducing possibly large performance variations.

We, then, take the simple but effective approach of using directly observable parameters to drive the system, where the selection process is guided by the observation of the actual capture-to-control time offered by available – active – pipelines. An important advantage of this strategy is that of being completely agnostic to the technologies and protocols used within the system, meaning that the state of all the pipelines is represented by a homogeneous set of variables. We remark that local computing at the UAV (local pipeline) is an option available to the UAV possibly with a reduced degree of uncertainty compared to offloading.

Although conceptually trivial, this approach presents an important challenge: only the capture-to-control delays associated with the currently used pipeline are observable. Intuitively, on the one hand, the activation of only one pipeline does not provide any information on other available pipelines. On the other hand, the activation of all the available pipelines – including the local one – would maximize the information available to the UAV, but maximizes the burden imposed to the surrounding networks and servers, possibly decreasing global performance. In fact, any active – non-local – pipeline uses the channel resource to support information exchange, and the server resource to complete the offloaded tasks.

Hydra takes this observation as a starting point to build an adaptive, and parsimonious, strategy for the exploration-exploitation of available resources. In Hydra, pipelines are, thus, activated to: (*i*) use the corresponding resources to generate control outputs; (*ii*) reduce uncertainty in the minimum capture-to-output time – that is, the first available outcome associated with an acquired sample; and (*iii*) update the state estimate of available pipelines. The key, then, is to control the activation of pipelines when necessary to one of these purposes, while minimizing the active time of pipelines whose output will not be used.

In order to control the activation of the pipelines, we define 3 operational modes, namely *Performance* ($\psi(t){=}P$), *Exploration* ($\psi(t){=}E$), and *Reliability* ($\psi(t){=}R$), where $\psi(t)$ is the mode at time t. In the *Performance* mode, the UAV utilizes only one, remote, pipeline, which is achieving the smallest known capture-to-output delay. In the *Exploration* mode, the

UAV activates available remote pipelines to update their known "state", and perform an informed selection. In *Reliability*, local computing is activated in addition to all the remote pipelines to guarantee an almost constant capture-to-control time when other options have degraded performance. Note that variations of these modes, where only subsets of pipelines are activated can be included in Hydra.

The selection of the mode is performed based on a recent window of capture-to-control times. Define $\tau_{p,j}$ as the capture-to-control time of image n processed through the pipeline p. At image N, the future mode and edge server selection are determined by the functions

$$E_p = \sum_{j=N-W+1}^{N} \alpha^{W-j} \tau_{(p,j)}, \quad M_p = \max_{j=N-W+1,\dots,N} \tau_{(p,j)}, \tag{3.1}$$

respectively corresponding to the moving average window and maximum of the last W images' capture-to-control delay.

Fig. 3.6 illustrates the modes and the selection strategy. Assuming pipeline p^* through an edge server is being currently used in *Performance* mode, the system switches to *Exploration* if M_p is larger than the threshold λ. In this mode, if for at least one of the pipelines M_p is below λ, then among those the pipeline with the smallest E_p is selected and the system returns to *Performance*. In *Exploration*, the system transitions to *Reliability* after δ samples unless at least one of the pipelines has delay below λ.

3.3 Experimental Results on Energy-Performance Trade-Off

We now presents and discuss experimental results illustrating relevant tradeoffs between accuracy and energy of available computing options. Experiments are performed placing

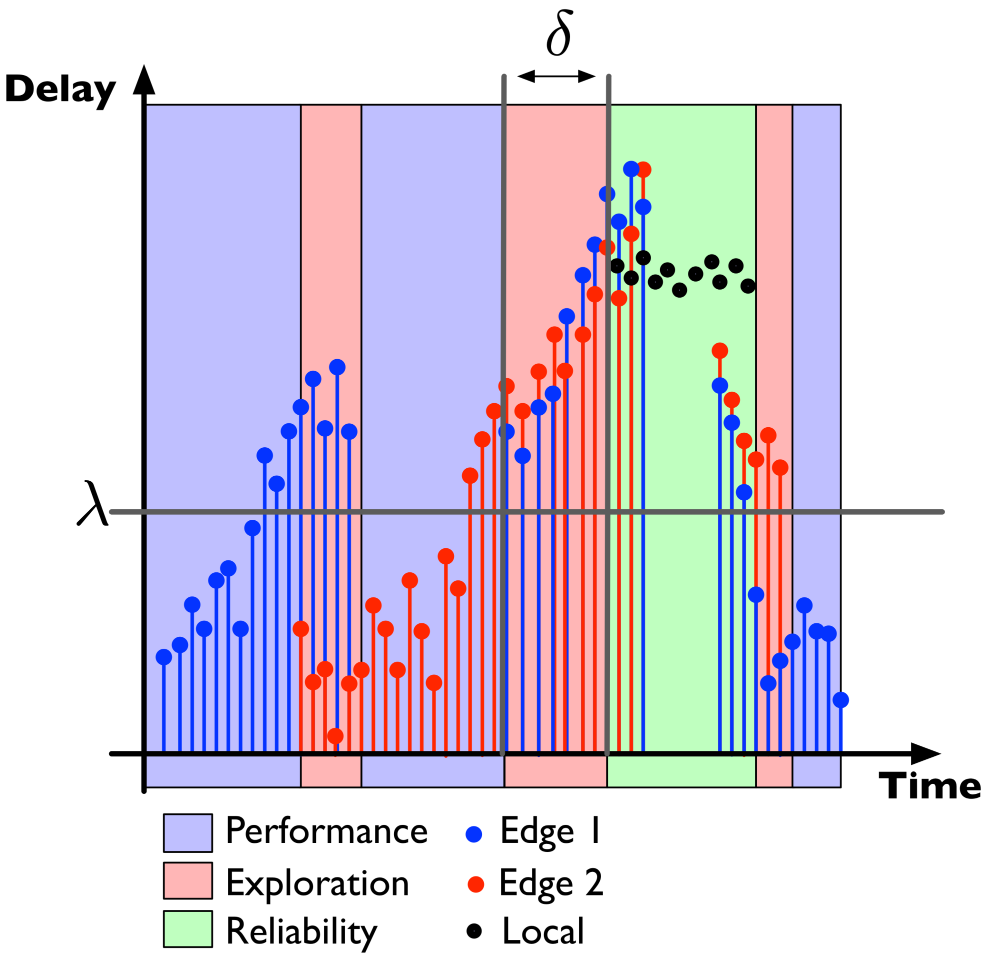


Figure 3.6: Illustration of the threshold based pipeline activation and selection strategy.

two edge servers at a distance of 25 m (Edge Server 1 and 2, respectively). A fixed image is used to provide a stable performance reference in all the experiments. The image is of size 480×640 RGB pixels and compressed using JPEG to 21 KB. The UAV is set to move for 10 minutes on the line between the two edge servers at a speed of 1 m/s to illustrate the impact of distance from the edge servers and guarantee reproducible experiments across our measurement campaign. Note that in the actual tracking application, variations in the capture-to-control time of different strategies could result in different motion trajectories of the UAV.

Fig. 3.7 depicts the average capture-to-control time and power consumption over the experiment as a function of the threshold λ. When λ is set to 0.1 s, Hydra almost exclusively

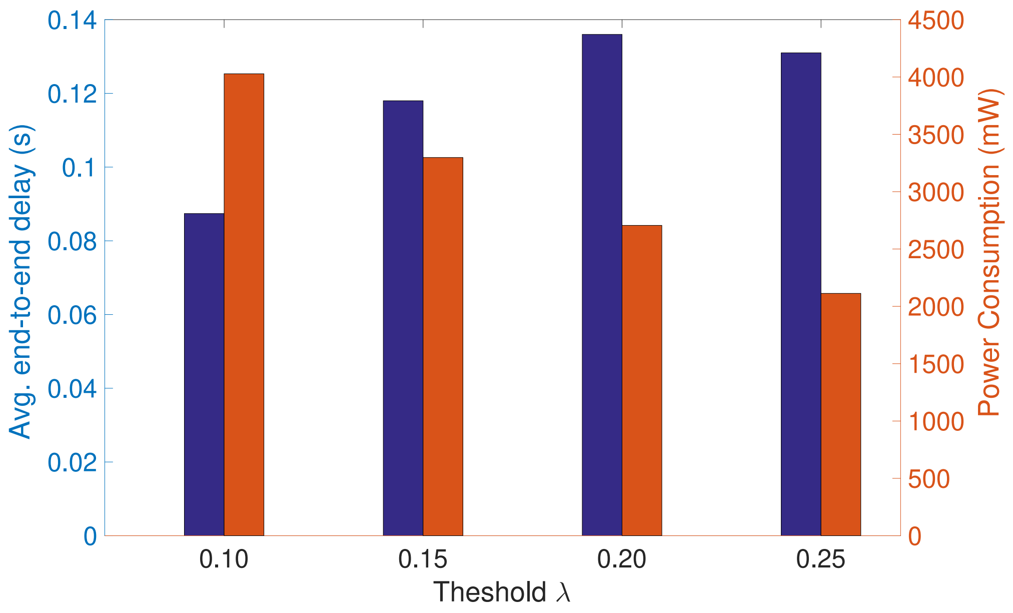


Figure 3.7: Avg. capture-to-control time and power consumption over the experiment as a function of the threshold λ.

chooses local computing over remote computing options. This results in a delay of about 0.09 s, and an energy consumption of approximately 4 W. As λ is increased, the system increases the fraction of time in which local computing is deactivated and one – or two – of the pipelines through the edge servers is kept active. At λ is set to 0.25 s, the delay increases to approximately 0.13 s, and the energy decreases to about 2 W. The trend is illustrated in Fig. 3.8, where the fraction of time in which local computing is active as a function of the threshold λ is shown.

The ability of the system to quickly adapt to delay variations is demonstrated in Fig. 3.9, which shows a temporal trace of the active pipelines. The UAV is at first connected to Edge Server 2. However, as it moves away from it, the delay degrades and Hydra transitions to *Reliability* mode, activating Edge Server 1. As the connection to Edge Server 1 is still flimsy, both edge-based pipelines have a large delay, and after δ samples local computing is activated. As the link to Edge 2 improves, windows of low delay allow the deactivation of local computing. We remark that when multiple pipelines are active, the first received control is used. Therefore, the smallest delay in the plot is the effective delay perceived by

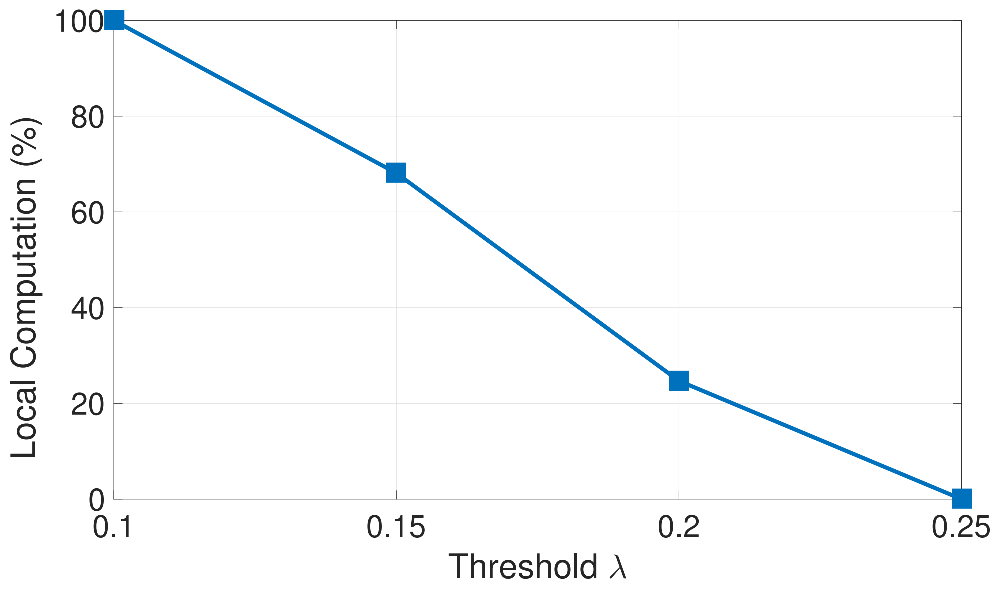


Figure 3.8: Fraction of time in which local computing is active as a function of the threshold λ.

the controller.

In order to further evaluate the system's dynamics, we inject in the channels used by Edge Server 1 and 2 traffic from external data streams. Specifically, we create high-volume traffic with a duty cycle of 20 s – 10 s active followed by 10 s inactive – with an offset of 5 s between the two channels. Thus, within one duty cycle, we have 5 s in which both channels experience congestion, 5 s in which both are congestion-free, and two periods of 5 s in which either one of the channel is congested and the other is congestion-free.

Fig. 3.10 shows a temporal trace of the capture-to-control time achieved by Hydra under these conditions. The cycles are apparent: the system switches from one edge server to another as the exogenous data streams are activated, and relies on local computing when the channel to both edge servers is congested. We note the erratic behavior of delay due to the interactions between data flows due to channel access and transport layer protocols.

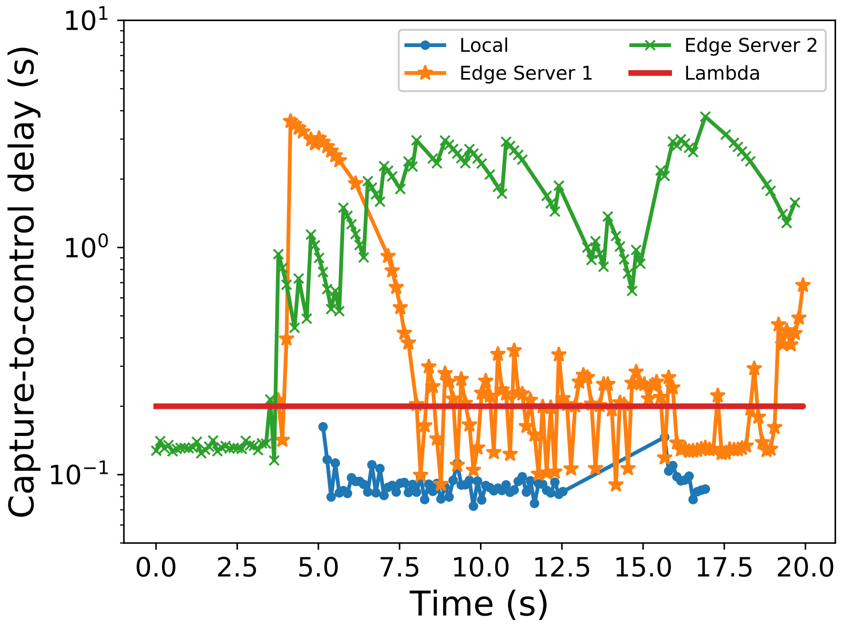


Figure 3.9: Temporal pattern of the capture-to-control time showing the switching between modes in Hydra.

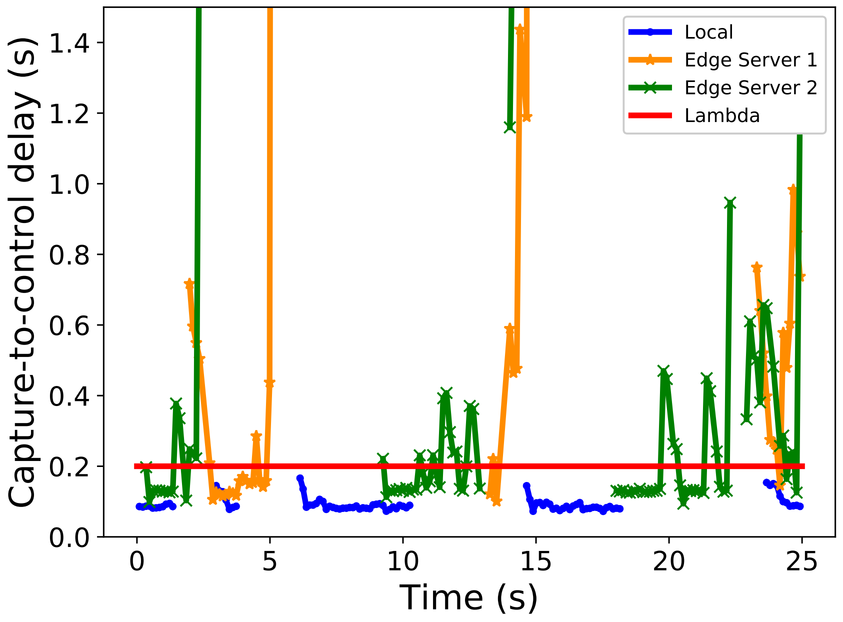


Figure 3.10: Temporal pattern of the capture-to-control time showing the switching between modes in Hydra.

Chapter 4

SeReMAS: Self-Resilient Task Allocation Mobile Autonomous Systems Through Predictive Computing

In this chapter, we tackle the challenging problem of providing *task-level* performance guarantees to a stream of computing tasks generated by an airborne MAS. Specifically, we impose a bound on the maximum time between data acquisition and the completion of the corresponding analysis task. We remark how a task-level perspective is necessary in this class of systems, where temporally local degradation of task delay can severely harm control loops in MASs. Figure 4.1 shows the temporal pattern of the end-to-end application delay (the different curves are different edge servers) obtained through our experimental drone testbed described in Section 4.4.1. We can observe that the delay exhibits significantly time-varying patterns, with a standard deviation 0.14 and a peak-to-peak difference reaching 0.43, which is 241% of the average value of 0.178s. We note that the experimental setting is in Line-of-Sight (LOS), and that more convoluted propagation environments would just aggravate this problem. A bound on the average delay would not guarantee that the task-level delay will be below a certain threshold for *each* of the tasks belonging to the task stream, which is key to guarantee correct functionality of stream-oriented edge-based MASs.

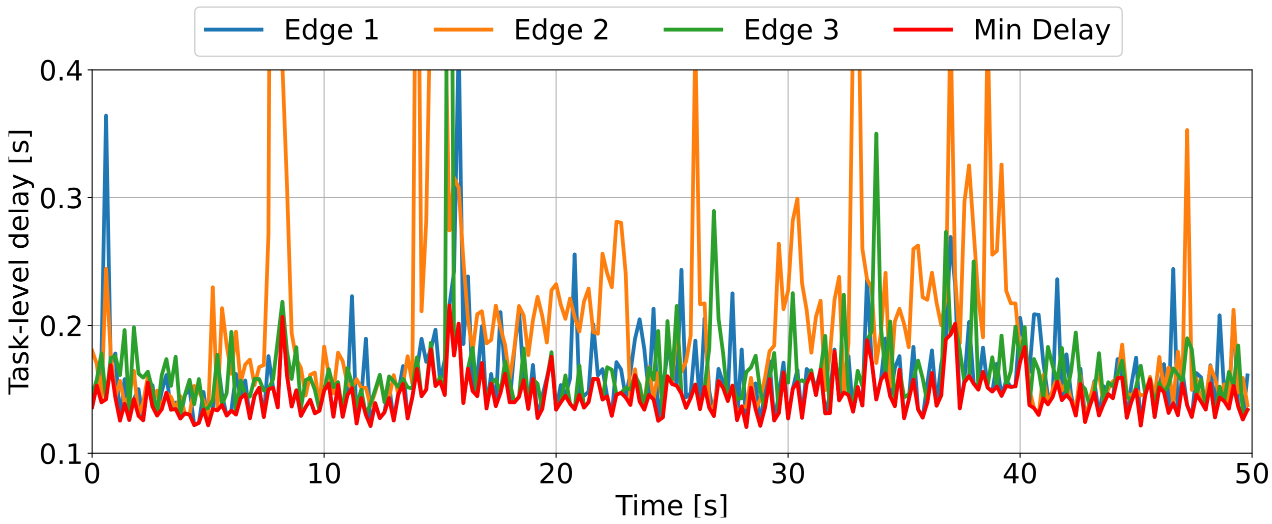


Figure 4.1: Example of task level delay from a flying drone to 3 edge servers, transmitted over WiFi 802.11n in a 50s interval.

Our vision is simple: *the seamless usage of edge resources by MASs necessarily requires techniques able to mitigate the impairments and erratic temporal patterns induced by the surrounding communication and computing ecosystems and the physics of the system itself.* Existing work – discussed in detail in Chapter 2 – has tackled the issue of MASs reliability in a piecemeal and often highly abstract fashion, by focusing on static optimization of either mobile device's trajectory [61, 18, 59, 25] or communication resources [57, 60, 6]. In Sec. 4.4.1, we show that edge selection methodologies based on channel quality would fail, and we conclude that new task offloading strategies are needed to stabilize task completion delay in MASs.

To address this challenging problem, we developed SeReMAS – Self-Resilient Mobile Autonomous Systems – a framework whose core is a dynamic task replication mechanism, where individual tasks are replicated and sent over multiple channel/edge server resources. The key intuition is that the task delay experienced by the MAS will be the minimum delay of each replica. Thus, the larger the number of channel/edge couples, the greater the probability that one task will satisfy the delay requirement, which however also implies increased resource usage. The objective of SeReMAS is to minimize resource usage under the constraint that the probability that the task-level delay bound will be met.

Our Approach. To drive our design, we implemented a testbed composed by an airborne MAS and multiple ground servers (Section 4.1). Specifically, we extracted a rich dataset from the system (Section 4.4.1), whose analysis demonstrates a lack of variables strongly correlated with the delay (Section 4.1.1). We show that the received signal strength indicator (RSSI), one of the key variables used to control connectivity and offloading, has limited influence on the delay. The dataset illustrates how in real-world MAS systems the delay pattern is the result of a wide variety of complex cross-variable interactions at various temporal scales. Importantly, influential variables are outside the network layers, and include *physical* variables such as orientation, acceleration and tilt.

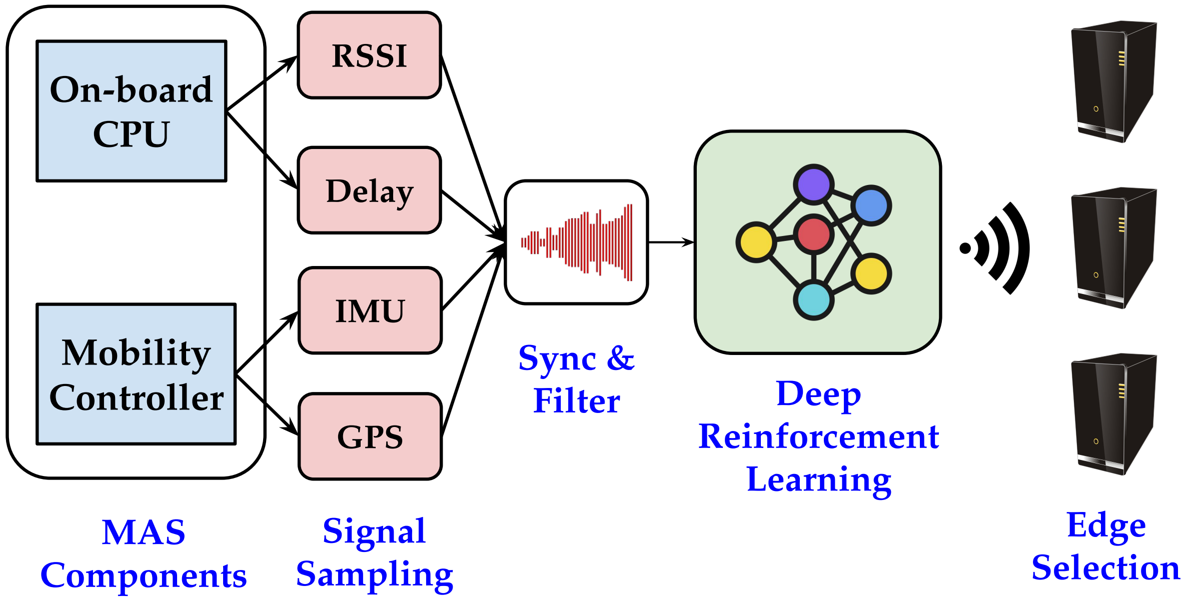


Figure 4.2: Our Architecture for Task Offloading in MASs.

Based on this considerations, SeReMAS embeds a *predictive* core based on Deep Reinforcement

Learning (DRL) to determine a compact set of computing pipelines dynamically assigned task-by-task based on the perceived state of the system. Fig. 4.2 depicts the high-level schematics of SeReMAS. The key intuition is that the selected set of channel/computing resources will influence future decision making, which DRL is able to capture. Some of the features – e.g., application and most network-related features – become available only if a resource is used. For instance, if a channel/edge server pipeline is not selected for a task, then the corresponding delay is not observed, which motivates the adoption of a DRL-based approach. By including future rewards in action selection and taking as input unprocessed features such as RSSI, end-to-end delay, inertial measurement unit (IMU) and global positioning system (GPS) coordinates, *the DRL algorithm will implicitly embed the impact of current computing pipelines selection on the efficacy of future decisions, as well as real-world phenomena that can be hardly modeled through explicit mathematical terms.*

Novel Contributions

• We design SeReMAS, a framework for the dynamic control of task offloading in MASs with extreme temporal variations (Section 4.2). SeReMAS is based on a preliminary experimental analysis (Section 4.1.1), which indicates that there is no dominant feature, including obvious features such as channel quality, and that prediction necessitates an ensemble of *weaker* features. We first mathematically formulate (Section 4.2.2) a Redundant Task Offloading Problem (RTOP). Then, we create predictors that can help managing the resource usage/performance trade-off. Specifically, we propose a myopic predictor as baseline (Section 4.2.3) and a DRL-based approach, which operates on a set of features from application, network and device-level components (Section 4.2.4). *To the best of our knowledge, SeReMAS is the first framework addressing the problem of redundant task offloading in MAS with a data-driven approach which efficacy is verified in a real-world testbed and with replicable dataset-based experiments.*

• We prototype SeReMAS on a drone-based experimental testbed (Section 4.3). The platform embeds a module for the real-time analysis of features, including the flight controller, tied to internal data routing control. As part of our prototype, we design a strategy to make the state representation compact (Section 4.3.2), and thus lower the complexity of the DRL agent, using an iterative feature selection procedure. We consider a real-time image analysis application through state-of-the-art edge-assisted object detection algorithms where a drone periodically acquires from onboard sensors data whose analysis is offloaded to edge servers on the ground (Section 4.4.1). We let the drone perform task offloading through multiple WiFi interfaces, and collect a total of 140 minutes of flying. *The dataset and the code produced as part of this chapter can be found at [9].*

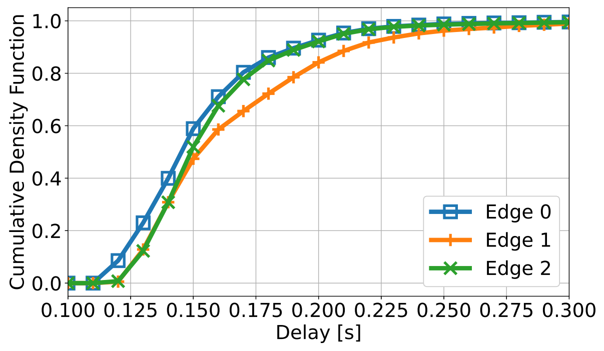
• Through experiments, we show how different subsets of features appear dominant at different time-scales (Section 4.4.2). We also show in Section 4.4.3 how the DRL approach improves by 17% the task execution probability with respect to a reactive approach [10], thanks to the ability to manage state uncertainty in the action selection problem, measured in terms of probability of meeting a delay requirement per amount of resource used, with respect to a myopic controller based on a one-shot selection of the next set of edge servers to be used.

4.1 Preliminary Experiments

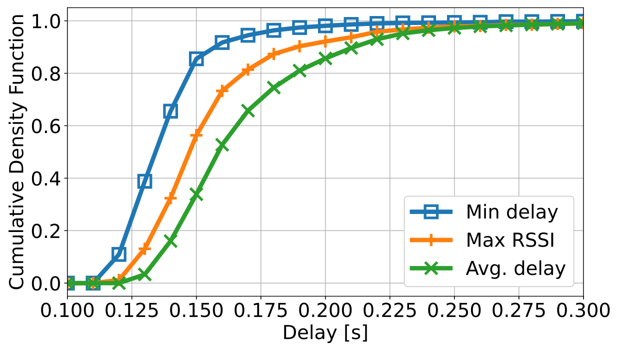
In our setting, a MAS is connected to N edge servers $\text{es}_1, \text{es}_2, \ldots, \text{es}_N$ through separate wireless channels. The device generates a sequence of tasks $t_1, t_2, t_3, \ldots$ with fixed inter-arrival time equal to T seconds. A task is described as a chunk of data to be processed with a predetermined analysis algorithm to produce an output. We assume that tasks are homogeneous, meaning that the amount of data associated with any task and the analysis algorithm are fixed. Let us define $\delta_n(t_i)$ as the capture-to-output delay of task t_i executed as edge server es_n, defined as the time from the generation of the task to the availability of its output at the edge server. The delay $\delta_n(t_i)$ is the composition of two delays: the transmission delay $\delta_n^{\text{comm}}(t_i)$ and the computing delay $\delta_n^{\text{comp}}(t_i)$. In real-world settings, both components are highly stochastic, and depend on a number of latent variable, parameters as well as states of protocols at various layers of the stack.

4.1.1 Preliminary Analysis

We motivate our study by analyzing the data obtained from real-world experiments. We consider an experimental setting, described in detail in Section 4.4.1, where a drone is offloading image processing tasks to three edge servers. Fig. 4.1 shows a section of the temporal pattern of the task-level delay δ_t at the three edge servers. We observe that the delay signals alternate low-delay ($150 - 175$ms) sections with spikes and higher delay sections. While some mild correlation between the delay signals is present, the minimum of the three signals provides the needed stability to the delay. Fig. 4.3.a shows the Cumulative Density Function (CDF) of the task-level delay δ_t for the three edge servers in our experiments. Note that in our scenario the task execution delay δ^{comp} is nearly deterministic. We remark that all the edge servers are within coverage, and that all the links are in Line of Sight (LoS). Most delays are in the range 120ms to 250ms, with about 40% of the delays below 135-145ms.



(a)



(b)

Figure 4.3: Cumulative density function of delay (a) for each edge server and (b) selecting the minimum delay, or the one with maximum RSSI, or the average of the available delays.

Fig. 4.3.b shows the distribution of the minimum delay $\delta_{\min}$ with respect to the cdf of the average delay and the delay associated with the edge server with the maximum channel quality index (RSSI). We observe that there is a noticeable difference between the minimum delay and the delay offered by the edge server with the best channel quality. Therefore, even a perfect SNR-based handover would fail to provide optimal performance in this context. This effect is the result of the convoluted interdependencies between protocol variables at the various layers and the physical and hardware properties of the system at multiple time scales.

We remark this important aspect by plotting in Fig. 4.4 the (delay, RSSI) and (delay, distance) mean and one standard deviation of the delay as a function of the two other variables. We can see the lack of a strong correlation between the delay and both RSSI and distance and emphasize again how experimental results unveil effects and interactions that are rarely captured in simulations and models.

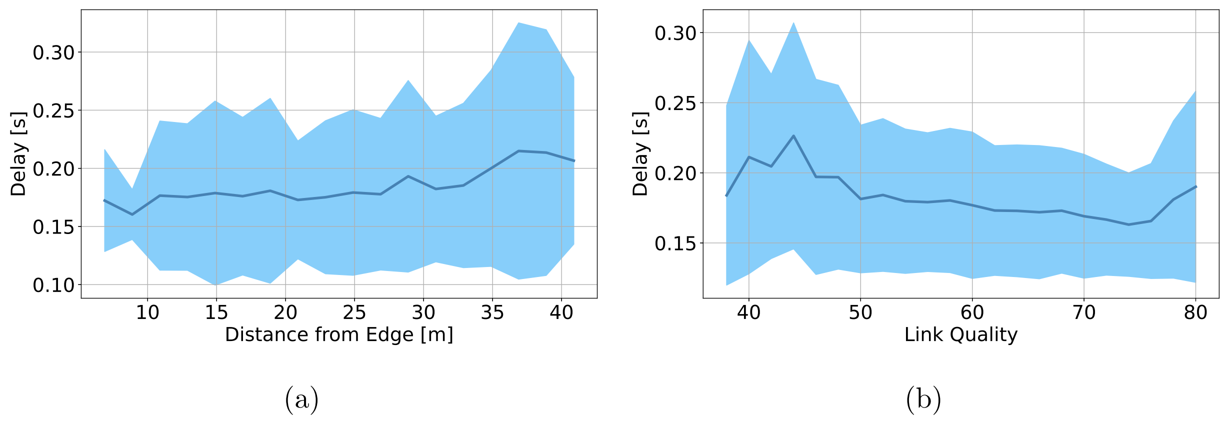


Figure 4.4: Distribution of task level delay as a function of distance from each of the edges and the RSSI.

4.2 The SeReMAS Framework

The results illustrated in the previous section emphasize the need for new techniques boosting the reliability of edge offloading for extreme real-time applications. In this section, we present

SeReMAS, a data-driven framework addressing the reliability of task offloading in MAS. We first present an overview of the main system blocks and functionalities in Section 4.2.1. Then, we formalize the learning-based redundant task offloading control problem in Section 4.2.2.

4.2.1 SeReMAS: A Walkthrough

SeReMAS [9] enables the data-driven control of task offloading from the MAS to the edge servers. The architecture of SeReMAS is depicted in Fig. 4.5, where we show the modules performing mobility control of the MAS (yellow) and control of task offloading (blue), and the modules – multiplexing and filter – handling the communication between the section of the platform at the MAS to the section at the various edge servers.

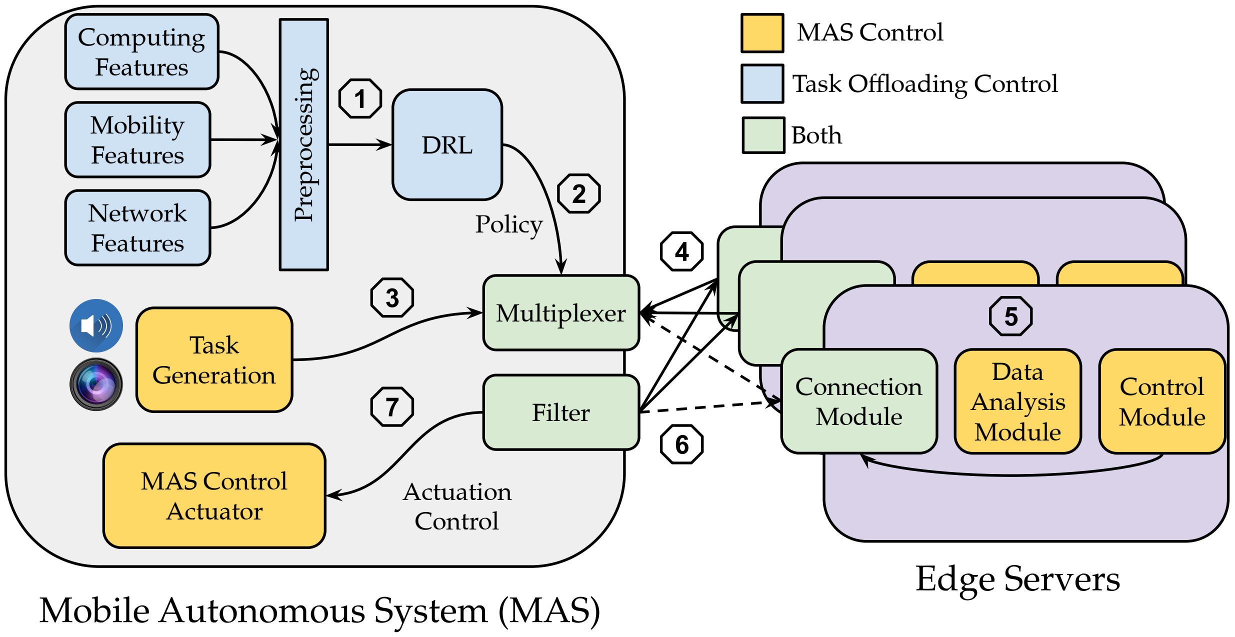


Figure 4.5: SeReMAS system architecture: two different control cycles intersect at the communication modules, where the DRL agent's policy is applied by means of task replication.

We now provide a walk-through of the main operations performed by SeReMAS, following the steps indicated in Fig. 4.5. First, the framework takes computing features (e.g., CPU, GPU, and RAM utilization), mobility features (e.g., accelerometer, gyroscope, GPS coordinates, etc), and network features (e.g., TCP state, RSSI) and applies pre-processing (step 1) to construct the input to a DRL model (see Section 4.2.2 for details). The extracted features and the composition of the state space are described in Section 4.3.2. Then, the DRL state is given as input to the DRL algorithm, which outputs ϕ, the set of edge servers to be used as task executors (step 2).

Tasks are generated (step 3) according to the current MAS needs (e.g., multimedia classification), and handled by module called *multiplexer* (step 4) which handles task replication across multiple edge servers. Specifically, the multiplexer is responsible for replicating and forwarding the tasks to the edge servers, and is directly controlled by the output ϕ of the DRL algorithm. The tasks are sent to the edges specified by ϕ, which are then executed (step 5). The knowledge produced by the task execution can be used to drive control decisions on the MAS. For example, in our prototype we use the task result to control the mobility of the MAS, as explained in Section 4.3. The related control messages generated by the edge server(s) are sent back to the MAS, and processed by the filter module (step 6), which eliminates replicated messages when more than one edge server is selected to avoid the re-execution of flight commands. Finally, the control messages are fed to the control actuator (step 7), which takes care of implementing the control action, if needed (e.g., flight control).

4.2.2 Redundant Task Offloading Problem (RTOP)

As part of the SeReMAS framework, we investigate the problem of redundant task offloading to replicate tasks and send them over multiple channel-edge server pipelines for increased reliability, which we call RTOP. This problem will drive our DRL design. We define the capture-to-output delay as the minimum of the delays associated with the task replicas:

$$\delta_{t_i}(\phi_i) = \min\{\delta_n(t_i) : n \in \phi_i\}, \tag{4.1}$$

where $\phi_i \subseteq \{1, \ldots, N\}$ is the subset of edge servers to which a replica of task t_i is sent. Then, we define a controller whose objective is to determine the sequence of edge servers $\boldsymbol{\phi}^* = [\phi^*_{t_1}, \phi^*_{t_2}, \ldots]$ solving the following optimization problem:

$$\arg\min_{\boldsymbol{\phi}} \quad E_i \left[|\phi_i| \right] \tag{4.2}$$

$$\text{s.t.} \quad E_i \left[I \left(\delta_{\min}(t_i) > \delta^* \right) | \phi_i \right] < \Delta, \tag{4.3}$$

where $I(\cdot)$ is the indicator function and expectation is computed over the task sequence. This formulation is different than imposing a constraint on the average delay, i.e., $E_i \left[\delta_{\min}(t_i) | \phi_i \right] < \delta^*$. The latter formulation would allow a possibly large number of delays above δ^*, while our formulation is equivalent to

$$\arg\min_{\boldsymbol{\phi}} \quad E \left[|\phi_t| \right] \tag{4.4}$$

$$\text{s.t.} \quad P \left(\delta_{\min}(t) > \delta^* | \phi_t \right) < \Delta. \tag{4.5}$$

Thus, we impose a constraint on the probability that the task completion time is above a threshold δ^* while striving to minimize resource usage.

Intuitively, the larger the number of edge servers selected, the larger the probability that the minimum of the delays is below the threshold. However, the inevitable limitations on channel access and maximum edge server load leads to a *task-level selection* problem, where the number and members of the chosen set is informed by the uncertainty regarding future delays and their expected values. In real-world settings, the resolution of the RTOP defined above necessitates the consideration of complex inter-variable and temporal interdependencies. For this reason, we resort to data-driven solutions methodologies decomposing the problem into sequences of local problems.

4.2.3 Myopic-based Baseline for RTOP

First, we formulate a myopic predictive solution to address the RTOP. We introduce the notion of state of the system $s_i = \{s_{i,n}\}_{n=1,\ldots,N}$, where $s_{i,n}$ is the feature matrix

$$s_{i,n} = \begin{bmatrix} \psi_{1,i-L+1,n} & \cdots & \psi_{1,i-1,n} & \psi_{1,i,n} \\ \psi_{2,i-L+1,n} & \cdots & \psi_{2,i-1,n} & \psi_{2,i,n} \\ \vdots & \vdots & \cdots & \vdots \\ \psi_{F,i-L+1,n} & \cdots & \psi_{F,i-1,n} & \psi_{F,i,n} \end{bmatrix}, \tag{4.6}$$

of $F \times L$ features, and $\psi_{f,j,n}$ is $f-th$ feature referring to task j and computing pipeline n. We describe the specific features and dataset in the Section 4.3.2. We train a probabilistic predictor as the function $p_{i+1,n} = \sigma(s_{i,n})$, where

$$p_{i+1,n} = P\left(\delta_n(t_{i+1}) > \delta^*\right). \tag{4.7}$$

We find the set ϕ_{i+1} with minimum cardinality such that

$$P\left(\delta_{t_{i+1}}(\phi_{i+1}) > \delta^*\right) < \Delta, \tag{4.8}$$

where the left-hand term is computed as

$$1 - \prod_{n \in \phi_{i+1}} (1 - p_{i+1,n}). \tag{4.9}$$

When more than one set with the same cardinality satisfies the constraint, then the one with the smaller probability is chosen. We note that stronger predictors $\sigma(\cdot)$ may lead to a reduced resource usage, as they would lead to reduced uncertainty in the class of the next delay (above and below threshold), and thus would allow the controller to bet on fewer remote computing pipelines. For example, let us assume that at least one of the pipelines has a next delay below threshold: an accurate and confident predictor returning probability 1 would allow the selection of only one edge server.

We extend the predictor to larger temporal windows to evaluate the predictive power of features blocks. We define

$$p_{i+1,n}^{W,y} = \sigma(s_{i,n}) \tag{4.10}$$

where

$$p_{i+1,n}^{W,y} = P\left(\sum_{\ell=0}^{W-1} I(\delta_n(t_{i+\ell}) > \delta^*) \geqslant y\right), \tag{4.11}$$

that is $p_{i+1,n}^{W,y}$ is the probability that at least y tasks will be completed with delay larger than δ^* in a window of W future tasks. We build a binary classifier from $\sigma(\cdot)$ by setting

$$p_{i+1,n}^{W,y} \underset{C_0}{\overset{C_1}{\gtrless}} \frac{1}{2} \tag{4.12}$$

4.2.4 Deep Q-Learning Approach for RTOP

The formulation above produces suboptimal control sequences. Thus, we adopt a Deep Q-Learning formulation to resolve the optimization problem. This formulation implicitly accounts for the impact of current decisions on the distribution of future states (and thus on the accuracy of control). In this case, the predictive function is defined to return the Q-values based on the state, that is,

$$Q(s_{i+1}, \phi_{i+1}) = \sigma_{DRL}(s_i), \tag{4.13}$$

where

$$Q(s_i, \phi_i) = E_{s_{i+1}|s_i,\phi_i}\left[E_{r_{i+1}|s_{i+1},\phi_i,s_i}\left[r_{i+1}|s_{i+1}, \phi_i, s_i|\right]\right]$$
$$+ \gamma \max_{\phi'} E_{s_{i+1}|s_i,\phi_i}\left[Q(s_{i+1}, \phi')\right]. \tag{4.14}$$

The cost variable c_i includes weighted penalties for the delay being above threshold and the cardinality of the selected set, that is

$$c_i = \lambda\ c_i^{\text{delay}} + (1{-}\lambda)\ c_i^{set}, \tag{4.15}$$

with

$$c_i^{\text{delay}}{=}I(\delta_{\min}(t_i) > \delta^*)S(\alpha^{\text{delay}}\delta_{\min} - \kappa^{\text{delay}}) \tag{4.16}$$

and

$$c_i^{\text{set}} = \alpha^{\text{set}}|\phi_i| - \kappa^{\text{set}}, \tag{4.17}$$

where α^{delay}, α^{set}, κ^{delay} and κ^{set} are normalization and offset parameters. $S(x){=}1/(1 + e^{-x})$ is the sigmoid function, here used to generate a smooth delay cost function which is 0 until δ^* and then progressively penalizing higher delay without overpenalizing tasks with poor channel conditions. Figure 4.6 shows the training procedure for our DRL-based approach.

We remark that the recursive formulation of the Q-values embeds the distribution of future states and costs given the current policy. The Q-values guide the selection of the actions according to the rule:

$$\phi_{i+1} = \begin{cases} \text{argmin}_{\phi_i} Q(s_i, \phi_i) & \text{with prob. } 1 - \epsilon_t \\ \mathcal{U}(\mathcal{P}(\phi)\backslash\varnothing) & \text{with prob. } \epsilon_t \end{cases} \tag{4.18}$$

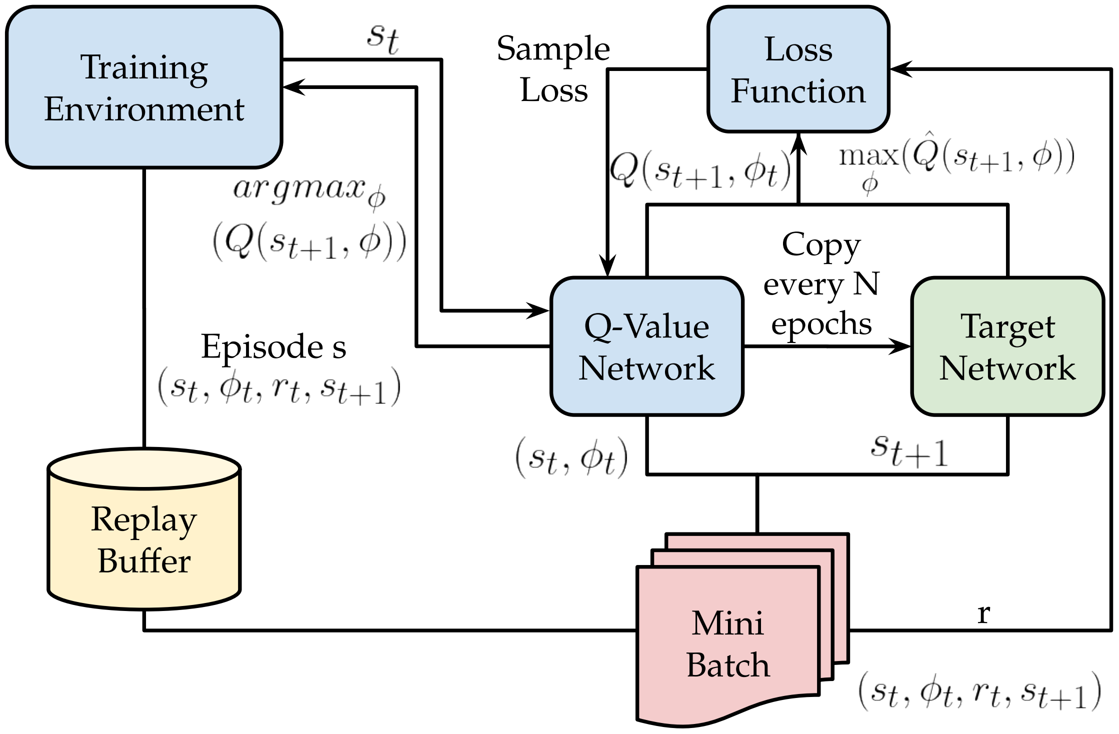
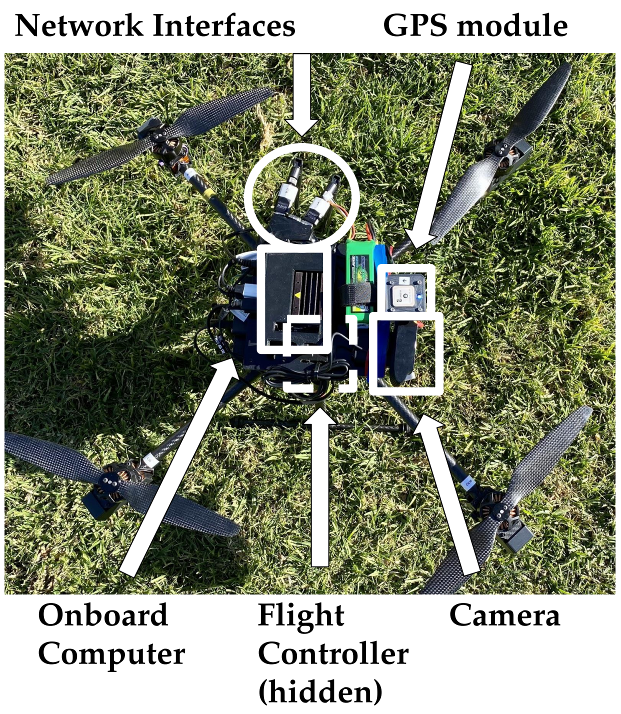


Figure 4.6: Training architecture using Double Deep Q-Learning.

where the best action (that is, subset of servers) is selected as the one maximizing the future reward with probability $1 - \epsilon$, and selected uniformly at random with probability ϵ. This is commonly known as a ϵ-*greedy* strategy, it is often used in practical problems to balance exploration/exploitation in DRL problems.

4.3 SeReMAS Prototype

We first describe the platform experimental components in Section 4.3.1, and then describe our feature selection process in Section 4.3.2. Finally, we explain how we implemented the SeReMAS predictors for the RTOP, both myopic and DRL, in Section 4.3.3.



(a)

(b)

Figure 4.7: (a) Drone prototype; (b) NVidia Jetson Xavier acting as edge server.

4.3.1 Platform Components

Figure 4.7 shows our experimental setup. Specifically, we use a Tarot650 quadcopter mounting a PixHawk flight controller. We connect Telem2 port on the PixHawk to a serial interface on a NVidia Jetson Nano board with 4GB of RAM. We use three NVidia Jetson Xavier development boards, operating in performance mode with 8 core ARM 64-bit processor, 32GB of main memory, 512-core Volta GPU. We use three IEEE 802.11n WiFi cards to interconnect the drone to the edge servers. These boards act as access points on different channels in the 2.4GHz WiFi spectrum.

4.3.2 DRL State Space and Feature Selection

We discuss how we create the input state for the DRL algorithm. We consider features at the application, network stack and device level as follows:

- **Application and Onboard Computer:** We track relevant application variables such as past capture-to-control delays, number of samples in the intermediate buffers, and selected actions. These will include real-time statistics relative to power consumption and resource allocation of CPU, GPU, and RAM.

- **Telemetry and Position:** We use MAVLink [32] protocol messages to register a listener to the flight controller. The onboard computer receives monitoring statistics from the Inertial Measurement Unit (IMU), Global Positioning System (GPS) and the power consumption of the vehicle. We include the edge servers' position, by including the distance from the drone using polar coordinates (Distance, Azimuth, Elevation) centered in the reference edge server. Distance is computed using the Harvesine formula. Moreover, we add the relative heading by computing the orientation of the drone with respect to the position of the edge server. Furthermore, we consider the $L2$-norm of multi-dimensional vectors (such as accelerometer and gyroscope data) and compute speed with respect to absolute reference frame and edge servers. All the features are synchronized at 5Hz.

- **Network:** We select relevant parameters such as TCP window and retransmissions, RSSI, and modulation/coding scheme (MCS) of the IEEE 802.11n protocol. We do so separately for each network interface available, so to isolate features relative to each edge server.

The details of the features are available in [9].

Feature Selection. We use feature importance methods such as Logistic Regression, Support Vector Machines and Random Forest as implemented in [45] and selected Logistic Regression with L1 regularizer due to the bias that Random Trees have towards features with high support's cardinality and the hybrid nature of the features, which include continuous and categorical variables. We then used a recursive algorithm, where at each iteration we train a predictor and discard the least influential features. We reduce the initial pool of 360 features to 73, maximizing accuracy on the validation set. Table I shows the normalized feature relevance predicting the number of high-delay tasks in a 1 s window.

Feature	Normalized Correlation
Round Trip Time average	1
Transmission timeout	-0.83
Packets Received	-0.80
Channel Level	-0.48
Inclination (magnitude)	-0.17
Position w.r.t Edge	0.16
Altitude	0.16
Last Sent	-0.15
Heading	0.13
Speed	0.08
Congestion Window	0.08

Table 4.1: Normalized feature relevance to a linear model predicting the number of high-delay tasks in a 1 s window.

Interestingly, while all available past delays are selected in the prediction (with $L = 3$ in Eq. 4.6), acceleration and inclination features are selected with a lag of 0.6s indicating a longer range dependency with the delay. Other relevant features include gyroscope and the increment of TCP fast retransmissions, failures, RSSI, and retries. The complete trend within the window is selected for these features. The selection shows how both vehicle and network parameters are relevant to characterize the state of the system and its future behavior, but their influence is expressed at different time scales.

4.3.3 Myopic Predictor and DRL Implementation

We provide the details of the myopic and DRL controllers.

Myopic Predictor - To implement the predictor $p^{W,y}_{i+1,n} = \sigma(s_{i,n})$ we train a series of dense DNNs (with two hidden layers at $[150, 50]$ nodes) using the Adam optimizer and trained for 100 epochs, with softmax output), which returns the probability that the next delay will belong to the predicted class.

Deep Q-Learning Agent - Naive implementations of Deep Q-Learning use one DNN function. However as demonstrated in [54], this approach may cause instability during training if the Q-values presents sudden changes. Due to the erratic behavior of the system we consider, we then take a Double Deep Q-Learning (DDQL) approach to build our DRL agent. In DDQL, two separate Deep Neural Networks (DNN) are used. Referring to Eq. (4.14), one network is trained to approximate $Q(\cdot) = Q(s_i, \phi_i)$, and the other one to approximate the future Q-value term in the expectation, that is $\hat{Q}(\cdot) = Q(s_{i+1}, \phi')$

Fig. 4.6 illustrates the DDQN architecture and the training procedure. We use a fully connected DNN, with $[200, 100, 50]$ hidden nodes, ReLu activation, and Huber Loss. During training, we apply backpropagation to $Q(\cdot)$ over the epochs $e = 1, ..., N$. We periodically copy DNN's parameters so that $\hat{Q} \leftarrow Q$, as a mean to reduce noise in during training. Note that we still choose the best action to learn on ϕ using the most updated $Q(\cdot)$, and in fact the decoupling between action selection and q-value function evaluation further stabilizes learning. We use a replay buffer during training, where the experiences in the form of $(s_i, \phi_i, r_i, s_{i+1})$ are stored and sampled randomly to avoid forgetting, which may occur if only the most recent experiences are used [31].

4.4 Experimental Results

We first present the experimental setting in Section 4.4.1, then the prediction performance in Section 4.4.2, and the task offloading results in Section 4.4.3.

4.4.1 Experimental setting

We consider a testbed illustrated in Fig. 4.8, which is composed of an airborne drone and N=3 ground edge servers in LOS. We consider an object tracking application where the MAS uses a camera to follow a predefined object at a certain distance. Specifically, the MAS captures images that are analyzed to extract the bounding box of the closest object of a certain class (*e.g.*, a person).

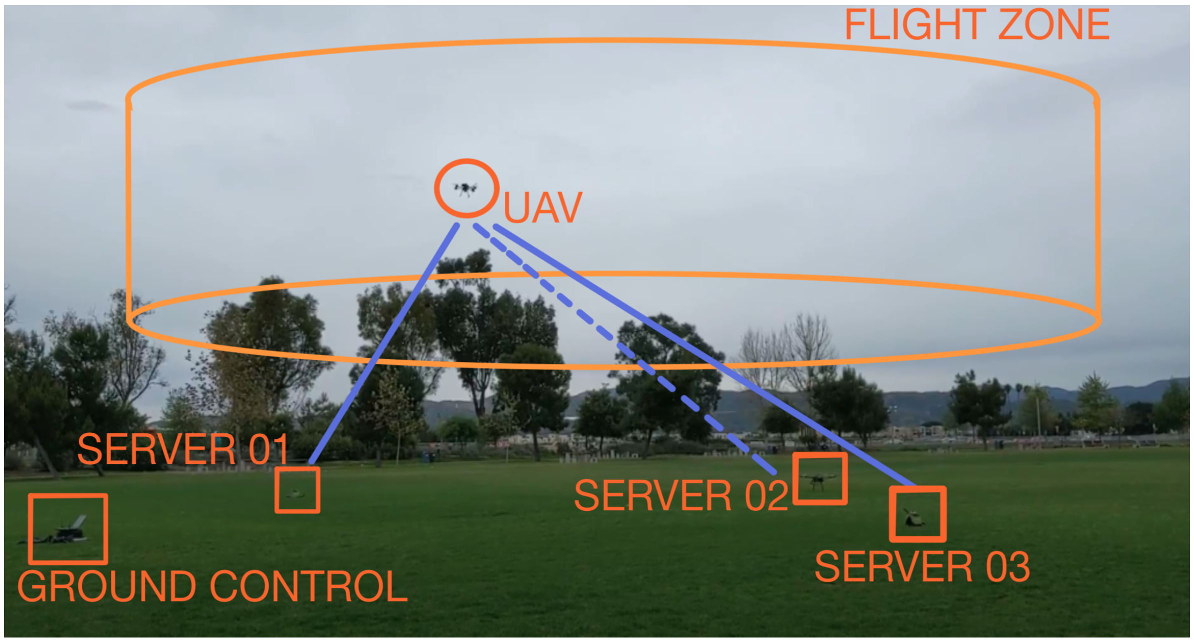


Figure 4.8: Schematic representation of the system setting: three ground edge servers, connected to the drone. Not all connections are continuously actively used (unused is dashed).

The controller steers the vehicle in the appropriate direction to (i) center the bounding box in the field of vision and (ii) obtain a bounding box of a predefined size by controlling the distance with respect to the object. In our testbed, the drone generates a regular stream of images to be analyzed using object detection. Specifically, the drone emits 15 images of size 19.5 kB per second. SSD-MobileNet-v2 model is used to analyze the images. In our measurements, the NVidia Jetson Xavier board takes 10 ms to execute the algorithm. Note that the onboard NVIDIA Jetson Nano takes 87 ms to complete the execution, however, power expenditure increases from 1.6 W to 4.2 W when the GPU is processing the images, that is, 11% of the power needed to fly.

To acquire a dataset for a wide-spectrum of flight parameters, we set the drone on a semi-random flight pattern around the edge servers. The pattern is defined by assigning uniformly distributed GPS way-points to the drone in a cylinder of radius equal to 30m centered on the edge server constellation and confining the altitude in the $[5, 15]m$ range. The maximum speed is randomly chosen for every new GPS waypoint between $[1, 4]m/s$. A new waypoint is set as soon as the drone reaches 3 meters from the current one, to obtain a smooth flight as similar as possible to a real application. In drone applications, the outcome of the object detection analysis is promptly needed to take control action and adjust the trajectory. While the action taken after the image analysis is beyond the scope of the current manuscript, we mention target tracking [42], object avoidance [55] as possible applications.

4.4.2 Prediction Performance

All results are based on an experimental dataset [9] collected using the randomized flight patterns described in Section 4.4.1. We first evaluate the prediction performance of the myopic predictors $p^{W,y}_{i+1,n} = \sigma(s_i)$ and associated binary classifier. In other words, the predictor determines whether at least half of the delay in the future window is below a given threshold, which we set to δ^*=175ms. We use the Area Under the Curve (AUC), integral of the ROC with respect to false positives, as performance metric, commonly used to evaluate algorithms predicting an imbalanced target.

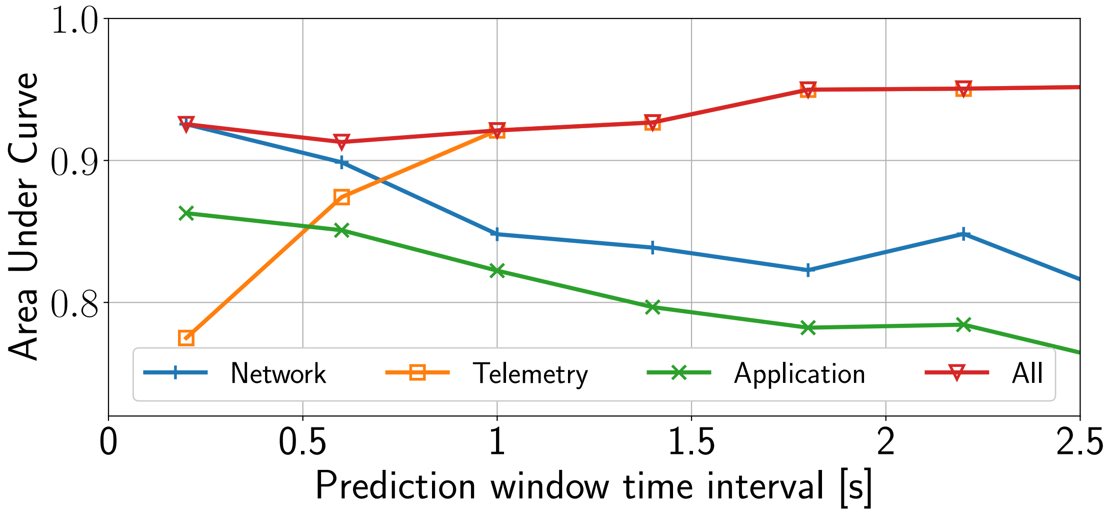


Figure 4.9: Performance of future delay classification for different sets of features. Length of the prediction window is expressed in seconds.

Fig. 4.9 shows the performance of the predictor trained on different feature blocks as a function of the window W (where we set $y = W/2$). The results highlight how semantic differences across subsets of features influence their predictive power in the short and long term. When the prediction window is small, most of the predictive power lies in networking features, which capture short-term correlations between high delay events. However, network variables struggle to capture longer-term trends, which are, instead predicted by telemetry variables. Indeed, the latter directly influence the distribution of fine-grain network events.

As noticed earlier, part of the network information is available only when offloading to a particular edge server. We now analyze how prediction performance is affected when several recent samples lack such information for one server. Fig. 4.10 shows how the lack of full state information (which is available only if the edge server is used) in recent samples (last one, last two, etc.) affects the ability of the myopic classifier to accurately predict future pipeline performance as a function of the prediction window W expressed in seconds. Missing information in one or few recent input samples, has a noticeable effect on classification in

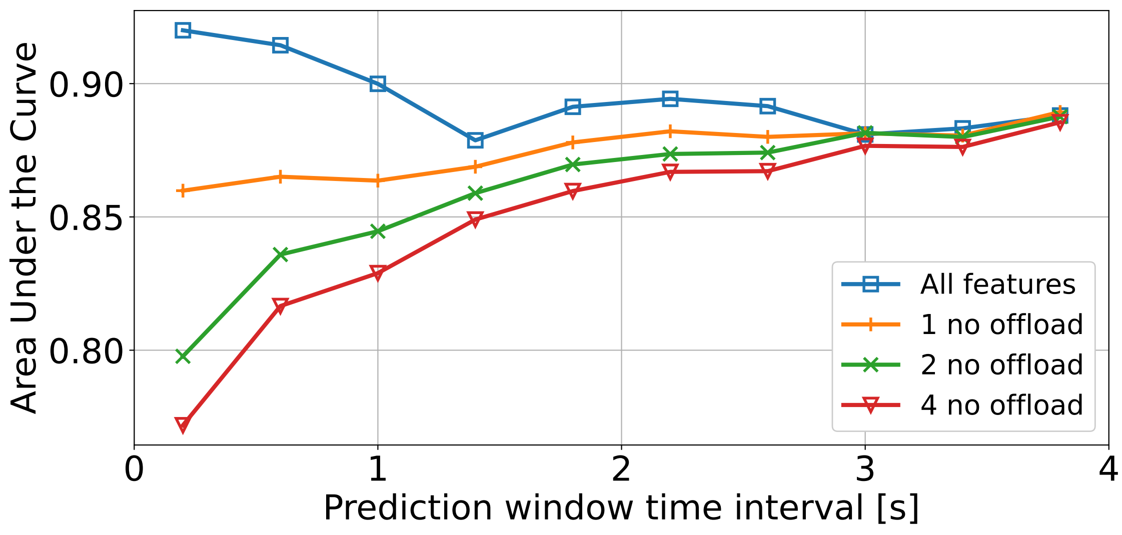


Figure 4.10: Performance in future delay classification in presence of partial information for recent time slots. Length of the prediction window is expressed in seconds.

the short term, as the AUC reduces by 5% for one sample and 10% for just two samples. On the other hand, as expected, the influence of recent samples fades out when predicting further points in the future. As the decisions of the DRL agent embed the future performance beyond the next delay sample, they also consider the availability of information in future decision instances.

4.4.3 Redundant Offloading

Fig. 4.11 shows the performance of the myopic and DRL selectors in terms of delay (percentage below threshold) and resource usage (average number of edge servers used). The different points for the myopic approach are obtained by varying the parameter Δ, i.e., the bound on the probability that the delay is below threshold.

The DRL approach, as described in Section 4.2.2, generates different points in the plot for different values of the weight λ in the cost function, where a larger λ favors low delay over resource usage. For comparison, we include a selector which uses all the available edge servers

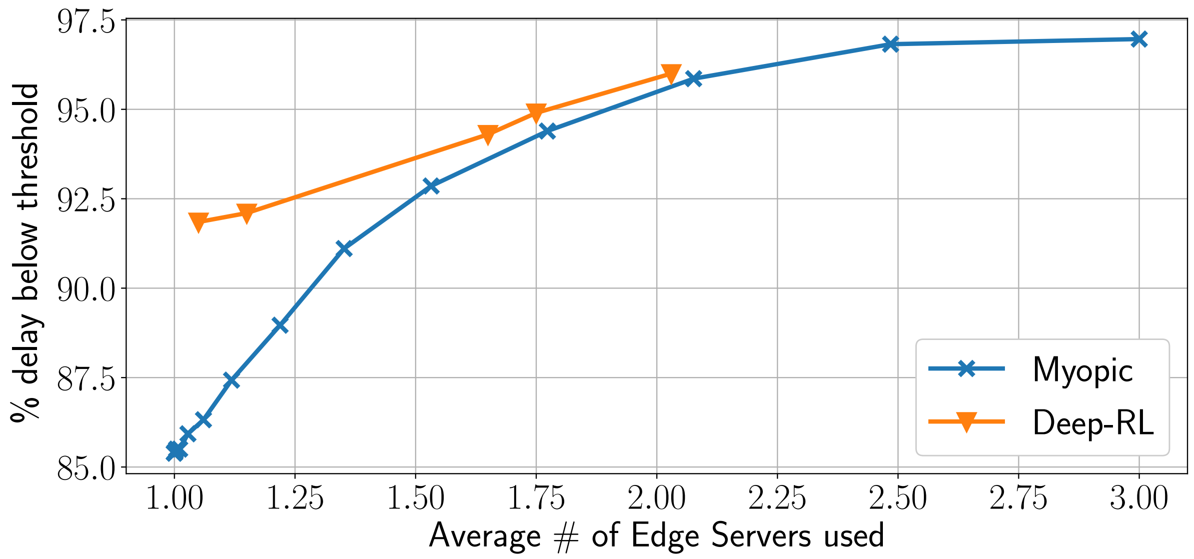


Figure 4.11: Delay performance and resource utilization trend of the myopic and DRL-based selector.

for all the tasks, and a selector which uses the edge server with the best channel quality index. When using all the three edge servers all the time, the myopic selector achieves maximum performance ($\sim$ 97%), whereas when using only one edge server, it achieves $\sim$ 85.5%. We note that a selector that chooses the edge server with the best channel quality achieves 75% of tasks with delays below threshold, w.r.t. which we improve 17%. Thus, *predictive control greatly improves performance compared to traditional options, even when idealized to task-level granularity without connection delay.* As we make the bound on Δ more tight, the myopic approach uses more and more resources.

We observe that using two edge servers, the myopic controller already achieves a performance roughly 2% worse than the three edge server option, demonstrating that prediction can reduce resource usage. However, when using a small amount of resources, the myopic controller's effectiveness sharply decreases. Conversely, the DRL is capable of effectively select small sets of computing pipelines while preserving delay performance. Using 1.1 edge servers on average, the DRL approach achieves $\sim$ 92%, that is, 7% more than the myopic approach. We explain this trend by observing that the DRL agent optimizes the information available to make future decisions, thus maximizing the overall prediction accuracy when resources are scarce and selection needs to be precise.

To further illustrate the behavior of the proposed approach, we show in Figure 4.12 a time series of delays and decisions (selected edge servers) of the DRL-based approach for two different λ (0.1 and 0.2) used in Eq. 4.15. We can see that the DRL agent can stabilize delay, where a larger use of resources leads to the avoidance of more delay peaks. We note how the DRL agent rotates the edge servers periodically to harvest information for more informed future decisions.

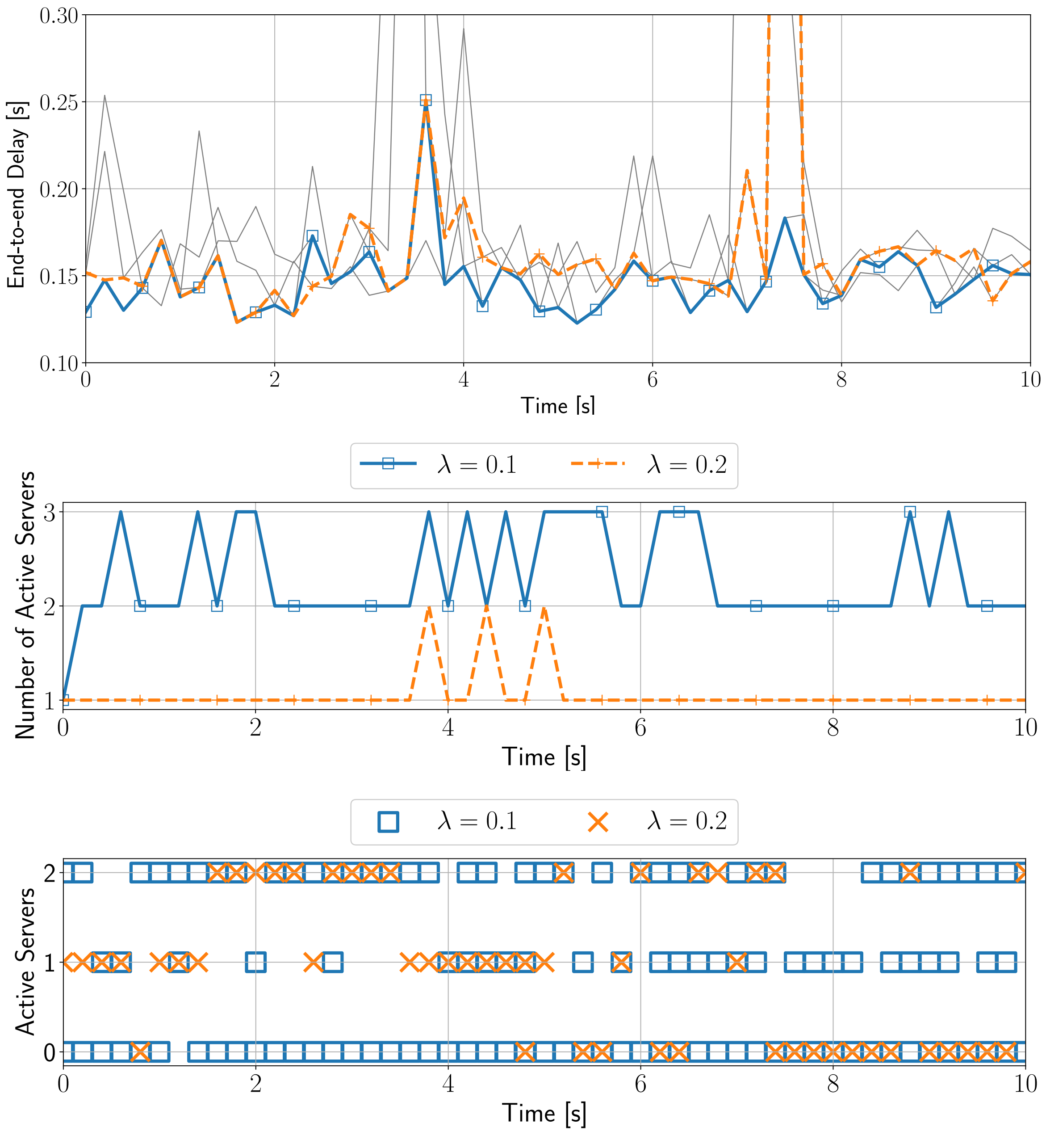


Figure 4.12: DRL agent improving delay by using task replication. We plot in grey the traces of the non-selected delays.

Part II

Theoretical Analysis of Task Offloading for MASs

Chapter 5

Optimal Edge Computing for Infrastructure-Assisted UAV Systems

In this chapter, we present an optimized decision process through which the UAV decides whether to process locally or offload the computation task to the edge server. The decision is based on a series of interactions between the UAV and the IoT system, where the UAV receives feedback on the state of the network and edge server, which allows the estimation of the residual time to task completion. Based on this information, the UAV solves an optimization problem aiming at the minimization of a weighted sum of delay and energy expense. Formally, the problem is formulated as an Optimal Stopping Time problem over a semi-Markov process.

Numerical results, which are based on parameters extracted from a real-world implementation of the system, demonstrate that the proposed intelligent and sequential probing technique effectively adapts the processing strategy to the instantaneous state of the network-edge server system. The outcome is a reduced processing delay and energy expense, two extremely important metrics in the considered application. These results are evaluated on the aforementioned urban scenario, where we place particular emphasis on the components that could decrease the performance of the UAV across a mission. In addition to analytical evaluations, we characterize the, temporal and average, performance of the adaptive offloading scheme we proposed in a scenario where a UAV mission requires to complete a trajectory around a building while analyzing images.

In the setting described above, we train a Deep Reinforcement Learning (DRL) agent capable of learning spatio-temporal characteristics of the environment to learn effective offloading policies. Results indicate the importance of features such as position and recent offloading outcomes in maximizing the ability of the controller to optimize its decisions across missions.

The rest of the chapter is organized as follows. Section 5.1 provides an overview of the system considered in this chapter. Section 5.2 describes in detail the parameters and operations of the UAV-edge server system. Section 5.3 introduces a Markovian description of the system's dynamics and formulates and solves the problem for the optimization of the offloading decisions. Section 5.4 presents numerical results illustrating the performance of the proposed adaptive offloading strategy, and comparing it with alternative solutions.

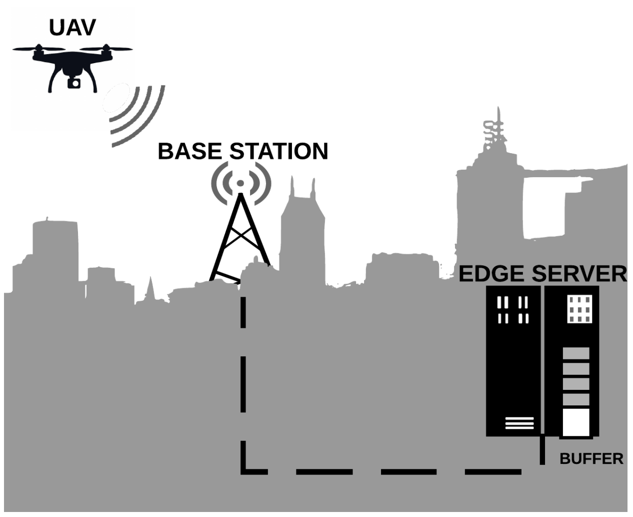


Figure 5.1: Illustration of the considered scenario and system: a UAV interconnects with an edge server through a low latency wireless link to offload computation tasks. Poor channel conditions and high processing load at the edge server may result in a larger delay and energy expense compared to local on-board processing.

5.1 System and Problem Overview

We consider a scenario where a UAV autonomously navigates an urban environment. The UAV is assigned the task to acquire and process complex data in predefined locations within the city, where the outcome of processing may influence sensing and navigation actions. A relevant case-study application is city-monitoring, in which the UAV captures a panoramic sequence of pictures at each location and process them using a classification algorithm to detect objects or situations of interest. In case of positive detection, the UAV may stay at the location to capture more detailed or higher-resolution pictures of a specific portion of its view.

Intuitively, processing information-rich signals using a computation-intense algorithm is a challenging task for inherently constrained platforms such as UAVs. In fact, the limited processing power of the on-board computation resources results in long capture-to-output time of the algorithm, which decreases responsiveness to stimuli and increases mission time. Additionally, on-board processing consumes a significant amount of energy, even when compared to motion and navigation, thus shortening the lifetime of these battery-powered systems. Note that in the scenario described above, the UAV is hovering while waiting for the classification algorithm to complete, as the outcome will determine its subsequent action. Thus, a large processing time incurs at additional energy expense penalty associated with longer flight time.

The UAV can leverage the resources of the surrounding urban IoT infrastructure to improve its performance. In the scenario at hand, edge servers placed at the network edge can take over the task of processing the data acquired by the UAV. Intuitively, the larger processing power of edge servers compared to that of UAVs grants a much faster completion of the processing task, thus allowing a faster decision making and a smaller capture-to-decision time, defined as the time between image capture and the availability of the its analysis' outcome.

Additionally, the UAV would be relieved from the energy expense burden of processing, at the price of energy expense associated with data transmission. We remark that a shorter time to receive the output of the classification algorithm also corresponds to a smaller energy expense associated with hovering.

However, as noted in the introduction, the urban IoT is a highly dynamic environment, where a myriad of data streams and services coexist and compete for the same communication resources. In the considered scenario, the wireless channel connecting the UAV to a wireless access point may have a low capacity due to the physical properties of signal propagation, but also due to the existence of interfering communications which use part of the time/frequency channel resource. Additionally, the edge servers may be serving other devices offloading their computation tasks, and the UAV task may suffer queuing delay, or a reduced processing speed. As a result, in certain conditions, offloading the computation task to an overloaded edge server connected to the UAV through a poor communication channel may lead to a longer capture-to-decision time. Again, this corresponds to less efficient mission operations, but also a possibly large energy expense due to hovering while waiting for a response.

In order to fully harness the possible performance gain granted by the available resources provided by the urban IoT infrastructure, the UAV needs to make informed decisions about whether or not to offload the execution of image analysis. To this aim, we equip the UAV with the ability to interact with the surrounding network and edge devices and acquire information regarding the status of the communication and processing pipeline. The information is used to evaluate the progress of the task and predict the future cost of the binary decision between local and edge-assisted computing.

We remark that the optimization framework can be used in scenarios with multiple base stations and edge servers. As the sequential decisions are made on a task by task basis, handover and connectivity can be managed by the network infrastructure without a major impact on decision making. However, we note that when considering agents learning spatio-temporal correlation properties (as those shown in Section V.C), the agent will need to implicitly incorporate connectivity information associated with spatial features. In fact, handover may cause abrupt loss of temporal correlation – for instance in server load if a different edge server is connected to the new base station.

5.2 System Model

In this section, we formalize and discuss an abstraction of the system composed of the UAV, a network access point and an edge server. We divide the description into modules focusing on the communication, computation and energy expense aspects of the system.

5.2.1 Communications

The UAV is connected to the network access point through a wireless channel of finite capacity. The data to be transferred for offloading have size L-bits. The UAV transmits with fixed power P and rate R in the finite set of $K+1$ transmission rates $\{R_0, R_1, R_2, \ldots, R_K\}$, where R_0=0 corresponds to disconnection from the network, and thus no data transmission. The link between the UAV and the AP is a wireless link affected by path loss, fading and noise. The SNR at the receiver is

$$\mathrm{SNR} = \frac{gP}{\sigma^2}, \tag{5.1}$$

where σ^2 is the noise power and g is the channel attenuation coefficient including path loss and fading. We assume exponential path loss and Rayleigh flat fading. Thus, the distribution of g is

$$\Theta_g(x) = \Pr(g \leqslant x) = 1 - e^{-\frac{x}{\gamma}}, \tag{5.2}$$

where γ is the path loss.

Assuming channel knowledge and a capacity achieving scheme, the selected transmission rate of the UAV is equal to R_i bits/s if $g \in (g_i, g_{i+1}]$, where

$$g_i = g : R_i = \mathcal{C}(g\text{SNR}), \quad i{=}1, \ldots, K, \tag{5.3}$$

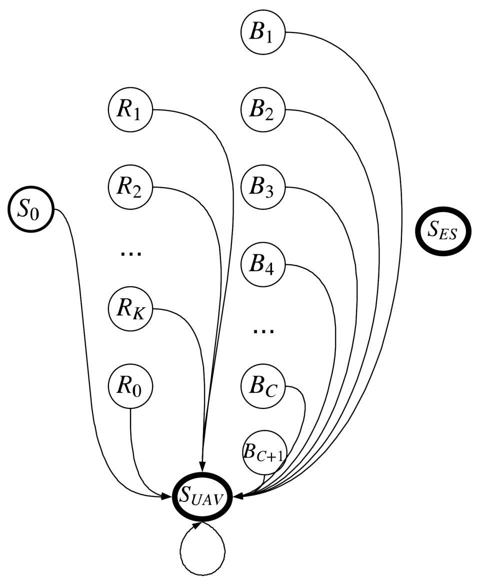
and

$$\mathcal{C}(x){=}\log(1+x). \tag{5.4}$$

The resulting transmission time is L/R_i seconds. In the chapter, we use a capacity model to abstract the communication layer, where the channel gain is matched with a maximum achievable data rate. The integration in the model of more realistic communication models, for instance to capture interactions between physical, channel access and transport layers, would lead to a much more convoluted analysis. We point the interested reader to our work [10] for the evaluation and analysis of real-world edge computing for UAVs with dynamic offloading.

5.2.2 Computation

The time to complete the computation task locally at the UAV and at the edge server are captured using the random variables X' and X, respectively. We assume that X' and X follow an exponential distribution of rate μ' and μ tasks/s, respectively. The edge server accumulates incoming computation tasks in a finite buffer of size B tasks. Excluding the task generated by the UAV, tasks arrive according to a Poisson process of rate λ tasks/s, with $\lambda{<}\mu$.



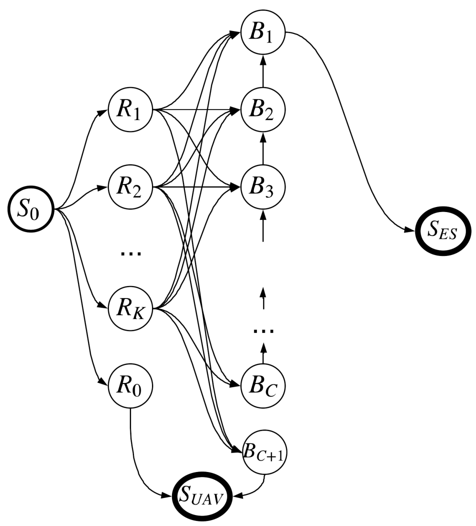


Figure 5.2: Representation of state transitions with non-zero probability in the Markov Chains associated with decision $u = 0$ (left) and $u = 1$ (right).

5.2.3 Energy

As described in the previous section, at each predefined location the UAV captures the data, and then completes the computation task – either locally or at the edge server – while hovering maintaining the position. We define a rate of energy expense for the two fundamental operational blocks that are influenced by the offloading decision: processing and hovering. Specifically, we define P_{P} and P_{H} as the Watts used to respectively process the data and hover. As mentioned earlier, the transmission power is equal to P Watts.

5.3 Optimal Offloading Decisions

In the considered scenario, the two most relevant performance metrics are energy expense E and time T per location. Herein, we assume the state of the system at each location to be independent. Importantly, the costs E and T are a function of the offloading decision, that is, whether the computation task is completed at the UAV or at the edge server.

Given the knowledge of the system parameters, the UAV can compute the average cost E and time T corresponding to each of the two options, where the average is over realizations of the stochastic process associated with the system dynamics. However, within that average there are realizations in which offloading is advantageous (high channel capacity and low processing congestion) or disadvantageous (low channel capacity and high processing congestion). In order to fully harness the performance gain edge computing can offer, while facing the dynamics of the IoT system, we develop a sequential probing and decision making framework. At each stage, the UAV observes the current realization, estimates the residual cost to complete the task, and makes a decision about whether to initiate local processing or not. This formulation corresponds to an optimal stopping time problem on a semi-Markov process.

Under the assumptions listed in the previous section, the temporal evolution of the system can be represented as a semi-Markov process. Let's define as t_j^+, $j{=}0, 1, 2, \ldots$ the time instants right after the occurrence of an event, defined as the establishment of the connection with the network, the delivery of the data to the edge server, or the completion of a computation task at the UAV or edge server. We denote the state of the system at time t_j^+ as the random variable $S(t_j^+)$. The state space $\mathcal{S}$ of $S(t_j^+)$ consists of an initial state s_0, two termination states s_{UAV} and s_{ES}, and a number of states describing data transmission and task queueing process. The termination states correspond to the computation task being completed locally at the UAV (s_{UAV}) and offloaded to the edge server (s_{ES}). Specifically, we include *(i)* a set of $K{+}1$ states $R_0, R_1, \ldots, R_K$ associated with a transmission rate, that is, a channel state in the ranges defined in the previous section; and *(ii)* a set of $C{+}1$ states $B_1, \ldots, B_{C+1}$ associated with the position of the UAV task in the task buffer at the edge server. Note that B_{C+1} corresponds to a full buffer at arrival, that is, the UAV task is rejected. It can be shown that the process $\mathbf{S}{=}(S(t_j^+))_{j=0,1,\ldots}$ is a Markov process.

At each time instant t_j^+, the UAV is notified of the state $S(t_j^+)$ from the network access point or the edge server, and makes a binary decision $u\in\{0,1\}$, where 0 and 1 correspond to local computing and continuing on the edge-assisted pipeline – that is, further deferring local computing, respectively.

5.3.1 Transition Probabilities

We now describe the transition probabilities governing the dynamics of the stochastic process $\mathbf{S}$. For the sake of notation clearness, we denote the time t_j^+ with its index j. We define, then

$$P(s'|s,u)= \Pr(S(j+1)=s'|S(j)=s, U(j)=u). \tag{5.5}$$

If the decision is equal to 0, the transition probabilities from any state s are

$$P(s'|s,0)= \begin{cases} 1 \quad \text{if} \quad s'=s_{\text{UAV}}; \\ 0 \quad \text{otherwise.} \end{cases} \tag{5.6}$$

That is, if the decision is to compute locally, the process moves to state s_{UAV} deterministically from any state.

We, then, analyze the transition probabilities if the decision is 1, that is, the UAV further defers the initiation of local computation. In such case, from the initial state s_0, the channel distribution is sampled, and the state moves to one of the pre-transmission states R_i with probability equal to that of the associated interval. Thus,

$$P(s'|s_0,1)= \begin{cases} \pi_i \quad \text{if} \quad s'=R_i, \; i=0,1,\dots,K; \\ 0 \quad \text{otherwise,} \end{cases} \tag{5.7}$$

where $\pi(i){=}\Theta_g(g_{i+1}) - \Theta_g(g_i)$.

In any state R_i, the UAV is reported the transmission rate, that is, the index i, from the wireless access point. If the decision is to defer local processing, the transition probabilities from R_i, i=1,...,K, are

$$P(s'|R_i,1){=}\begin{cases}\sigma_{c-1} & \text{if } s'{=}B_c\\ 0 & \text{otherwise.}\end{cases} \tag{5.8}$$

σ_c is the probability that the UAV task will find c tasks stored in the edge server buffer at arrival. It is known that

$$\sigma_c{=}\frac{(1-\lambda/\mu)(\lambda/\mu)^c}{1-(\lambda/\mu)^{C+1}}. \tag{5.9}$$

The state R_0, corresponding to disconnection from the network, deterministically leads to s_{UAV}.

At the beginning of any state B_c, the UAV is notified of the index c. For states B_c, $c{=}2,\dots,C$, the transition probabilities are

$$P(s'|B_c,1){=}\begin{cases}1 & \text{if } s'{=}B_{c-1}\\ 0 & \text{otherwise.}\end{cases} \tag{5.10}$$

State B_{C+1} corresponds to a full task queue and, thus, rejection of the UAV task. Therefore, from B_{C+1} the system deterministically moves to s_{UAV}. State B_1 corresponds to the UAV task being in the first position, and deterministically leads to s_{ES}.

5.3.2 Cost Functions and Optimal Policy

With the transition probabilities conditioned on the state and action, we can now build the optimization process. We consider a formulation where the objective of the UAV is to minimize $E(V)$, with

$$V = \omega E + (1{-}\omega)T, \tag{5.11}$$

where ω is a positive weight in $[0, 1]$.

To this aim, define the time and energy spent in state $s{\in}\mathcal{S}$ as $\Phi(s, u)$ and $\Psi(s, u)$ conditioned on the action u, respectively. Note that both the latter and the former are random variables. We denote their average as $\phi(s, u){=}E(\Phi(s, u))$ and $\psi(s, u){=}E(\Psi(s, u))$. We further define $C(s, u){=}\omega\Phi(s, u) + (1{-}\omega)\Psi(s, u)$, with average $c(s, u)$.

The average time and energy cost associated with the initial state are equal to 0. In the termination states s_{UAV} and s_{ES}, we have

$$\phi(s_{\text{UAV}}) = 1/\mu', \tag{5.12}$$

$$\psi(s_{\text{UAV}}) = (P_{\text{P}} + P_{\text{H}})/\mu', \tag{5.13}$$

and

$$\phi(s_{\text{ES}}) = 1/\mu, \tag{5.14}$$

$$\psi(s_{\text{ES}}) = P_{\text{H}}/\mu. \tag{5.15}$$

Herein, based on actual value obtained by means of experimental evaluations, we assume that the transmission energy expense PL/R_i is negligible compared to the processing and hovering energy expense. Note that in the termination states the action is pre-determined

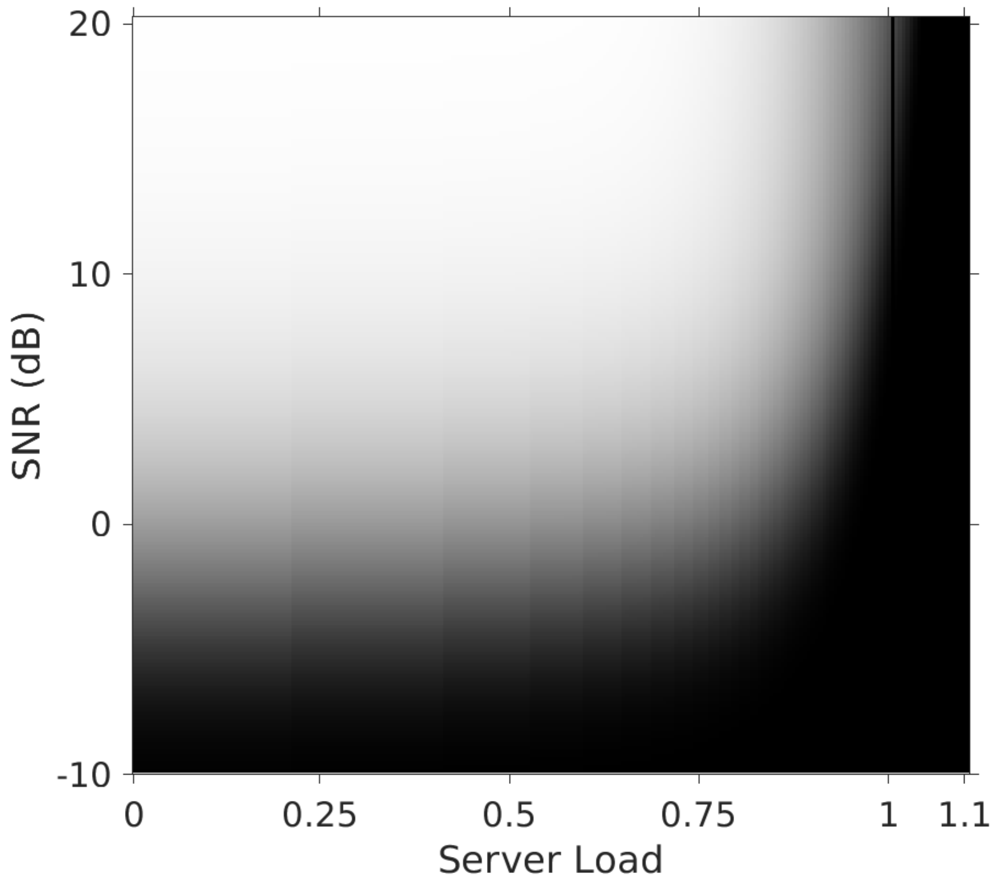


Figure 5.3: Probability of offloading to the edge server (lighter shades corresponds to higher probability) with $\omega = 0$.

and does not need to be formally included in the cost. From any transmission and queueing state $R_0, \dots, R_K$ and $B_1, \dots, B_{C+1}$, if the decision is to initiate local processing at the UAV (u=0), the process immediately moves to s_{UAV} and the energy and time cost are both equal to 0. Note that such decision is forced in states R_0 and B_{C+1}.

If the decision is to defer local processing (u=1), the costs are

$$\phi(R_i, 1) = L/R_i, \tag{5.16}$$

$$\psi(R_i, 1) = (P_{\mathrm{H}} + P)L/R_i. \tag{5.17}$$

with i=$, 1, \dots, K$, and

$$\phi(B_i, 1) = 1/\mu, \tag{5.18}$$

$$\psi(B_i, 1) = P_{\mathrm{H}}/\mu. \tag{5.19}$$

Herein, based on measurements obtained by means of experimental evaluations, we assume that the transmission energy PL/R_i is negligible compared to the computing and hovering energy expense.

The problem of minimizing the expected total cost can be rephrased as a Markov Decision Process over a finite temporal horizon. We aim at finding, then, the (deterministic) optimal policy $u^*(s)$, where

$$u^*(s) = \arg\min_{u=\{0,1\}} E\big(V_{\mathrm{res}}(s, u)\big), \tag{5.20}$$

where $E(V_{\mathrm{res}}(s, u))$ is the expected minimum cumulative residual cost to a termination state

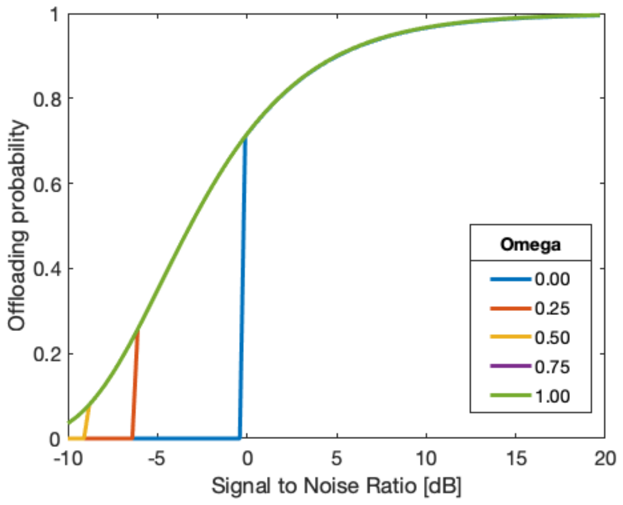


Figure 5.4: Probability of offloading to the edge server for different values of ω, as a function of channel quality (SNR) with $\rho = 0.5$. We use $\mu = 1/0.461/s$ to emphasize the observed effects.

$s^{\dagger}$ from state s given that decision u is selected, that is,

$$\min_{\mathbf{U_1}^{j^\dagger}} E\left(\sum_{j=0}^{j^\dagger} c(S(j), U(j)|U(0){=}u, S(0){=}s)\right), \tag{5.21}$$

where

$$j^\dagger = \min(j : S(j){\in}\{s_{\mathrm{UAV}}, s_{\mathrm{ES}}\}), \tag{5.22}$$

and $\mathbf{U}_1^{j^\dagger} = (U(0), \ldots, U(j^\dagger))$.

We compute the optimal policy using the Value Iteration method [5], which focuses on iterations producing policies achieving performance increasingly close to the optimal point. In our case, the optimal value form the starting state yields the experienced delay and energy consumption for each image, when $\omega = 0$ or $\omega = 1$ respectively. Values at other states, at convergence, represent the expected future cost from that state obtained using the optimal policy. Let V_t be the vector whose elements are the value function for each state in the state space, where t is the number of steps taken in the recursion. Then $V_t \in \mathcal{R}^{5+K+C}$, where the size of the vector derives directly from the states definition in Figure 5.2. The arbitrarily initialized vector V_0, is then updated using the Bellman equation as follows:

$$\begin{aligned} Q_{k+1}(s,a) &= \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma V_k(s') \\ V_k(s) &= \min_a Q_k(s,a), \end{aligned} \tag{5.23}$$

where $Q_t(s,a)$ represents the expected cost taking action a. Note that, to reduce the time to convergence, the updates are usually computed backwards, from the end nodes (in our case $S_{ES},\ S_{UAV}$) to the input node S_0.

5.4 Numerical Results

This section presents and discusses results obtained using the model and optimization technique proposed in this chapter. First, we analyze performance metrics and offloading probabilities exploring parameters describing channel quality and server load. We, then, characterize the performance of the proposed scheme in a realistic environment, where channel parameters (that is, SNR) is obtained based on trajectory of the UAV. Note that in this latter section of the results, we can analyze the temporal behavior of the system during the mission.

To make our observations more meaningful, we derive the optimal policies under different channel and load conditions using parameters obtained from real-word experimentation. These values are used unless otherwise stated. Specifically, we used a 3DR Solo Drone mounting a Pixhawk flight controller running ArduCopter connected to a Raspberry Pi model 3B as companion computer. We use as edge server a Laptop with 16GB RAM and Intel Core i7-6700HQ processor with Nvidia GM204M GPU. We set the number of pictures collected in each location to 1, where each picture has resolution equal to 720×480. The average

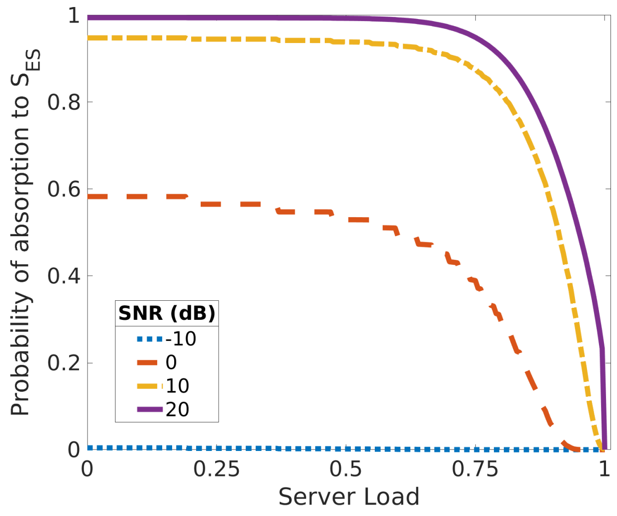


Figure 5.5: Probability of offloading the computation to the edge server as a function of the server load ρ.

size of each picture after encoding is 80 KB. The pictures are processed implementing a face recognition algorithm using a multi-scale Haar Cascade, which takes on average $1/\mu'$=0.56s at the UAV and $1/\mu$=0.046s at the edge server. We consider SNR values in the range $[-10, 20]$ dB and transmission rates in the range from 1 Mbps to 11 Mbps (matching a system using Wi-Fi IEEE 802.11). Power consumption rates are based on battery level readings in the same set up: in particular, we set $P_h = 0.1$ levels/s, $P_p = 10\% \cdot P_h$ levels/s. The optimal deterministic policy $\mathbf{U}_1^{j^\dagger}$ given the parameters is computed using Equation (5.21). We note here that our performance analysis will be impacted by an error due to the resolution chosen on the set of available rates and positions. As a result of the position error, a different expected SNR will be used, causing a difference in the probability distribution over the rates. These will be averaged out on multiple runs and so our results still hold. We consider on the other hand the rate resolution error in this case acceptable: the difference in Mbps in the Wifi protocol is much higher than the one considered. Our investigation takes into consideration several aspects of the system, and even though the error margin at small rates might is large ($100ms$), it is worth noting that in all other scenarios it reduces to a few milliseconds. Furthermore the overall policy behaviour does not change due to these effects, and in this respect while absolute values might differ, the statistics and system behaviour will be consistent with our results.

5.4.1 Performance Analysis

In Figure 5.3, we show the probability of offloading to the Edge Server as a function of SNR and server load $\rho = \lambda/\mu$. This probability corresponds to the probability of the process being absorbed in S_{ES} from S_0 conditioned on the control policy, defined as

$$P^{\infty}_{S_0}(Y) = \lim_{t \to \infty} P(S(t) = S_Y | S(0) = S_0, U = \mathbf{U}_1^{j^\dagger}) \tag{5.24}$$

where $Y \in \{S_{\mathrm{UAV}}, S_{\mathrm{ES}}\}$, and $P^{\infty}_{S_0}(S_{\mathrm{UAV}}) + P^{\infty}_{S_0}(S_{\mathrm{ES}})$=1. In Figure 5.3, we plot $P^{\infty}_{S_0}(S_{ES})$, using lighter pixel color for higher probabilities. As expected, for low values of ρ and high values of the SNR, the offloading probability is almost equal to 1, that is, the UAV offloads computation when system conditions are favorable. When the SNR is sufficiently low, the UAV will likely be disconnected, or the cost of offloading might exceed that of local computation due to the large time needed to transport the data to the edge server. Similarly, if the load parameter ρ is large, that is, the ES buffer has frequent arrivals or computation tasks take a large time to be completed, the UAV chooses to compute locally.

In Figure 5.4 we show the effect of ω, the parameter that controls the tradeoff between energy and delay in the objective function, over the optimal policy and consequently on the offloading probability. In the plot, each line corresponds to a different value of $\omega \in [0, 1]$, where the larger ω the larger the weight of energy cost. The impact of including energy in the optimization is apparent: the larger ω, the larger the offloading probability, even for low SNR, where transmitting over the channel may result in an increased overall delay. In fact, while a larger delay leads to a larger hovering time, and thus more hovering energy expense, offloading eliminates the energy cost associated with local processing. We observe

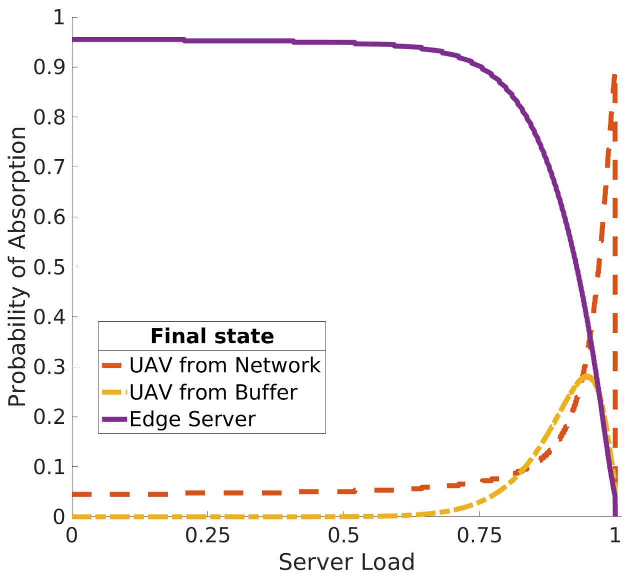
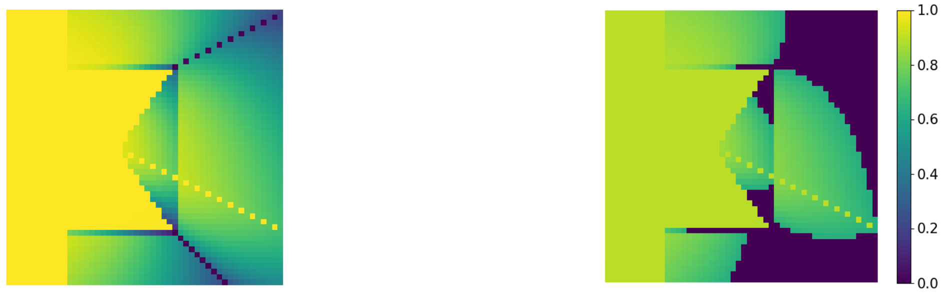


Figure 5.6: Probability of selecting local computation in the three main decision stages or offloading to the edge server.



(a)

(b)

Figure 5.7: Offloading probability on the considered map. The average SNR is set to 9dB and server load set to (a) 0% and (b) 70%.

an interesting threshold effect, where the policy transitions from fully local computing to partial offloading at an SNR value which is a function of ω.

In Figure 5.5 we fix the SNR, and show $P^{\infty}_{S_0}(S_{ES})$ as a function of ρ. As the SNR decreases, the probability of offloading to the edge server decreases as well. Intuitively, the SNR influences the shape of the probability curve. Interestingly, high SNR values show a sharp transition from offloading to local computing, whereas low SNR values have a more progressive transition, most likely due to the distribution of the communication time.

Finally we illustrate the value of probing compared to a simple decision informed by the average delay pre-computed based on a priori knowledge of the parameters. Figure 5.6 shows the probability that the decision of processing locally is taken at the different stages or that offloading is selected. Specifically, the decision stages are:

- **Stage 0:** the initial stage S_0, where the UAV knows the parameters, but not the channel or queue state;
- **Stage 1:** R_i, where the UAV has connected with the network and is aware of the maximum transmission rate;
- **Stage 2:** B_j, where the UAV reached the edge server, that is, upon transmission after transmission, and is reported the position in the processing queue.

For small values of ρ, offloading is predominant, with a small probability of local computing decision forced by extremely poor channel conditions. As ρ increases, the set of rates corresponding to local computing decisions increases. In fact, the delay requirement for the data transportation becomes more stringent as the average time spent in the edge server buffer increases. In the transition phase between offloading and local computation, we can observe a spike in the probability that the UAV will select local computing after the edge server is reached, as the probability finding a number of tasks in the buffer sufficiently large to make offloading disadvantageous increases before a region in which probing is not even attempted. The policies resulting from the abstraction of the system we adopt have a simple structure. Across the phases of the decision making, the agent will identify thresholds within layers of the Markov process corresponding to binary decisions. While the structure is simple, the thresholds are function of the distributions of channel quality and incoming load at the server.

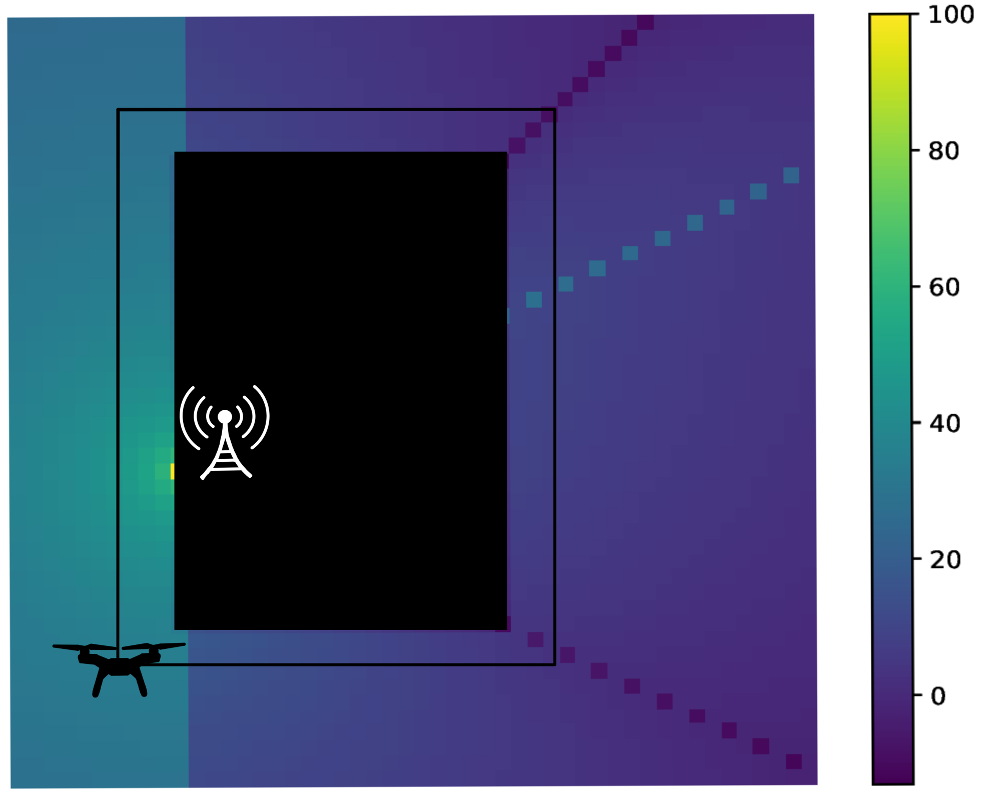


Figure 5.8: Map of the considered area, centered around a 30 m high building. Symbols display the access point's placement and the drone trajectory. Different shades show Signal To Noise ratio in dB across the area.

5.4.2 Mission Trajectory

We now analyze a mission trajectory of the UAV in an urban scenario inspired by applications such as city monitoring and building inspection. We consider the trajectory illustrated in Fig. 5.8, where the UAV flies at fixed altitude and constant distance from the building's external surface in a loop starting from the lower left corner and proceeding in clockwise direction. The map shown has delimits a 50×50 meters area, where both the access point and the UAV are at $15m$ altitude, and the building's width, length and height are 20m, 30m, 30m.

In the scenario, we consider a setting with one access point and one building, and we compute the SNR – to be plugged in our model, see Eq. (5.3) – using the building shadowing model [15]

(added to the attenuation caused by free-space propagation)

$$L = \alpha n + \beta d_0, \tag{5.25}$$

where α is the attenuation per wall (dB), n is the number of walls penetrated, β is the attenuation per meter (dB) and d_o the distance in meters traveled through obstacles. We use experimentally validated coefficients in [15] $\alpha = 9dBm$ and $\beta = 0.9dB/m$

Using the same set of parameters as in the previous set of results, we find the optimal policy for each position in the considered map. In Fig. 5.7, we show how the probability of offloading to the edge server evolves along the trajectory. In Fig. 5.7.a, the edge server is dedicated to the UAV. The impact of the additional attenuation effect of the building on the strategy is apparent: the offloading probability decreases in regions that are more affected by the additional attenuation. Overall, the low server load leads to offloading being a predominant strategy. Fig. 5.7.b shows the same map where we increase the server load to 70%, and we can see that the adaptive scheme reacts selecting edge computing as the best strategy in a smaller fraction of realizations. This effect is due to the higher chances that the task generated by the UAV will find several other tasks in the server's buffer.

We now consider the delay performance over the full trajectory, and illustrate how the strategy evolves. In Fig. 5.9, we plot the average capture-to-output delay achieved by the optimal strategy. We can observe how, for different values of SNR, low performance regions expand. This is due to the higher probability that local computing will be chosen due to the low data rate supported by the channel. Interestingly, we can observe how new spikes and low performance regions emerge as noise increases and the strategy switches to different modalities in some regions.

The server load ρ has a much different impact, as shown in Fig. 5.10. The average delay increases homogeneously along the trajectory to reach a cap determined by the policy always

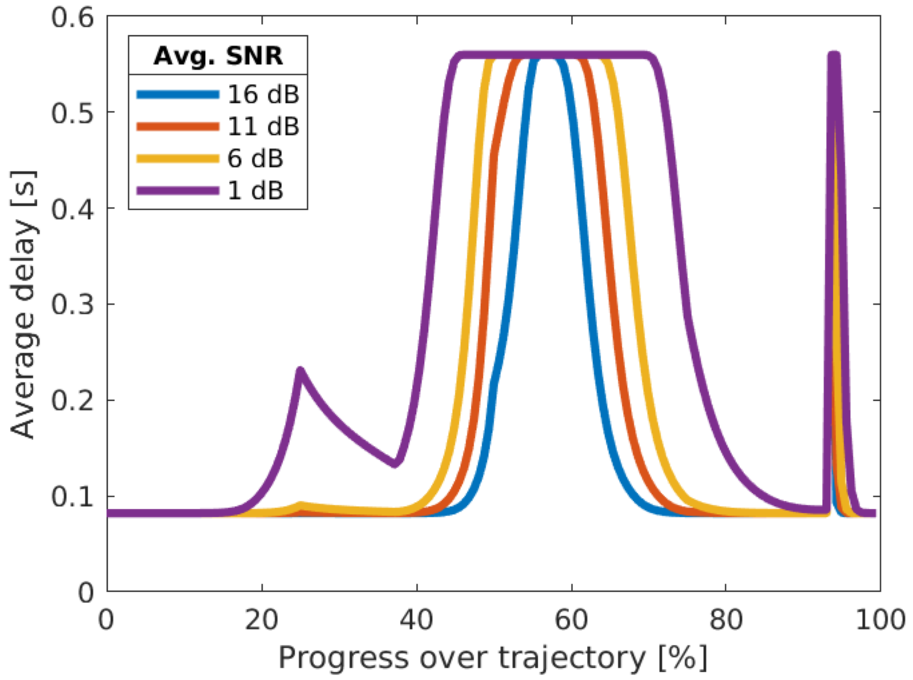


Figure 5.9: End-to-end delay over the described trajectory for different values of external interference with no server load.

choosing local analysis. Additionally, we observe how moderate server loads have relatively low impact (*e.g.*, 25% vs 50%).

Considering the average over the full trajectory, we now explore how some parameters affect performance and strategies. In Fig. 5.11 and Fig. 5.12, we display the average delay (in blue) and offloading probability (in orange), averaged over the trajectory, for different noise levels and edge server's load. Interestingly, while the delay has a clear inverse relationship with the offloading probability when varying average SNR, the relationship between average delay and offloading probability is less marked when varying the edge server's load. In the former plot (Fig. 5.11), we have a sharp change at $\approx\ 10dB$, where the probability of a successful offload sharply decreases and the delay increases due to the more frequent selection of local processing. In the latter plot (5.12), we observe a low sensitivity of the delay, where in the low to moderate load region we have an increasing average delay, but a minor change in the offloading strategy. The delay experiences a sharp increase when the offloading probability

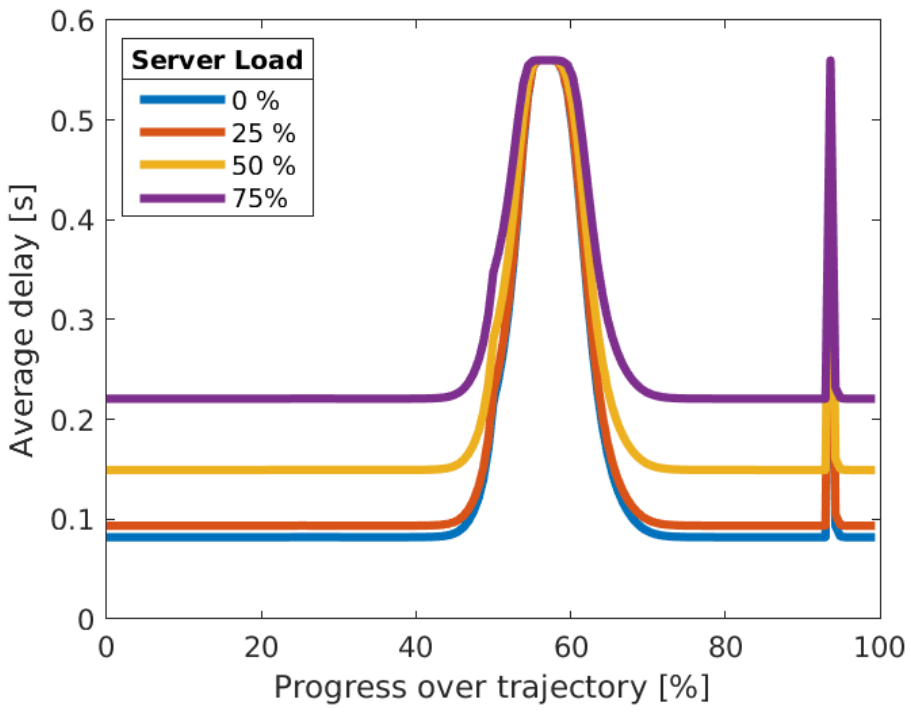


Figure 5.10: End-to-end delay over the trajectory for different values of server load with average SNR of 16dB.

sharply decreases in the high load region. This effect is due to the more gradual degradation imposed by increasing load compared to that of a worsening SNR.

We now characterize the impact of the computing capacity of the devices, expressed as their service rates. We explore a range of values of $\mu \in [1, 40]$ for the edge server, $\mu' \in [1, 8]$ for the UAV, to evaluate the impact of different design choices and operational settings. We highlight the advantage of an adaptive approach, by depicting the percent delay increase when a fixed offloading policy is used. We remark that we are still considering averages over the entire trajectory.

The gain is shown in Fig. 5.14, where we we set the load to 0 (a) and 70% (b). Note in both graphs that higher gains (darker regions), are focused in high local service rates and relatively low edge server service rates areas. This conveys the fact that adaptation can improve performance only when the strategy is non-trivial and adaptation can bring benefit.

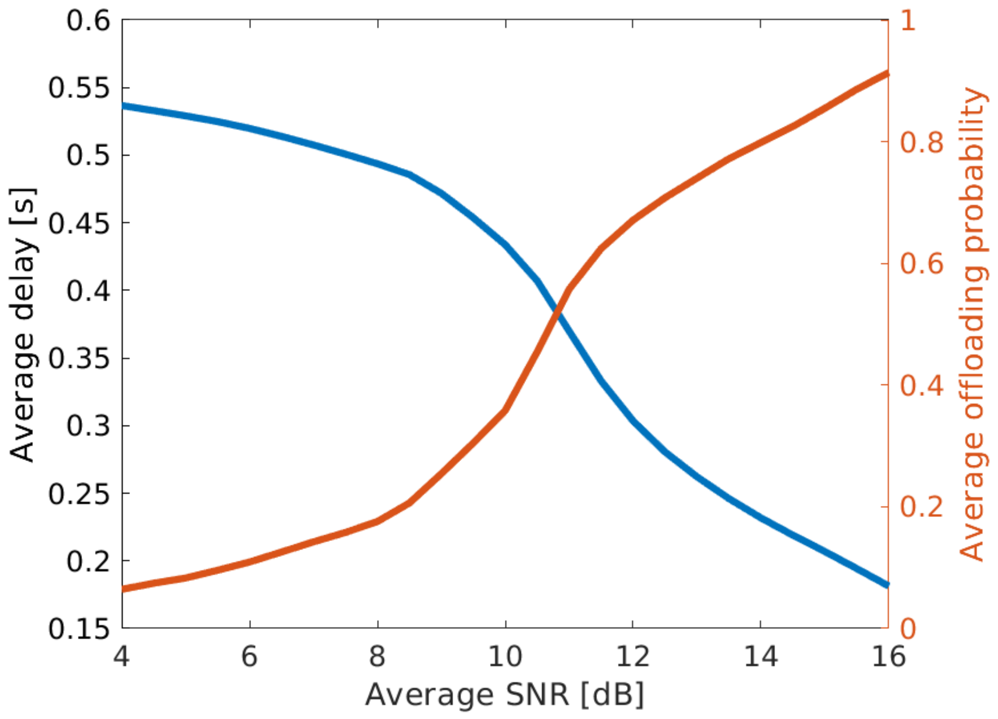


Figure 5.11: Delay average over the full trajectory for varying average SNR for both only offloading policy and our approach. We also plot the probability for the UAV to successfully offload in our schema.

When local service rate is small (left portion of the graphs), we indeed observe than the offloading policy,

Interestingly, we see how the gain granted by the adaptive strategy is higher when the load is higher, this due to the fact that our policy can effectively fall back to local computing when the buffer is busy in specific realizations of the process.

In Fig 5.15.a we can see how the average delay using an adaptive policy is very sensitive to the local processing capacity, but has a weak dependency on the edge server processing capacity. However, as shown in Fig 5.15.b, we can see that the delay's variance is higher in the areas where the gain is small. In fact, the advantage of the adaptive technique is to choose local processing whenever it seems advantageous, and that is shown in the areas where lower variability maps to local processing being the optimal policy.

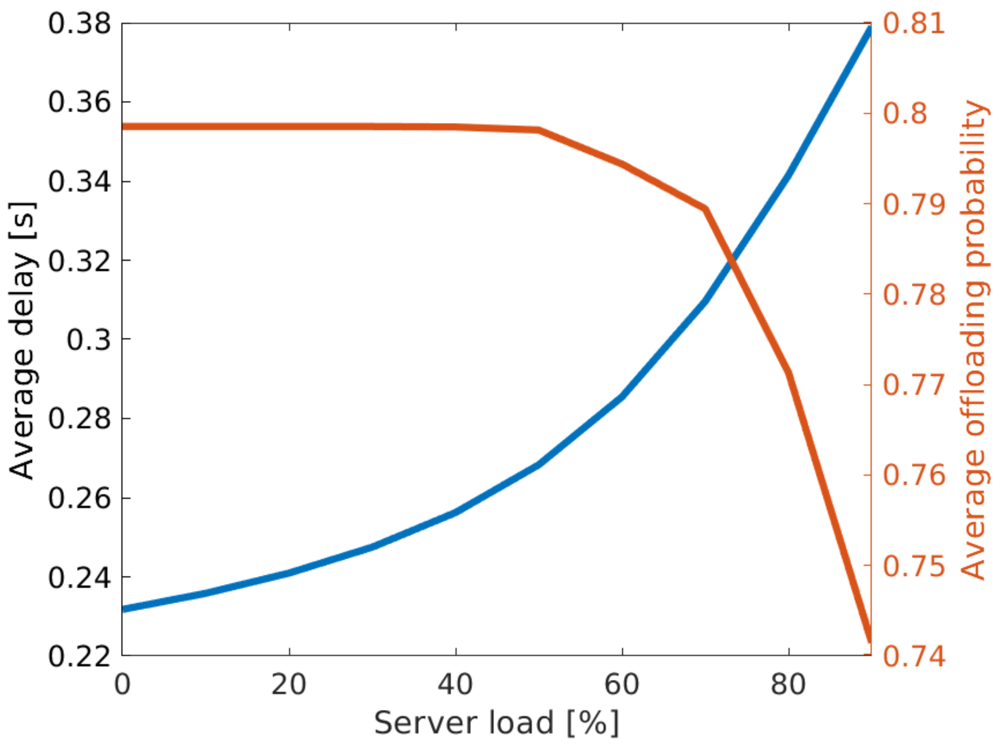
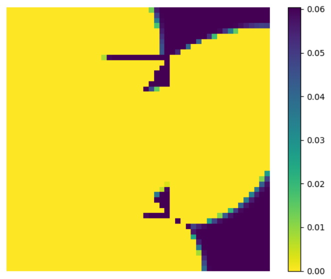
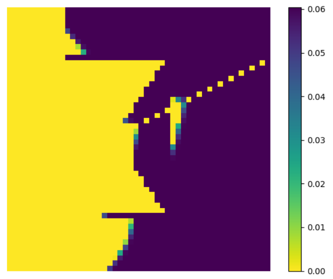


Figure 5.12: Delay average over the full trajectory for varying server load, and the probability to successfully offload.



(a) average SNR of 16dB

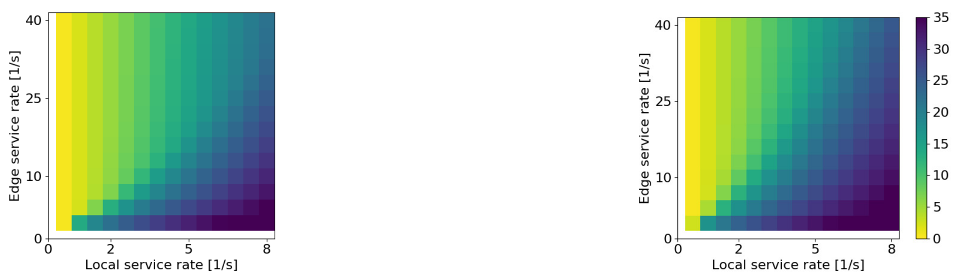


(b) average SNR of 6dB

Figure 5.13: Delay improvement using adaptive schema compared to constant offloading at different average SNRs.

5.4.3 Characterization of State in Temporally Correlated Environment

We built on these results and created an event based simulator that allows us to capture the temporal correlations between subsequent positions along the trajectory. We use this tool in



(a) $\rho = 0\%$ (b) $\rho = 70\%$

Figure 5.14: Gain percentage over a full trajectory for different hardware configurations. Processing speeds are referenced as serving rates μ. Both cases have average SNR at 16 dB, but they differ in edge load ρ.

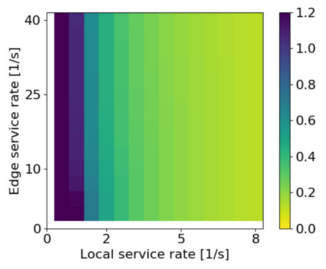
order to study the impact of different state representations on the performance of a decision agent.

In the state representations we include:

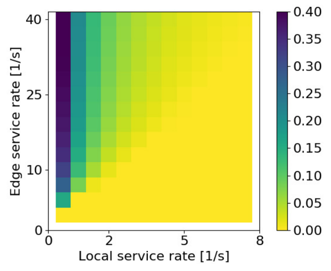
- phase: before offloading, at the edge server or processing locally
- position: the (x, y) coordinates on the map
- E[SNR]: average SNR at the current position
- num. jobs: number of jobs in the queue
- past (action, delay) tuples: we include the last 3 actions and delays.

Although in a collaborative system we would expect to collect all the above, if using a third-party infrastructure or different networking protocols, it could be difficult or impossible to collect some of this dynamic information. For this reason we study the same system using three different observed states:

1. full state
2. phase, position, past action-delays



(a)



(b)

Figure 5.15: (a) Delay and (b) standard deviation for delay over a trajectory for different hardware configurations with average SNR of $16dB$ and no server load. Processing speeds are referenced as serving rates μ.

3. phase, past action-delays

Due to the nature of this state, where some elements are discrete, other continuous, we involve the use of a function approximator, such as Deep Neural Networks (DNN), to learn the q-function. In order to learn from experience in this new setting, we resort to some techniques investigated in Deep Reinforcement Learning (DRL). DRL has been successful in environments such as Atari games [41], the game of Go [51] and many others [54]. As shown in these works, there are many details that help DRL converging to a stable approximation for the q-function. In fact naive approaches often do not work, due to the inherent difference in the way DNNs learn with respect to the tabular approach. For example, DNNs are subject to catastrophic forgetting [22], the tendency of forgetting early examples seen in the training procedure. To address this problem Schaul et al. introduced replay buffer [49], a buffer where samples are stored after observation. We use samples randomly extracted from the buffer, and feed them to the DNN in batches. The model will compute the function $Q(\cdot)$, and return a vector with q-values for each of the actions. However since we can take only one action, we will observe only one reward, and therefore we will update, i.e. compute the loss and apply back-propagation, only relative to one output. We use the Bellman equation based on one step Temporal Difference to compute the estimate q-value for the next step:

$$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(R(s,a) + \gamma \texttt{argmax}_{a'} \hat{Q}(s',a')) \tag{5.26}$$

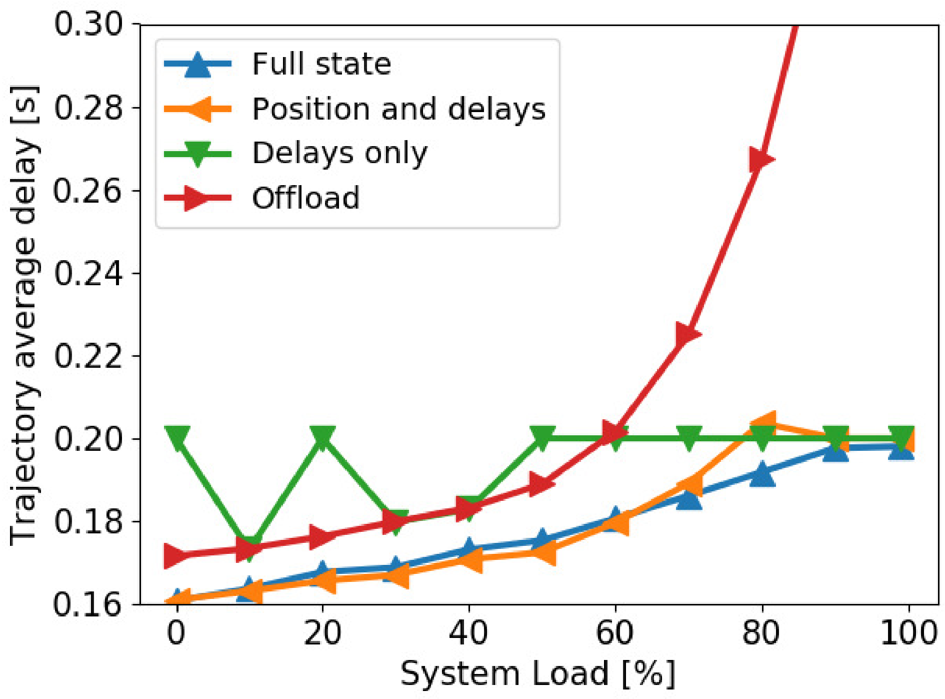
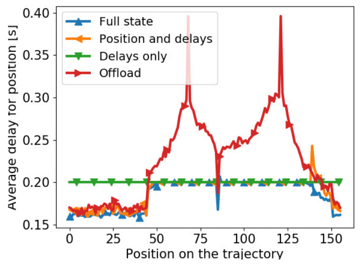


Figure 5.16: Average delay over the full trajectory for different server loads shows the adaptability of agents that have spatial awareness.

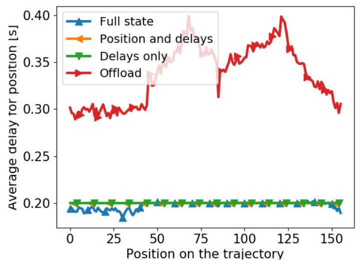
where α is learning rate, γ is discounting factor, and $\hat{Q}$ is an old version of the DNN. We use to different networks $\hat{Q}, Q$, so that the q-value estimation does not vary too quickly, causing divergent behaviour. We update $\hat{Q} = Q$ periodically to improve our approximation of the q-function.

As mentioned, we maintain the setup the same as in the previous setting, where the agent has to decide whether to continue offloading or not at two stages: when it arrives to a position and when it gets in queue to the edge server. From the agent's point of view all transitions are probabilistic, since the rate might be 0 and the queue might be full. In order to discourage fixed policy of only offloading, we have a small network probing term that is added to the delay when a transmission is unsuccessful.

We trained different agents using the state representation described earlier, and obtained insights into what carries useful information in predicting the optimal policy at each position.



(a) $\rho = 70\%$



(b) $\rho = 90\%$

Figure 5.17: Delay over the trajectory for different state representations at different server loads.

In Fig. 5.16 we display the average delay of each of the agents going through the trajectory. We can observe that using all the information available gives the agent an advantage similar to what we observed earlier, since it is able to choose the correct policy for each given channel situation. Interestingly, removing average SNR and position in queue does not influence the performance for loads lesser than 70%. There we see the two lines (blue for full state and orange for only position and delays) diverging, with the less informative state having higher average delay. The critical advantage of these two policies on delays only and fixed policies, can be observed in Fig. 5.17: we notice that the central part of the trajectory, where the line of sight is obstructed by the building, offload strategy incur in very high delays. For higher loads, we can also see how the position in queue at the edge server changes the ability of the user to exploit offloading only when advantageous: in Fig. 5.17 we see that only the agent that has access to the position in the server buffer is able to exploit successfully offloading in positions 0-50 in high load regimes.

This study reveals how the position along the trajectory is a viable proxy for the average Signal to Noise Ratio, and how on the other hand previous delays are not as predictive of the number of processes in queue at the server. Furthermore this opens the opportunity for further investigation in spatial-temporal maps that can embed this information and allow dynamic adaptation in continuous learning settings.

Chapter 6

Optimal Task Allocation for Edge Computing Systems with Split DNN Computing

Many modern mobile applications rely on complex machine learning algorithms, such as Deep Neural Networks (DNNs), to analyze images and extract information on their content. The high computational complexity of these algorithms clashes with the constrained computing and energy resources available to mobile devices. To address this issue, the research community proposed two main approaches: (*i*) reducing the complexity of the DNN models to fit within the constraints of the mobile device [53], and (*ii*) offloading the computing task to more powerful computers, such as edge servers [4]. On the one hand, the former option inevitably results in some degradation of the DNN output with respect to full models. On the other hand, edge computing necessitates the transfer of – possibly high resolution – images over wireless links. The instability of wireless links, and network load patterns in general, may degrade the performance of this strategy.

A recent trend of contributions proposed splitting the execution of DNNs models between the mobile device and edge server [29, 21, 36, 38, 37]. The idea is to divide DNN models into head and tail portions, which are executed at the mobile device and edge server, respectively. The channel, then, transports the output tensor of the head model to the edge server. Unfortunately, the structure of DNN models for vision task does not allow effective splitting, as they typically concentrate most computational complexity in the early layers, where they also tend to amplify the input size. Intuitively, splitting such architectures would result in an excessive computation load to the mobile device, as well as no advantage in terms of channel load. Some split DNN approaches, then, introduce a bottleneck layer early in the DNN structure to compress the input image into a small tensor, thus mitigating channel impairments that are the main source of performance degradation in edge computing-based systems [21, 36, 38, 37].

In this chapter, we take as a starting point the splitting approach we presented in [36], where the modification of the architecture was paired with a specific training strategy - Knowledge Distillation - applied to the first section of the model. Knowledge Distillation trains that portion of the modified model – which contains the bottleneck – to mimic the output of the original section of the model. The model is then split at the bottleneck to achieve compression. This approach showed some important advantages, including the ability to generate small bottlenecks without sacrificing overall accuracy even in complex vision tasks, such as classification on the ImageNet dataset.

However, splitting is of course optimal only in some regions of parameters describing the channel capacity and the characteristics and state of the edge server (*e.g.*, task queue length and computing capacity). In general, the three main options, local computing, edge computing and split computing, may be optimal in different conditions. As the system state evolves over time due to channel and queueing dynamics, a scheduling problem arises, where the mobile device needs to determine which one of the three options to choose. Intuitively, as tasks may accumulate, the decision needs to be optimal considering the statistics of the future system's state.

In this chapter, we present a scheduling problem determining how images periodically produced by a mobile device are processed. We consider a system including a sensor generating images at the mobile device, processing units at the mobile device and edge server, a time-varying communication channel, and a selector deciding how each image is processed. As task flow may exceed the capabilities of the processing units and communication channel, we include in the system finite buffers to accumulate tasks to be completed and data to be transmitted. Notably, the presence of buffers, a crucial components of real-world systems, induces temporal correlation, where the choices of the selector at a given time influence the distribution of future system's states.

We model the system as a Markov process, and formulate an optimization problem whose objective is to minimize the average time between image capture and the availability of the analysis outcome and the number of images rejected by the buffers. The problem is mapped to a Linear Fractional Program (LFP), whose optimization variables are state-action stationary frequencies.

The rest of the chapter is organized as follows. In Section 6.1, we provide an overview of prior work, and briefly explain the split DNN technique we proposed in [36]. Sections 6.2 and 6.3 describe the system and the Markov process used to model its dynamics. Section 6.4 formulates the optimization problem and presents the resolution methods. Results are shown and discussed in Section 6.5.

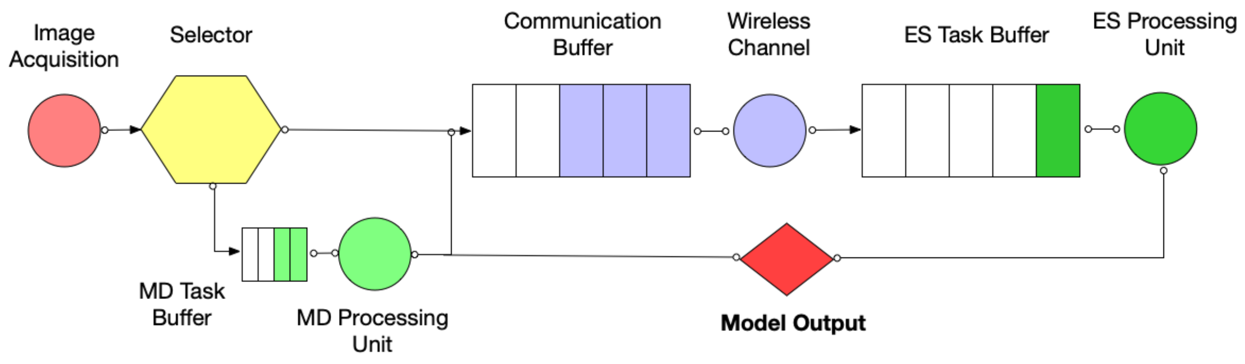


Figure 6.1: Schematics of the system considered in this chapter.

6.1 Distilled Split DNN Models

There are several methods to make DNN models deployable such as training lightweight models [48, 53], model compression and pruning [23, 33]. Such approaches, however, often experience significantly degraded accuracy of the model predictions and/or require many iterations of complex operations in training and optimization.

Kang *et al.* [29] and Jeong *et al.* [27] propose to simply split DNN models in an edge computing scenario. However, such approaches are not well motivated, as many state-of-the-art DNN models do not present bottlenecks – that is, layers with few nodes – in the early layers. As a result, splitting is often suboptimal compared to local or edge computing from a point of view of total inference time. Recent studies attempt to introduce bottlenecks by modifying the architecture and training of the models [21, 36, 38, 37].

For the sake of completeness, we briefly summarize here the split DNN technique we proposed in [36]. As mentioned in the introduction, the core idea is to introduce a bottleneck layer, that is, a layer composed of few nodes, in the early stages of the model. This enables to achieve in-network compression of the input while limiting the computing load assigned to the mobile device. We focus on complex DNN models: DenseNet-169, -201 [26], ResNet-152 and Inception-v3 [52], where we first modified the models to introduce such bottlenecks, and retrained the altered model using a technique called head network distillation [36]. The

Table 6.1: Classification performance of head-distilled (student) models with bottlenecks [36]

Altered model	DenseNet-169	DenseNet-201	ResNet-152	Inception-v3
Test accuracy [%]	83.3 (-1.2)	84.1 (-1.1)	83.2 (-1.1)	85.7 (-0.8)
Data size [%]	1.68	1.68	1.68	1.53

* The numbers in brackets indicate difference in accuracy from the original (teacher) models.

technique stems from *knowledge distillation* [24], a procedure used to train a smaller (student) model to mimic a bigger (teacher) model's output. Interestingly, it is reported that student models trained to mimic the teacher model often outperform equivalent models trained using a vanilla method. Since our focus is on introducing bottlenecks to the early layers of the pretrained model, we only train the head portion of the altered model to reduce training time.

Table 6.1 summarizes the head-distilled models' accuracy and bottleneck tensor size scaled by the input tensor size ($3 \times 299 \times 299$ for Inception-v3 and $3 \times 224 \times 224$ for others) as reported in our previous work [36]. With small test accuracy loss, our introduced bottlenecks reduce the size of data to be transferred to the edge computer by approximately 98% compared to the input tensor size. The reduction in network payload corresponds to a reduced communication delay withe respect to transmitting the input data, as shown in Section 6.5.

6.2 System Description

We consider a system composed of a mobile device (MD) and an edge server (ES). The overall objective of the system is to analyze images acquired by the MD in the shortest possible time and with the highest possible accuracy with the assistance of the ES. We consider a specific family of vision tasks, that is, image classification. In this work, we focus on a specific classification model being used to analyze all the images produced by the MD.

We denote the total time lapse from image acquisition to the availability of the output as T. To minimize T, the MD has three options:

Local Computing: the MD executes the DNN model using its own resources.

Edge Computing: The MD transmits the full image to the ES, which executes the model and transmits the outcome to the MD.

Split Computing: The MD executes the head model, transmits over the wireless channel the output tensor to the ES, which executes the tail model and sends back the outcome to the MD.

As explained in [36], the three different choices are optimal in different regions of parameters describing the computing capacity of the MD and ES, as well as the capacity of the wireless channel connecting them.

Herein, we develop a technique to allow the MD to dynamically select the best option in response to the system state. As both computing and communication tasks may accumulate within the system, we describe the latter as the concatenation of queues illustrated in Fig. 6.1. The MD acquires images, which are forwarded to a selector to determine which of local, edge and split computing is used. In the first case, the task is forwarded to the task buffer of the MD, and eventually executed by the embedded MD's processor. In the second case, the full input image is forwarded to the communication buffer and eventually delivered over the communication channel to the ES task buffer for processing. In the third case, the image is sent to the local task buffer, but only the head portion of the model is executed by the embedded processor, which then forwards the tensor to the communication buffer for transmission to the ES task buffer. The ES then executes the tail portion. Note that in the edge and split computing options, the model output needs to be transmitted back to the MD. The size of the MD task buffer, ES task buffer and communication buffer are denoted with

N_{md}, N_{es} and N_{comm}, respectively. We adopt a First-In First-Out service model.

The total delay T_i of image i is the sum of many components, whose value depends on the computing capacity of the MD and ES – which here is assumed fixed – as well as the current channel capacity and the state of the buffers, which vary over time. In order to minimize the average delay, the selector, then, inevitably needs to implement a policy capable of reacting to the dynamics of the system's state.

6.3 Stochastic Model

In this section, we characterize the state space and dynamics of the stochastic process associated with the system described earlier. We note that in the following we use capital and lowercase letters to denote random variables and their values, respectively.

6.3.1 State Space

We define the state space of the system as the vector

$$\mathbf{s} = [c, \mathbf{b}_{\mathrm{md}}, \mathbf{b}_{\mathrm{comm}}, \mathbf{b}_{\mathrm{es}}], \tag{6.1}$$

where c is the state of the wireless channel, and $\mathbf{b}_{\mathrm{md}}$, $\mathbf{b}_{\mathrm{comm}}$, and $\mathbf{b}_{\mathrm{es}}$ are vectors describing the state of the MD task buffer, communication buffer and ES task buffer, respectively. The state of the channel corresponds to the Signal-to-Noise-Ratio (SNR) experienced by the link. We quantize the SNR to define C transmission rates, obtained using a capacity model, that is

$$\Psi(c) = W \log_2(1 + \mathrm{SNR}_{\mathrm{c}}), \tag{6.2}$$

where SNR_c is the SNR associated with channel state $c \in \{1, \ldots, C\}$, and W is the channel bandwidth.

The vector $\mathbf{b}_{\text{md}} = \{a_k\}_{1,\ldots,N_{\text{md}}}$ contains N_{md} elements each of those is associated with a slot in the MD task buffer. The variables a_k lie in the set $\{0, \text{full}, \text{split}\}$, whose elements respectively correspond to an empty slot, a slot containing an input image to be processed using the original full model, and an image to be processed using the head model only. The buffer is organized to rank tasks in order of arrival, that is, the oldest task is in the first position, the second oldest on the second and so on, and empty slots are at the vector end. We define the functions

$$Z_{\text{md}}(\mathbf{z}) = z, \quad F_{\text{md}}(\mathbf{z}) = a_1, \tag{6.3}$$

where z is the number of empty slots and a_1 is the first element in the vector $\mathbf{b}_{\text{md}}$ within the state $\mathbf{z}$. We define analogous functions extracting the same quantities from $\mathbf{b}_{\text{es}}$ and $\mathbf{b}_{\text{comm}}$, whose definition is analogous. The elements a_k of $\mathbf{b}_{\text{comm}}$ are associated with empty slots (a_k=0), input images (a_k=full), and tensors (a_k=split) to be delivered to the ES buffer for processing. The elements a_k of $\mathbf{b}_{\text{es}}$ correspond to empty slots (a_k=0), input images to be fully processed by the ES (a_k=full), and tensors (a_k=split) that are inputs to the tail DNN model. On these models, we define similar functions as in Eq. 6.3.

In addition to the system state, we define the decision variable $u \in \{\text{lc}, \text{ec}, \text{split}\}$. The components of the decision space correspond to a task being fully executed locally at the MD, being fully offloaded to the ES, and split computing.

6.3.2 System Dynamics

In order to analyze the system and locate the optimal selection policy, we make some assumptions that are common in queueing system analysis. Specifically, we assume that the image interarrival time at the MD, the data transfer time, and the task execution time are exponentially distributed random variables. The distributions are centered on values extracted from the real-world experiments reported in [36].

We define, then, the following parameters

• λ as the arrival rate of images (that is, $1/\lambda$ is the average inter-capture time of images),

• γ_{full} and γ_{head} as the execution rate of the full and head DNN models at the MD, respectively.

• ρ_{full} and ρ_{tail} as the execution rate of the full and tail DNN models at the ES, respectively.

• $\nu_{(\text{i,in})}$ and $\nu_{(\text{i,head})}$ as the transmission rate (*i.e.*, the channel service rate) of full images and output tensors, respectively. The parameters are computed as the channel capacity of that state divided by the data size of the input image/tensor.

• Finally, we assume a *jump* model for the channel, where the channel state switches from state i to state j with probability ϕ_{ij} after an exponentially distributed time with parameter ω.

We emphasize that the service rate of the MD and ES task buffers depends on the nature of the task being executed (oldest in the buffer), which can be either the execution of the full DNN model, or the execution of the head (MD) and tail (ES) DNN models. Similarly, the channel service rate depends on the tag associated with the oldest data chunk in the buffer, but also on the current channel capacity (described by the element c in the overall state vector).

Under the assumption that the service rates are exponentially distributed as defined above, the system dynamics can be described as a stationary Semi-Markov process $\{S_t\}$ with a finite state space as described in Section 6.3.1, where $t{=}1, 2, \ldots$. Different from plain Markov processes, in Semi-Markov processes the permanence time in each state is a random variable. The discrete temporal index t, then, refers to time instants right after a state change. We remark that as the timing of all events in the system is determined by exponentially distributed random variables, then the probability that a particular event is the next is computed as the probability that the corresponding random variable is the smallest. Moreover, again due to the exponential nature of inter-event time, the residual time of the variables upon the occurrence of an event has the same distribution. Importantly, the overall inter-event time is distributed as the minimum set of exponentially distributed variables.

Assuming a recurrent Markov chain, the long-term dynamics of the process are fully defined by the transition probabilities

$$P_u(ij) = \Pr\{S_{t+1}{=}j \mid S_t{=}i, U_t{=}u\}, \tag{6.4}$$

where U_t is the decision variable at time t. Listing the transition probabilities is laborious and cumbersome. Instead, we provide an operational description of the process dynamics and associated probabilities.

Consider an empty system, that is

$$\mathbf{s}_{0,c} = [c, \mathbf{0}, \mathbf{0}, \mathbf{0}], \tag{6.5}$$

where $\mathbf{0}$ are zero-vectors of an appropriate size. In this state, then, all the buffers are empty, and the channel is in state c. Intuitively, the next "event" driving the system in a different state can be either (i) the channel state changes, or (ii) a new image arrives. The probability of event (i) being the first to happen is simply $\omega/(\omega + \lambda)$. Then the probability that the process transitions from $\mathbf{s}_{0,c}$ to $\mathbf{s}_{0,c'}$ is equal to $\phi_{cc'}\omega/(\omega + \lambda)$. Note that u is irrelevant in this case. The probability that the next event is (*ii*) is $\lambda/(\omega + \lambda)$. In this case, the decision variable determines the next state, and we have the following transitions with probability $\lambda/(\omega + \lambda)$:

$$\mathbf{s}_{0,c} \rightarrow [c, [0, \ldots, 0, \text{full}], \mathbf{0}, \mathbf{0}] \quad \text{if} \quad u = \text{lc}, \tag{6.6}$$

$$\mathbf{s}_{0,c} \rightarrow [c, \mathbf{0}, [0, \ldots, 0, \text{full}], \mathbf{0}] \quad \text{if} \quad u = \text{ec}, \tag{6.7}$$

$$\mathbf{s}_{0,c} \rightarrow [c, [0, \ldots, 0, \text{split}], \mathbf{0}, \mathbf{0}] \quad \text{if} \quad u = \text{split}. \tag{6.8}$$

Thus, if u=lc a full size image is sent to the MD's task buffer for full processing (note that the full DNN model is immediately executed locally as the task is in first position), if u=ec a full size image is sent to the communication buffer (and transmission immediately begins), and if u=split then the full image is sent to the MD's task buffer and the head model is used to generate a tensor. From $\mathbf{s}_{0,c}$, all the other transitions have a probability equal to zero. Consider a state $\mathbf{s} = [c, \mathbf{b}_{\text{md}}, \mathbf{b}_{\text{comm}}, \mathbf{b}_{\text{es}}]$, where $0 < Z_{\text{md}}(\mathbf{s}) < N_{\text{md}}$, $0 < Z_{\text{comm}}(\mathbf{s}) < N_{\text{comm}}$, and $0 < Z_{\text{es}}(\mathbf{s}) < N_{\text{es}}$. Thus, all the buffers are non-empty, but also non-full.

From $\mathbf{s}$, the following events might be the first to occur: (*i*) a new image arrives; (*ii*) a task is completed in the MD task buffer; (*iii*) a task is completed in the ES task buffer; (*iv*) the transmission of a data chunk from the communication buffer is completed; and (*v*) the channel changes its state. As expected, when (*i*) to (*iv*) occurs, the task/data chunk is removed from the buffer, and either sent to the next buffer or removed from the system. Event (*v*) is different, in that the state vector remains the same excluding (possibly) the channel state variable c.

Again, the probability that the first event is a specific one in the set of the five possible is the probability that the corresponding exponential variable controlling the time is the smallest in the set. We, then, define the variables

$$\gamma = \gamma_{\text{full}}\mathbf{1}(F_{\text{md}}(\mathbf{z}) = \text{full}){+}\gamma_{\text{head}}\mathbf{1}(F_{\text{md}}(\mathbf{z}) = \text{split}) \tag{6.9}$$

$$\rho = \rho_{\text{full}}\mathbf{1}(F_{\text{es}}(\mathbf{z}) = \text{full}){+}\rho_{\text{tail}}\mathbf{1}(F_{\text{es}}(\mathbf{z}) = \text{split}) \tag{6.10}$$

$$\nu = \nu_{c,\text{in}}\mathbf{1}(F_{\text{comm}}(\mathbf{z}) = \text{full}){+}\nu_{c,\text{head}}\mathbf{1}(F_{\text{comm}}(\mathbf{z}) = \text{split}) \tag{6.11}$$

where $\mathbf{1}(x)$ is the indicator function of event A. We remark that λ and ω are the parameters of the variables determining image arrival and channel state change frequency, respectively. We also define $\mu{=}\lambda + \omega + \gamma + \rho + \nu$.

The probability that the first event from $\mathbf{z}$ is (i), (ii), (iii), (iv) or (v) is λ/μ, γ/μ, ρ/μ, ν/μ and ω/μ, respectively. Given the occurrence of any of these specific events, the next state is deterministic irrespective of the decision variable u, excluding in (v), where the channel transition statistics determine the next state.

Let's define the state vector $\mathbf{s}' = [c', \mathbf{b}'_{\text{md}}, \mathbf{b}'_{\text{comm}}, \mathbf{b}'_{\text{es}}]$. Given the occurrence of (v), the state remains the same excluding the channel component, that is, the state transitions to $\mathbf{s}' = [c', \mathbf{b}_{\text{md}}, \mathbf{b}_{\text{comm}}, \mathbf{b}_{\text{es}}]$ with probability $\phi_{cc'}$.

Let's now look at the individual state vectors:

$$\mathbf{b}_{\text{md}} = [0 \dots 0 \;\; a_{k_{\text{md}}} \dots a_1], \tag{6.12}$$

$$\mathbf{b}_{\text{comm}} = [0 \dots 0 \;\; a_{k_{\text{comm}}} \dots a_1], \tag{6.13}$$

$$\mathbf{b}_{\text{es}} = [0 \dots 0 \;\; a_{k_{\text{es}}} \dots a_1], \tag{6.14}$$

where $k_{\text{md}}{=}N_{\text{md}}{-}Z_{\text{md}}(\mathbf{z})$, $k_{\text{es}}{=}N_{\text{es}}{-}Z_{\text{es}}(\mathbf{z})$, and
$k_{\text{comm}}{=}N_{\text{comm}}{-}Z_{\text{comm}}(\mathbf{z})$. Given the occurrence of (i), the process transitions to the following

states

$$\mathbf{s}\rightarrow[c\ [0\ldots\ \text{full}\ a_{k_{\text{md}}}\ldots a_1]\ \mathbf{b}_{\text{comm}}\ \mathbf{b}_{\text{es}}]\ \ \text{if}\ \ u{=}\text{lc}, \tag{6.15}$$

$$\mathbf{s}\rightarrow[c\ \mathbf{b}_{\text{md}}\ [0\ldots\ \text{full}\ a_{k_{\text{md}}}\ldots a_1]\ \mathbf{b}_{\text{es}}]\ \ \text{if}\ \ u{=}\text{es}, \tag{6.16}$$

$$\mathbf{s}\rightarrow[c\ \mathbf{b}_{\text{md}}\ [0\ldots\ \text{split}\ a_{k_{\text{md}}}\ldots a_1]\ \mathbf{b}_{\text{es}}]\ \ \text{if}\ \ u{=}\text{split}, \tag{6.17}$$

with probability one.

If (*ii*) occurs and the first element of $\mathbf{b}_{md}$ is equal to **full** then the vector shifts right (removing the first element) and a zero is appended. If the first element of $\mathbf{b}_{\text{md}}$ is equal to **split**, then that first element is moved to replace a zero in $\mathbf{b}_{\text{comm}}$, the vector $\mathbf{b}_{\text{md}}$ shifts right (removing the first element) and a zero is appended. If (*iii*) occurs, then the vector $\mathbf{b}_{\text{es}}$ shifts right (removing the first element) and a zero is appended. If (*iv*) occurs, then the first element of the $\mathbf{b}_{\text{comm}}$ is moved to replace a zero in $\mathbf{b}_{\text{es}}$, the vector $\mathbf{b}_{\text{comm}}$ shifts right (removing the first element) and a zero is appended.

If one or more buffers are empty, then the corresponding event has probability equal to zero, and the respective rates are removed from the denominator of the transition probabilities. If one or more buffer is full, then the transitions described above need to be modified to account for the fact that tasks/data cannot be moved to them and are, instead, erased.

6.4 Optimal Policy

The transition probabilities described in the previous section are conditioned on the action u chosen by the selector. We define, then, the policy ξ guiding such choice.

6.4.1 Performance Metrics and Cost Functions

We are interested in two key metrics: (a) the average time lapse between task arrival and departure from the system (total inference time) and (b) the task loss rate, that is, the probability that a task is sent to a full buffer. Given the complexity of the system, the definition of such metrics is non-trivial. Due to space constraints, we again provide an operational description that allows the derivation of the metrics.

Given the nature of the process, for each metric $r_i : \mathcal{S} \times \mathcal{S} \times \mathcal{U} \to \mathbb{R}$, that assigns a cost to the tuple (state, state, action), we associate the time-average of the metric r_i defined as

$$\bar{r_i} = \lim_{n \to +\infty} \frac{1}{n} \sum_{k=1}^{n} E\left[r_i(s_k, s'_k, u_k)\right]. \tag{6.18}$$

We now express the quantities needed to compute the performance metrics listed above. The average number of tasks successfully completed is the time average of the cost function:

$$r_1(s_k, s'_k, u_k) = \begin{cases} 1 & \text{if task successful at MD or ES} \\ 0 & \text{otherwise.} \end{cases} \tag{6.19}$$

Similarly we denote $r_2(s_k, s'_k, u_k)$ the average number of tasks discarded and $r_3(s_k, s'_k, u_k)$ the time passed.

The average total inference time is the ratio between the accumulated delay of all the images and the number of images whose analysis is completed. Therefore we can express it as

$$\frac{\bar{r}_3}{\bar{r}_1} = \frac{\lim_{n \to +\infty} \sum_{k=1}^{n} E[r_3(s_k, s'_k, u_k)]}{\lim_{n \to +\infty} \sum_{k=1}^{n} E[r_1(s_k, s'_k, u_k)]}. \tag{6.20}$$

6.4.2 Optimization Problem

Let us introduce $\hat{\mathbf{U}}$, the sequence of actions that minimizes the objective $R(\mathbf{U})$ over the space U^{∞} of all possible infinite action sequences under M_c linear constraints. Then we can express the problem as a Linear Fractional Program:

$$\begin{aligned}
\hat{\mathbf{U}} = {} & \arg\inf_{\xi} \frac{\sum_{s\in\mathcal{S}} \pi_{\xi}(s) r_3(s,\xi)}{\sum_{s\in\mathcal{S}} \pi_{\xi}(s) r_1(s,\xi)} \\
s.t. \quad & \beta_q \frac{\sum_{s\in\mathcal{S}} \pi_{\xi}(s) r_{c_n(q)}(s,\xi)}{\sum_{s\in\mathcal{S}} \pi_{\xi}(s) r_{c_d(q)}(s,\xi)} + \lambda_q \leqslant \gamma_q
\end{aligned} \tag{6.21}$$

where $q = 1, 2, ..., M_c$ enumerates the program constraints and $c_n(q), c_d(q)$ are indexes for the necessary cost functions appearing in the constraints. The problem stated above has multiple dependencies between the randomized stationary policy $\xi(s,u)$: $\pi_{\xi}(s)$, the stationary distribution of being in a given state, and the cost $r_i(s, \xi(s,u))$.

For this reason we define $\boldsymbol{\kappa} = g\boldsymbol{\omega} = g\pi_{\xi}(s)\xi(s,u)$, where the intermediate variables ω can be interpreted as the probability that the process is in state s and action u is taken. The change of variables $\boldsymbol{\kappa} = g\boldsymbol{\omega}$ is necessary to scale the magnitude of probabilities to the one of the cost functions, reaching the final problem formulation:

$$\begin{aligned}
\{\hat{\boldsymbol{\kappa}}, \hat{g}\} = {} & arg \min_{\boldsymbol{\kappa}, g} r_3^T \boldsymbol{\kappa} \\
s.t. \quad & (\beta r_{c_n} + (\lambda - \gamma) r_{c_d})^T \boldsymbol{\kappa} \leqslant \mathbf{0}_{M_c,1} \\
& \mathbf{1}_{1,A}\boldsymbol{\kappa} - g = 0 \\
& r_{c_d}^T \boldsymbol{\kappa} = 1 \\
& \mathbf{P}\boldsymbol{\kappa} = \mathbf{0}_{|\mathcal{S}|,1} \\
& g \geqslant 0, \kappa_{s,u} \geqslant 0 \quad \forall s \in \mathcal{S}, \forall u \in \mathcal{U},
\end{aligned} \tag{6.22}$$

where $A = |\mathcal{S} \times \mathcal{U}|$ is the cardinality of the state-action space, and where $\mathbf{0}_{m,n}, \mathbf{1}_{m,n}$ are matrices with m rows and n columns, with all elements equal to 0 and 1 respectively.

6.4.3 Optimal Policy

As demonstrated in [46], if the Markov process is *unichain*, that is, the whole state space is a single recurrent class, then at least one optimal memoryless stationary policy exists in the form of the following conditional probability

$$\xi(\mathbf{s}, u) = \Pr\{U_t = u \mid S_t = \mathbf{s}\}. \tag{6.23}$$

We use the problem formulated above to find the policy ξ through direct computation using the Linear Fractional Problem defined in the previous section. Applying the Simplex Algorithm to the problem in Eq. (6.22) we find $\kappa_{s,u}$, from which we can extract the policy using the following formulas:

$$\hat{\xi}(s, u) = \frac{\kappa(s, u)}{\sum_{u \in \mathcal{U}} \kappa(s, u)} \tag{6.24}$$

where if $\kappa(s, u) = 0$ the state is transient under the optimal policy, making $\hat{\xi}(s, u) = 1$ for one random action and $\hat{\xi}(s, u) = 0$ otherwise.

6.5 Results

In this section, we provide results discussing the performance of the dynamic selection we proposed with respect to edge computing. In order to find the parameters to use in solving LFP, we measured the inference time of DenseNet-169 over a large image classification dataset, on two devices: a Raspberry Pi 3b+ and Nvidia Jetson TX2. We created a split architecture using the head distillation technique described in [36] with a similarly shaped bottleneck. Note that we selected the smallest bottleneck size that preserved accuracy. The inference time and size of the data to be transmitted over the network are reported in Table 6.2.

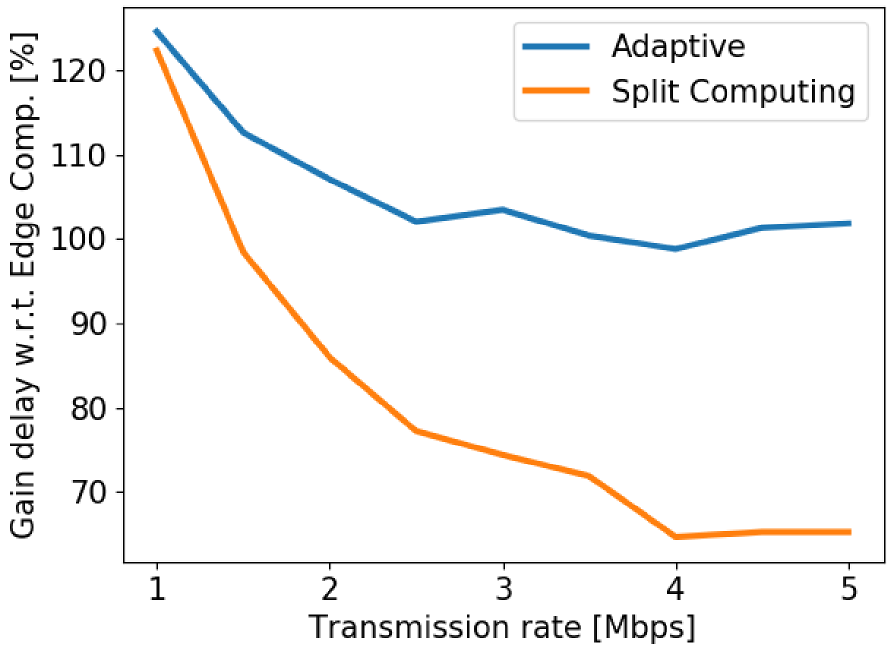


Figure 6.2: Delay ratio w.r.t. edge computing as a function of transmission rate.

Fig. 6.2 shows the ratio between the delay incurred using a fixed edge computing policy and the split computing. We remark that both adaptive and Split Computing outperform simple task offloading when transmission rates are below 2 Mbps. From the plot, it is clear that edge computing is the best strategy when data rates are sufficiently large (over 10 Mbps in the considered setting). The optimal policy implements a hybrid behavior. Whe the transmission rate is small, it mainly uses split computing. As the transmission rate increases, edge computing is used more often.

In Fig. 6.3, we plot the tradeoff between the number of successful tasks and the average delay. It can be observed how not only the delay is reduced using an adaptive approach, but also the number of successfully processed tasks increases. This is again due to the flexibility that split computing offers in a range of data rates where edge offloading is simply unusable due to the size of the full frame.

Table 6.2: Inference time and size of data to be transferred

	Mobile Device	Edge Server	Data
Full comp.	7.4 s	0.3 s	600 Kb
Split comp.	0.44 s	0.25 s	85 Kb

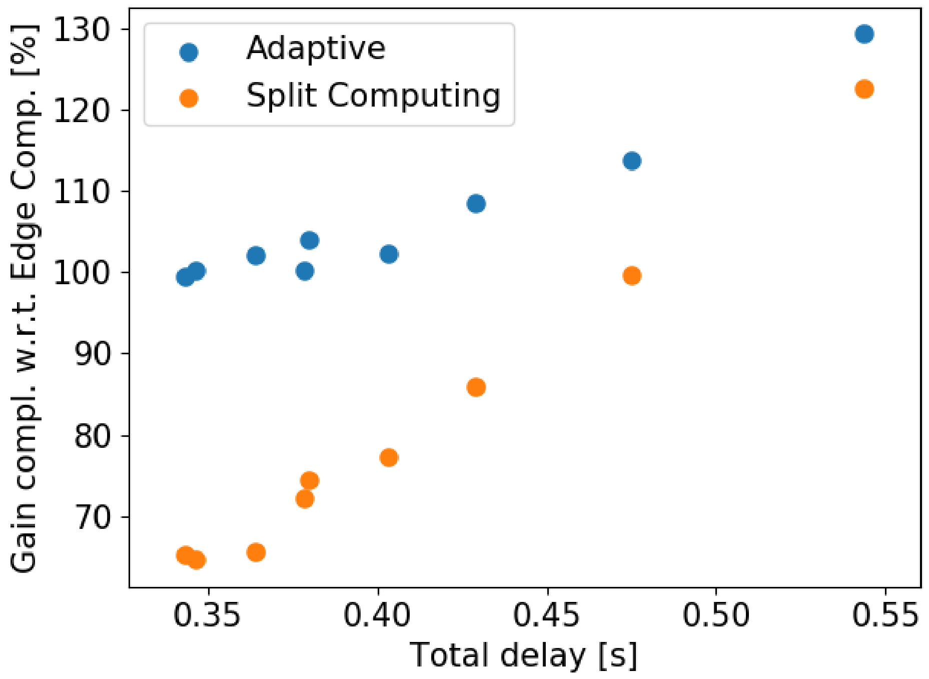


Figure 6.3: Gain in task computed w.r.t. edge computing, as a function of the total inference delay.

Chapter 7

Conclusions & Future Work

In this thesis we have shown different approaches to the problem of real-time analytics on high dimensional data. In particular we investigated how real-time video analytics used to control mobile autonomous systems, can be performed in a reliable and resilient way at the edge servers.

We presented a series of results derived from our real-world testbed. The development and utilization of our open-source framework HyDRAhelped us understanding what are the challenges in this space. We targeted reliability in edge offloading in MASs, proposing a reactive and a predictive approach. The reactive solution is described in Chapter 3, where we explored how an edge computing system can react in order to satisfy it's latency based computational requirements, while minimizing the impact on the network resources. The predictive solution is the topic of Chapter 4, where we show how pulling information from heterogeneous resources can help predicting the task delay. Moreover we have shown how to build intelligent agents on top of such prediction and how using Deep-Q Learning can further enhance the system's capabilities.

We then explored two different theoretical formulations regarding the same set of problems.

In Chapter 5 we show how formulating the problem using MDPs we can effectively uncover the characteristics of the optimal decision policies. In Chapter 6 we introduce an original formulation of the scheduling problem using heterogeneous tasks into a MDP. In this case we are also able to show how different networking scenario trigger different optimal policies to balance resource utilization and performance.

In all the techniques explored in this thesis we assume that the trajectory of the vehicle is independent from the decision making algorithm. If we were to relax this hypothesis, we would be able to create proactive approaches. In such scenarios the drone might still have a preferred region in which to be (for example $10m$ from the target), but if could optimize it's position in such region to increase the real-time performance loop (e.g. reducing the task delay).

Similar line of investigation could also be conducted on different data types: for example multi-spectral lidar is a technology used on drones in agricultural applications. Introducing offloading and a control loop to sample more often regions that have higher uncertainty could be beneficial in extending operation times and overall accuracy of classification.

Finally we suggest a different area to explore, where we consider heterogeneous tasks. In video analytics a recent idea is to alternate object detection and object tracking in order to provide some computational advantage at the embedded device. We envision how the same could be done in a distributed manner, including edge computing to improve allow the device to use the outcome of more precise models when applying tracking.

Bibliography

[1] Tensorflow detection model zoo. `https://github.com/tensorflow/models`, 2019.

[2] A. Alioua, H. eddine Djeghri, M. E. T. Cherif, S.-M. Senouci, and H. Sedjelmaci. Uavs for traffic monitoring: A sequential game-based computation offloading/sharing approach. *Computer Networks*, 177:107273, 2020.

[3] S. Baidya, Z. Shaikh, and M. Levorato. FlyNetSim: An Open Source Synchronized UAV Network Simulator Based on NS-3 and Ardupilot. In *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, pages 37–45, 2018.

[4] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *Proceedings of IEEE INFOCOM 2013*, pages 1285–1293, 2013.

[5] R. Bellman. Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences of the United States of America*, 42(10):767, 1956.

[6] L. Bertizzolo, S. D'Oro, L. Ferranti, et al. SwarmControl: An Automated Distributed Control Framework for Self-Optimizing Drone Networks. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 1768–1777, 2020.

[7] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[8] D. Callegaro. Hydra - distributed processing in heterogeneous cloud robotics systems. `https://github.com/uci-iasl/HYDRA`, 2019.

[9] D. Callegaro. SeReMAS: Resilience Through Task Replication in Mobile Autonomous Systems with Predictive Capabilities, 2021.

[10] D. Callegaro, S. Baidya, and M. Levorato. Dynamic distributed computing for Infrastructure-Assisted autonomous uavs. In *2020 IEEE International Conference on Communications (ICC): SAC Tactile Internet Track (IEEE ICC'20 - SAC-10 TI Track)*, Dublin, Ireland, June 2020.

[11] D. Callegaro and M. Levorato. Optimal computation offloading in edge-assisted uav systems. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2018.

[12] D. Callegaro and M. Levorato. Optimal Edge Computing for Infrastructure-Assisted UAV Systems. *IEEE Transactions on Vehicular Technology*, 70(2):1782–1792, 2021.

[13] B. Cao, L. Zhang, Y. Li, D. Feng, and W. Cao. Intelligent Offloading in Multi-access Edge Computing: A State-of-the-art Review and Framework. *IEEE Communications Magazine*, 57(3):56–62, 2019.

[14] X. Cao, J. Xu, and R. Zhangt. Mobile edge computing for cellular-connected uav: Computation offloading and trajectory optimization. In *2018 IEEE 19th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5. IEEE, 2018.

[15] S. E. Carpenter and M. L. Sichitiu. An obstacle model implementation for evaluating radio shadowing with ns-3. In *Proceedings of the 2015 Workshop on Ns-3*, WNS3 '15, page 17–24, New York, NY, USA, 2015. Association for Computing Machinery.

[16] J. Chen, S. Chen, S. Luo, Q. Wang, B. Cao, and X. Li. An intelligent task offloading algorithm (itoa) for uav edge computing network. *Digital Communications and Networks*, 2020.

[17] M. Chen, S. C. Liew, Z. Shao, and C. Kai. Markov approximation for combinatorial network optimization. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, 2010.

[18] F. Cheng, S. Zhang, Li, et al. UAV Trajectory Optimization for Data Offloading at the Edge of Multiple Cells. *IEEE Transactions on Vehicular Technology*, 67(7):6732–6736, 2018.

[19] N. Cheng, F. Lyu, W. Quan, C. Zhou, H. He, W. Shi, and X. Shen. Space/Aerial-Assisted Computing Offloading for IoT Applications: A Learning-Based Approach. *IEEE J. on Sel. Areas in Communications*, 37(5):1117–1129, 2019.

[20] N. Cheng, W. Xu, W. Shi, Y. Zhou, N. Lu, H. Zhou, and X. Shen. Air-ground integrated mobile edge networks: Architecture, challenges, and opportunities. *IEEE Communications Magazine*, 56(8):26–32, 2018.

[21] A. E. Eshratifar, A. Esmaili, and M. Pedram. Bottlenet: A deep learning architecture for intelligent mobile cloud computing services. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2019.

[22] I. J. Goodfellow, M. Mirza, D. Xiao, A. Courville, and Y. Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[23] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *Fourth International Conference on Learning Representations*, 2016.

[24] G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. In *Deep Learning and Representation Learning Workshop: NIPS 2014*, 2014.

[25] Q. Hu, Y. Cai, G. Yu, Z. Qin, M. Zhao, and G. Y. Li. Joint Offloading and Trajectory Design for UAV-enabled Mobile Edge Computing Systems. *IEEE Internet of Things Journal*, 6(2):1879–1892, 2018.

[26] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, volume 1, page 3, 2017.

[27] H.-J. Jeong, I. Jeong, H.-J. Lee, and S.-M. Moon. Computation offloading for machine learning web apps in the edge server environment. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1492–1499. IEEE, 2018.

[28] S. Jeong, O. Simeone, and J. Kang. Mobile edge computing via a uav-mounted cloudlet: Optimization of bit allocation and path planning. *IEEE Transactions on Vehicular Technology*, 67(3):2049–2063, 2017.

[29] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of ACM ASPLOS*, pages 615–629, New York, NY, USA, 2017. ACM.

[30] A. Karanika, P. Oikonomou, K. Kolomvatsos, and T. Loukopoulos. A demand-driven, proactive tasks management model at the edge. In *2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1–8, 2020.

[31] J. Kirkpatrick, R. Pascanu, et al. Overcoming Catastrophic Forgetting in Neural Networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017.

[32] A. Koubâa, A. Allouch, M. Alajlan, Y. Javed, A. Belghith, and M. Khalgui. Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey. *IEEE Access*, 7:87658–87680, 2019.

[33] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf. Pruning filters for efficient convnets. In *Fourth International Conference on Learning Representations*, 2016.

[34] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[35] B. Liu, H. Huang, S. Guo, W. Chen, and Z. Zheng. Joint computation offloading and routing optimization for uav-edge-cloud computing environments. In *2018 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computing, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, pages 1745–1752, 2018.

[36] Y. Matsubara, S. Baidya, D. Callegaro, M. Levorato, and S. Singh. Distilled split deep neural networks for edge-assisted real-time systems. In *Proceedings of the 2019 MobiCom Workshop on Hot Topics in Video Analytics and Intelligent Edges*, pages 21–26, 2019.

[37] Y. Matsubara and M. Levorato. Neural compression and filtering for edge-assisted real-time object detection in challenged networks. *arXiv preprint arXiv:2007.15818*, 2020.

[38] Y. Matsubara and M. Levorato. Split computing for complex object detectors: Challenges and preliminary results. *arXiv preprint arXiv:2007.13312*, 2020.

[39] M.-A. Messous, A. Arfaoui, A. Alioua, and S.-M. Senouci. A sequential game approach for computation-offloading in an uav network. In *GLOBECOM 2017-2017 IEEE Global Communications Conference*, pages 1–7. IEEE, 2017.

[40] M.-A. Messous, H. Sedjelmaci, N. Houari, and S.-M. Senouci. Computation offloading game for an uav network in mobile edge computing. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2017.

[41] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv*, 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[42] M. Mueller, N. Smith, and B. Ghanem. A Benchmark and Simulator for UAV Tracking. In *European Conference on Computer Vision*, pages 445–461. Springer, 2016.

[43] M. Narang, S. Xiang, W. Liu, J. Gutierrez, L. Chiaraviglio, A. Sathiaseelan, and A. Merwaday. Uav-assisted edge infrastructure for challenged networks. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 60–65. IEEE, 2017.

[44] Z. Ning, K. Zhang, X. Wang, L. Guo, X. Hu, J. Huang, B. Hu, and R. Y. K. Kwok. Intelligent Edge Computing in Internet of Vehicles: A Joint Computation Offloading and Caching Solution. *IEEE Transactions on Intelligent Transportation Systems*, pages 1–14, 2020.

[45] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine Learning in Python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[46] K. W. Ross. Randomized and past-dependent policies for markov decision processes with multiple constraints. *Operations Research*, 37(3), 1989.

[47] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[48] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, 2018.

[49] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[50] M. Schmittle, A. Lukina, L. Vacek, J. Das, C. P. Buskirk, S. Rees, J. Sztipanovits, R. Grosu, and V. Kumar. OpenUAV: A UAV Testbed for the CPS and Robotics Community. In *2018 ACM/IEEE 9th Int. Conference on Cyber-Physical Systems (ICCPS)*, pages 130–139. IEEE, 2018.

[51] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[52] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[53] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[54] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning. *CoRR*, abs/1509.06461, 2015.

[55] D. Wang, W. Li, X. Liu, N. Li, and C. Zhang. UAV Environmental Perception and Autonomous Obstacle Avoidance: A Deep Learning and Depth Camera Combined Solution. *Computers and Electronics in Agriculture*, 175:105523, 2020.

[56] F. Wang, J. Xu, X. Wang, and S. Cui. Joint offloading and computing optimization in wireless powered mobile-edge computing systems. *IEEE Transactions on Wireless Communications*, 17(3):1784–1797, 2017.

[57] Z. Yang, C. Pan, K. Wang, and M. Shikh-Bahaei. Energy Efficient Resource Allocation in UAV-enabled Mobile Edge Computing Networks. *IEEE Transactions on Wireless Communications*, 18(9):4576–4589, 2019.

[58] B. Zhang, G. Zhang, W. Sun, and K. Yang. Task Offloading with Power Ccontrol for Mobile Edge Computing Using Reinforcement Learning-Based Markov Decision Process. *Mobile Information Systems*, 2020, 2020.

[59] J. Zhang, L. Zhou, Q. Tang, E. C.-H. Ngai, X. Hu, H. Zhao, and J. Wei. Stochastic Computation Offloading and Trajectory Scheduling for UAV-assisted Mobile Edge Computing. *IEEE Internet of Things Journal*, 6(2):3688–3699, 2018.

[60] T. Zhang, Y. Xu, J. Loo, D. Yang, and L. Xiao. Joint Computation and Communication Design for UAV-Assisted Mobile Edge Computing in IoT. *IEEE Transactions on Industrial Informatics*, 16(8):5505–5516, 2020.

[61] F. Zhou, Y. Wu, R. Q. Hu, and Y. Qian. Computation Rate Maximization in UAV-Enabled Wireless-Powered Mobile-Edge Computing Systems. *IEEE Journal on Selected Areas in Communications*, 36(9):1927–1941, 2018.

[62] S. Zhu, L. Gui, J. Chen, Q. Zhang, and N. Zhang. Cooperative computation offloading for uavs: A joint radio and computing resource allocation approach. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 74–79, 2018.