UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Intelligenza Artificiale on the Edge: sviluppo di un'architettura per riconoscimento di attività umane indoor da stream video multi-camera

Artificial Intelligence on the Edge: development of an architecture for Indoor Human Activity Recognition from multi-camera video streams

Relatore:
Prof. Emanuele FRONTONI
Correlatori:
Sara MOCCIA, PhD

Tesi di Laurea Magistrale di:
Morris CAPPARUCCINI
Matr. 1090958

Anno Accademico 2020-2021

Sommario

La recente disponibilità di una sempre più elevata potenza di calcolo, abbinata all'evoluzione costante delle tecnologie ed al continuo progresso nel campo della ricerca, sta determinando un interesse sempre maggiore della comunità scientifica verso il campo dell'Intelligenza Artificiale (IA). Fra i vari task di questa disciplina spicca la *Human Activity Recognition* (HAR) la quale, grazie alla sua versatilità, sta prendendo sempre più piede in svariati ambiti applicativi come la domotica, i sistemi di sorveglianza e l'assistenza sanitaria verso gli anziani. Il progresso tecnologico, abbinato ad una maggiore disponibilità di risorse hardware, ha anche permesso lo sviluppo di nuovi paradigmi di computazione, tra i quali spicca l'*Edge Computing*, in grado di supportare al meglio l'esecuzione di questo tipo di task e di introdurli all'interno della vita di tutti i giorni.

Lo scopo di questo lavoro è la realizzazione di un framework di Deep Learning che si basa sul paradigma di elaborazione dell'Edge Computing in grado di classificare in tempo reale azioni umane in ambienti indoor di alto livello tramite l'utilizzo di reti neurali convoluzionali e ricorrenti che permettano di analizzare immagini provenienti da molteplici stream video real-time. In particolare è stata proposta una architettura di rete neurale per il riconoscimento di attività umane il cui modello realizzato è stato distribuito su hardware con risorse limitate, la Nvidia Jetson Nano, facente parte di una ampia rete di dispositivi edge. Gli stream video catturati dalle telecamere che compongono questa rete, inoltre, sono stati gestiti mediante due diverse configurazioni multi-threading al fine di ottimizzare l'uso delle limitate risorse hardware disponibili.

Questo lavoro di tesi è articolato come di seguito riportato:

Nel **capitolo 1** verrà introdotta l'intera pipeline di lavoro, presentando le tematiche trattate e gli obiettivi che il progetto si pone.

Nel **capitolo 2** verrà data una panoramica in merito al paradigma di computazione Edge Computing ed alla problematica relativa al task di HAR, ponendo un focus anche sui molteplici ambiti applicativi di entrambe le tematiche.

Nel **capitolo 3** verranno presentati i diversi utilizzi dell'Edge Computing presenti in letteratura, concentrandosi anche sulle diverse strutture realizzative di questo

paradigma di computazione, e gli approcci di Deep Learning più performanti atti alla risoluzione del task di HAR.

Nel **capitolo 4** verrà discussa l'architettura di Deep Learning proposta per il riconoscimento di attività umane, l'hardware su cui ottimizzare ed eseguire il modello di rete neurale addestrato e le differenti configurazioni di gestione delle telecamere che fanno parte della rete di dispositivi edge per la realizzazione del framework.

Nel **capitolo 5** verranno mostrati i risultati ottenuti sia dal modello di rete neurale proposto, sia le performance delle varie configurazioni adottate per la gestione dei molteplici stream video al variare del numero di telecamere che compongono la rete.

Nel **capitolo 6** saranno discussi nel dettaglio i risultati mostrati nel capitolo precedente, ponendo enfasi sui pregi e difetti del modello e delle configurazioni adottate.

Infine, nel **capitolo 7** saranno esposte le conclusioni tratte da quanto detto nei capitoli precedenti, sottolineando i risultati raggiunti e gli sviluppi futuri che potrebbero migliorare quanto realizzato in questo lavoro di tesi.

Summary

The recent availability of an increasing computing power, combined with the constant evolution of technologies and the continuous progress in the research field, is determining an increasing interest of the scientific community in Artificial Intelligence (AI). Among the various tasks of this discipline, *Human Activity Recognition* (HAR) stands out, which, thanks to its versatility, is becoming increasingly popular in various application areas such as home automation, surveillance systems and health care for the elderly. Technological progress, coupled with the increased availability of hardware resources, has also led to the development of new computing paradigms, including the *Edge Computing*, which can better support the execution of Artificial Intelligence tasks and introduce them into everyday life. The aim of this work is the realization of a Deep Learning framework based on the Edge Computing paradigm able to classify high-level indoor human activities in real-time through the use of Convolutional and Recurrent Neural Networks that will analyze images coming from multiple video streams. In particular, a neural network architecture have been proposed for Human Activity Recognition, whose trained model has been distributed on hardware with limited resources, the Nvidia Jetson Nano, which is part of a wider edge devices network. Furthermore, the video streams captured by the cameras that belong to the network were managed through two different multi-threading configurations in order to optimize the limited available hardware resources.

This thesis work is structured as follows:

In **chapter 1** the entire work pipeline will be introduced, presenting the subjects dealt with and the objectives of the project.

In **chapter 2** will be given an overview of the Edge Computing paradigm and the Human Activity Recognition task, focusing also on the multiple application fields of both topics.

In **chapter 3** will be presented the different Edge Computing usages available so far in the literature, focusing also on the different structures of this computation paradigm, and the most performing Deep Learning approaches to solve the Human Activity Recognition task.

In **chapter 4** the proposed Deep Learning architecture for the recognition of human activities, the hardware on which to optimize and execute the trained neural network model and the different handling configurations of the cameras that belong to the edge devices network will be discussed.

In **chapter 5** the results obtained from the proposed neural network model and the performances of the various configurations adopted to manage the multiple video streams varying the number of cameras in the edge devices network will be shown.

In **chapter 6** the results shown in the previous chapter will be discussed in detail, with emphasis on the strengths and weaknesses of the model and of the adopted configurations.

Finally, in the **chapter 7** the conclusions derived from what has been discussed in the previous chapters will be exposed, highlighting the results achieved and future developments that could improve what has been achieved in this thesis work.

Acknowledgements

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Artificial Intelligence (AI) is a very broad discipline with many different application domains that, in recent years, has become increasingly important and present in people's daily lifes. This is due to various factors such as, for example, the technological progress and the greater availability of computational resources that are enabling its great growth. It is therefore natural that the scientific community is interested in this branch of computer science and, as a result, more and more applications, which are able to exploit the different tasks that AI allows to perform, are being created.

The Human Activity Recognition (HAR) is one of the AI tasks that is emerging the most and which is becoming more and more central in many sectors such as, for example, domotics, video-surveillance, health assistance to the elderly and the monitoring of physical activities. The goal of HAR is to detect and classify high-level human activities from different types of data that allow the extraction and analysis of spatial and temporal features [1].

There are two main types of approaches for this type of activity and they are distinguished by the type of data used: the first type, on which this thesis work is based, includes all those methods that exploit optical data, such as, for example, images from camera streams, while the second type includes all those methods that, instead, use non-optical data like all those data extracted from sensors such as, for example, accelerometers, gyroscopes and motion sensors.

Due to the fact that most of the applications that perform the HAR task involve the real-time analysis of the above mentioned data and due to their nature and significance, one of the most appropriate paradigms for their development is Edge Computing: a computation model that foresees the processing of the data close to the place where them are collected. Using Edge Computing it is possible to reduce or eliminate latency times and guarantee real-time results by using small data

centers located near to the sensors[1]. These features have made this computation model one of the most widely used in several sectors, such as communication and security, and have allowed the realization and diffusion of technologies such as IoT.

The aim of this thesis is to use the Edge Computing paradigm for the realization of a Deep Learning framework that implements real-time Human Activity Recognition on a resource-limited hardware based on RGB images extracted from multiple camera streams. The entire work was carried out in collaboration with Inim Electronics S.r.l., a company based in Monteprandone (AP) that offers numerous solutions in the security field, which allowed the purchase and provision of all the necessary technologies to achieve the set objective.

[1] `https://www.criticalcase.com/it/blog/che-cose-edge-computing-definizione-e-vantaggi.html`

Chapter 2

Edge Computing and HAR

2.1 Edge Computing

The term Edge Computing refers to a distributed computational model in which data processing takes place close to the place where the data are collected[1]. The principle on which this computation model is based is to keep data collection and processing physically close, as opposed to the Cloud Computing computation model which instead aims to perform processing in a single central data center to which all data must be sent, as represented in figure 2.1[2].

Thanks to the increasing presence of smart devices, sensors and the decreasing cost of hardware components, Edge Computing is becoming more and more popular in everyday life, allowing the realization of a large number of applications in the fields of AI, home automation, healthcare, communication and security.

2.1.1 Edge Computing properties

Through the use of many small data centers located close to the sensors or the use of devices capable of collecting and analyzing data autonomously, Edge Computing computation model allows to handle a network composed of multiple nodes, i.e. the devices, guaranteeing a considerable reduction in network costs and bandwidth constraints, the decrease or elimination of delays in data transmission, the limitation of service errors and a better control of sensitive data transfers. In addition to all these advantages, there is also a reduction in loading times, greater closeness of

[1] https://www.stratus.com/it/edge-computing/

[2] https://www.kalrayinc.com/the-edge-computing-and-intelligent-systems-revolution

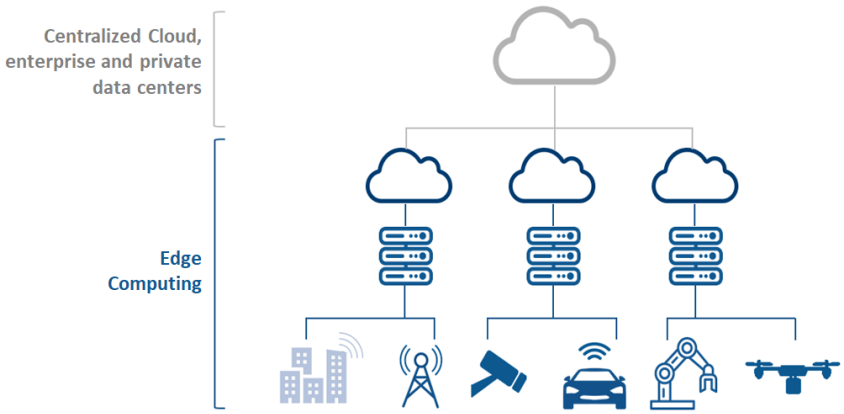

Figure 2.1: Computation model types

service delivery to users and simpler and more immediate control of the data[3].

Thanks to its decentralized structure and proximity to the data sources, Edge Computing can guarantee the following properties[4]:

- *Velocity*: a low response time is essential to complete certain types of tasks in many applications, such as the real-time video streams analysis. The use of Edge Computing, in these cases, is the best choice to carry out these types of tasks. Indeed, the presence of local mini data centers connected directly to the same network of sensors and cameras makes possible to greatly reduce latency and data transmission times.

- *Reliability*: Edge Computing systems make possible to guarantee different levels of reliability and alarm mechanisms in case of a malfunction in the various data centers. In fact, it is easy to detect a device that is not working properly and to repair or to replace it without affecting the operations of the others. Edge computing allows to keep active the several services offered by the various systems even in case of Internet connection problems, since all processing takes place locally.

- *Efficiency*: thanks to the proximity of the processing devices to the sensors, it is possible to obtain many advantages in terms of computational efficiency. This is the case both if the processing of the collected data is entirely carried

[3] https://www.redhat.com/it/topics/edge-computing/what-is-edge-computing

[4] https://en.wikipedia.org/wiki/Edge_computing

out within the local node, and if the considered node is part of a more complex system where this pre-processed data must then be further sent to a larger centralized data center for further analysis.

- *Scalability and modularity*: Edge Computing allows an easy extension of the processing capacity and variety of tasks that can be performed due to the wide availability of IoT devices and small perimeter data centers that will constitute additional nodes to be added to the existing network[5].

- *Security and privacy*: Having several decentralized processing devices has the great advantage that a potential attacker would not be able to obtain, in case of a security breach on one device, all the data coming from all sensors, as would happen in the case of Cloud Computing where such data must reach the single central data center. A protection policy must be implemented for each small data center in the local network or directly on the data collecting and processing device, if the latter is capable of performing both tasks. It is therefore necessary to use different protection mechanisms from those of Cloud Computing, where the central data center where all the processing takes place must be protected.

Due to its structure, however, Edge Computing entails some disadvantages: having many small data centers does not provide a lot of computing capacity to perform resource demanding tasks such as neural networks training and inference. It is therefore essential to make trade-offs between the task to be performed and the computational capacity of the edge devices. In case of particularly demanding tasks it is possible to consider the hybrid approach between Edge and Cloud computing by delegating the computation of the resource demanding operations to the centralized data center.

2.1.2 Edge Computing applications

Edge Computing is the optimal computation model for all those applications and tasks that cannot be efficiently managed by a centralized approach. Many of the limitations of this second approach, in fact, lie in the network requirements necessary for its operations or in the impossibility of satisfying the time constraints needed for the realization of the considered task.

[5] `https://www.cybersecurity360.it/soluzioni-aziendali/edge-computing-in-crescita-ecco-vantaggi-e-fronti-critici-per-le-aziende/`

Among the variety of applications that can be realized using this distributed model there are[6]:

- *Computer Vision applications*: real-time image analysis extracted by video streams using Artificial Intelligence algorithms. Examples of this applications are Classification, Face Recognition and Object Detection. Images are extracted from cameras located at the edge of the network and the common measurement unit used to evaluate the speed of the analysis are *Fps* (Frames per second) [2]. These tasks are widely used in several areas, such as the security field where image analysis must be completed in the shortest possible time to ensure that, in the event of anomalies occurring, the alarm is triggered and the necessary action taken. Having several data centers located near the cameras reduce the transmission time and latency.

- *Self-driving cars*: autonomous driving applications for vehicles. The data collected from all the sensors located on board of the car must be analyzed in real-time in order to be able to promptly perform various vehicle operations such as changing the trajectory and stopping. These are life concerning tasks so respecting time constraints is imperative and using edge devices helps to save time.

- *Financianl Transactions*: real-time analysis of financial transactions made via POS in dedicated micro data centers within bank branches. This is done in order to find anomalous transactions that need to be intercepted and blocked. Edge devices, in this case, performs the action before sending data to the central data center avoiding the overloading of requests.

- *Augmented Reality*: Use of an Edge computing platform to support augmented reality services by providing highly localized data specific to the user's point of interest such as the retail sector[7].

2.2 Human Activity Recognition (HAR)

The term Human Activity Recognition defines the discipline that deals with the recognition of activities, both simple and complex, of one or more agents in different

[6] https://www.internet4things.it/edge-computing/edge-platform/edge-computing-cosa-e-benefici/

[7] https://www.internet4things.it/edge-computing/edge-platform/edge-computing-ecco-i-casi-concreti-di-applicazione/

contexts of the real world starting from a series of observations based on the agents actions extrapolated from multiple nature sensorial data[8].

To understand this type of task and to better comprehend the difficulties which surround it, it is necessary to better define the concepts of *activity* and *action*. Among the various terminologies found in the literature [3], for this thesis work the following definitions for the concepts defined above were adopted:

- *Primitive (of an action)*: atomic movement that can be described at limb level such as bending or stretching a leg.
- *Action*: set of different *primitives* that form a complex and possibly cyclic movement of the body. An example of an action is the jump, an action composed of the *primitives* of bending and stretching of the legs and ending with the landing of the subject's body back to the neutral starting position.
- *Activity*: set of sequential actions that give an interpretation of the movement performed by the agent. With reference to the previous example, an activity is therefore that of jumping, composed of several jumps each representing a single action.

2.2.1 Types of Human Activity Recognition approaches

As already mentioned in the introduction of this thesis, the main approaches to Human Activity Recognition belong to one of the following two main categories:

- *Image-based approaches*: this type of approach is based on the use of RGB or RGBD cameras capable of capturing the actions carried out by agents in a non-invasive and continuous manner. The data streams collected will therefore be image streams from which it will be necessary to remove information not necessary to recognize the activity, such as the surrounding environment, by means of appropriate pre-processing operations.
- *Sensor-based approaches*: This category of approaches includes all methods that use non-optical data for HAR. Some of the most used devices for this type of task are the wearable sensors, i.e. sensors that are directly worn by the agent performing the action such as, for example, smartphones, fitness trackers and smartwatches[9]. These types of devices use sensors inside them such as accelerometers and gyroscopes that allow them to collect information about the movement and rotation carried out by the agent from which the

[8] `https://en.wikipedia.org/wiki/Activity_recognition`

[9] `https://madoc.bib.uni-mannheim.de/49914/1/thesis_compressed.pdf`

activities he or she is performing can be deduced. Also in this case, it is necessary to eliminate the noise produced by the sensors themselves through a pre-processing phase of the data obtained.

2.2.2 Human Activity Recognition phases

In the context of automatic methods for solving complex tasks, the one utilized in this thesis work is based on Deep Learning, in particular on the realization of a neural network model able to analyze the input data and to recognize the action performed.

For this type of methods, the HAR task consists of three different phases [4]:

1. *Data Collection*: in this phase, all the data necessary for training the neural network model responsible for carrying out the recognition task are collected.

2. *Model Training*: In this phase, the network model is trained on the basis of the data collected in the previous step.

3. *Activity Recognition*: In this phase the model trained in the previous phase is used on new data collected by the sensors in order to perform activity recognition.

All these steps are shown in figure 2.2 and will be described in detail below.

Data Collection

In order to train a neural network model, a large quantity of data is necessary since the greater the number of examples of a certain activity is submitted to the model in the training phase, the greater will be the capacity of the model to recognize that activity once trained. All data that can be used for training can come from different sensor types and are known as *Raw data*. Once the *Raw data* have been collected, they need to be cleaned of noise generated, by the sensor itself or by other types of noises, and saved in an easier form for the model to train. All these activities are included, as shown in fig. 2.2, in the *Pre-processing* sub-phase (1) and, once performed, it will be possible to use the pre-processed data for the actual training of the model. Subsequently, in the case in which the training of the model is of supervised type, it is then necessary to label these data with the type of activity that they represent. This activity is carried out in the second sub-phase of the *Data Collection* and is called, with reference to fig.1, *Activity labels* (2). After preparing all the necessary data, it is possible to train the model.

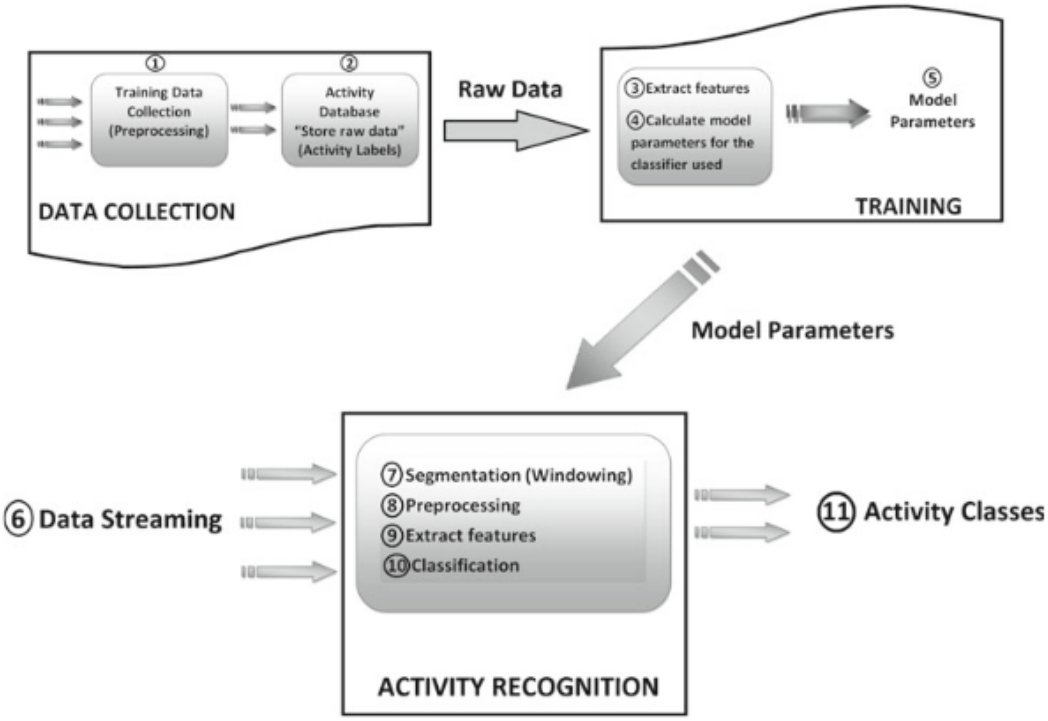

Figure 2.2: Human Activity Recognition phases.
Image taken from [4]

Model Training

There are two main types of training for neural network models:

- *Supervised Learning*: learning type that allows the model to make predictions based on ideal examples consisting of the input data to be analyzed and the expected output[10].
- *Unsupervised Learning*: learning type that allows the model to make predictions based on ideal examples classified according to common characteristics[11].

Usually, for the HAR task, the supervised learning is adopted. During this process, the model decomposes the input data provided in smaller sets of features (3) that will be exploited to modify the model *parameters* (4) in order to classify in the best way the represented activities. This operation will be carried out several

[10] `https://it.wikipedia.org/wiki/Apprendimento_supervisionato`

[11] `https://it.wikipedia.org/wiki/Apprendimento_non_supervisionato`

times until certain performances are reached, evaluated through appropriate metrics such as *Precision*, *Recall* and *Accuracy*. Once the optimal *parameters* (5) have been achieved, it will be possible to use the model to carry out the classification of the human activities starting from new data flows.

Activity Recognition

Since the analysis of continuous data streams is quite expensive in terms of computational resources, it is necessary to find techniques to reduce the complexity of the task. One of these techniques is the *Segmentation* technique (7), that allows to divide the input signal in smaller temporal segments [4]. Starting from these segments it is possible to carry out operations of pre-processing and feature extraction, in order to finally carry out the classification of the data and obtain in output the activity predicted by the model.

2.3 Human Activity Recognition applications

Nowadays, there are many applications that exploit this type of AI related task and most of them belong to one of the following three application domains [1]:

- *Smart-home Ambient Assisted Living*: this application domain encompasses all those technologies that make the living environment more interactive and intelligent as well as providing support, well-being and safety in everyday life[12]. An example of this are all those smart home applications that allow to learn, using Human Activity Recognition, the habits of residents to adjust different settings and provide them with additional comfort independently.

- *Health care*: this application domain includes all those applications that, by recognizing the activities performed, allow the monitoring of people's health conditions and that, in case of danger, are able to alert medical staff autonomously. An example of application in the health domain is the Fall Detection, which allows to detect the fall of the subject under consideration, typically an elderly person, in order to guarantee rapid assistance.

- *Monitoring and Surveillance*: the applications related to this application domain, of which the one presented in this thesis work is an example, make it possible to use the Human Activity Recognition task to detect potential harmful and/or illegal actions in the environment to be controlled and to activate the corresponding alarms. Currently, all these activities are carried

[12] `http://www.foritaal2012.unipr.it/ambient-assisted-living`

out by human operators who can make mistakes. Automated applications, on the other hand, make it possible to constantly monitor these places and avoid potential distraction errors by ensuring the highest level of vigilance.

2.4 Aim of the thesis

The aim of the following thesis is to create a Deep Learning framework based on Edge Computing computation paradigm in order to solve the task of Indoor Human Activity Recognition optimized for the execution on hardware with limited computational resources. This hardware, specifically the Nvidia Jetson Nano board, will represent a single node in a wider network of devices which will make inference on multiple video streams retrieved from fixed cameras. A neural network model will be realized, composed of a portion of a Convolutional Neural Network called backbone used for the features extraction, and of a Recurrent Neural Network Bi-LSTM (Bidirectional Long-Short Term Memory) with Dense layers for the feature analysis, both spatial and temporal. Three different backbones will be taken into account that are MobileNetV2, MobileNetV3Small and MobileNetV3Large. Once trained, the model will then be optimized to run on Nvidia Jetson Nano where it will be used for inference on data coming from multiple video streams. The whole framework will be managed through two different multithreading configurations, of which each available in two different versions, in order to find the one with the best performance evaluated both in terms of *Time* and *Fps*.

Chapter 3

State of the Art

In this chapter the different approaches available in literature useful to solve the Human Activity Recognition task using the Edge Computing computation paradigm are described and it is divided in two main sections: in section 3.1 both the main efforts to use Deep Learning (DL) techniques using Edge Computing paradigm and examples of complex DL applications that exploit this computation paradigm are described, while in section 3.2 the most performing approaches to solve the HAR tasks are presented, both for sensor-based activity recognition and image-based activity recognition.

3.1 Edge Computing and Deep Learning

Since the last few years researchers are focusing on the development of applications which, while exploiting the potential of Deep Learning techniques, are based on the Edge Computing paradigm, so to take advantage of the positive aspects of both. The biggest obstacle in the combined use of Artificial Intelligence and Edge Computing lies in the low availability of computational resources that edge computing devices make at disposal. This limitation poses inevitably a strict upper bound on the complexity of the DL models to use to address a certain task, affecting the effectiveness of the whole approach. In order to overcome this limitation, to enable the scientific community to develop AI applications on the edge, part of the research focused on the development of architectures that allow to realize Deep Learning applications even in resource-constrained settings. According to Chen et al. [2], three main architecture categories can be identified, as shown in figure 3.1, each of which includes architectures aimed at obtaining a higher overall speed in the models execution, i.e. the inference phase, and a more efficient management of the devices hardware resources:

1. *On-device computation*

2. *Edge server computation*

3. *Computing across edge devices*

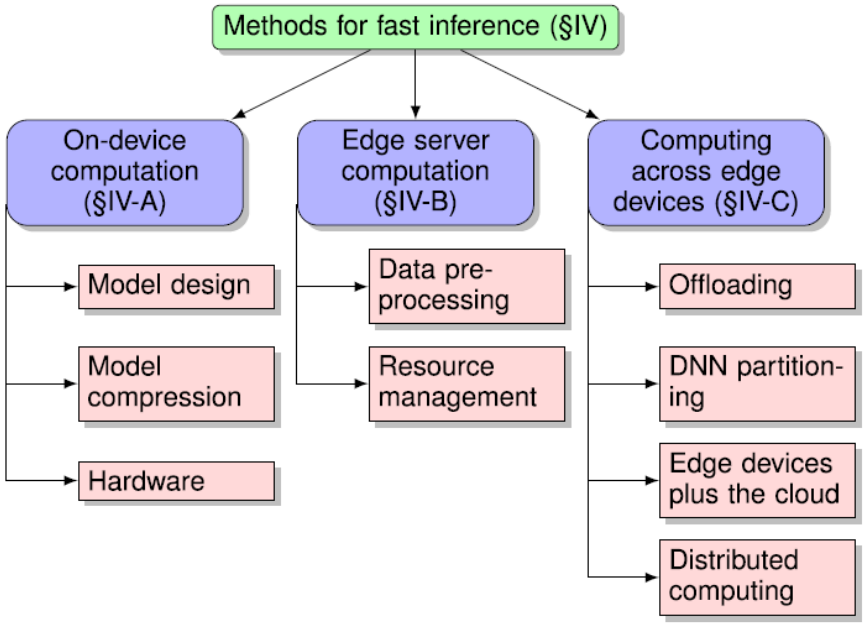

Figure 3.1: Taxonomy of DNN inference speedup methods on the edge
Image taken from [2]

More in details, the *on-device computation* category contains all the available methodologies, which can be used alone or in combination, for the optimization of the network model used within the edge device. Such methodologies are oriented mainly to the *model design*, *model compression* and *hardware optimization*.

As regards the *model design*, in order to avoid the use of hardware accelerators to run DL models on a single device, researchers effort focused on the optimization of these models, which must be as light as possible, to run on edge devices. There are several studies in the literature that allow the development of different types of network architectures with a reduced number of *parameters* that are, therefore, particularly suitable for devices with low computational performance. Among the others, [5] developed a lightweight CNN called MobileNet which is able to decompose the standard convolution operations for kernel-based features extraction into factorized convolution operations that greatly reduce the computational cost, [6] designed YOLO, a CNN model which is able to perform the objects classification and detection tasks in an image at the same time, and [7] developed the Squeezenet, a CNN able to maintain excellent *Precision* by reducing the dimensionality of the

input data and decreasing the filters size of the convolution operations. These networks can run on a single device but they have a great limitation caused by their lightweight structure, in fact the lighter is the model and the lower are the performances achieved, especially for complex tasks such as HAR.

Among the studies in the literature dealing with *model compression* there are approaches such as AdaDeep [8], a framework designed by Yao et al. able to choose the right combination of compression methods and to offer the best trade-off between dimensionality reduction and model *Accuracy* loss, DeepIoT [9], a unified compression method designed by Liu et al. for all common Deep Learning architectures suitable for IoT devices such as Intel Edison, and DeepMoon [10], a framework for running neural network models with multiple optimizations such as caching the results of convolutional layers combined with *parameters* quantization and more efficient matrix operations.

Hardware optimization-related works aim at exploiting all the optimizations made available by hardware boards manufacturers and by all the producers of Application-Specific Integrated Circuits (ASICs), i.e. special processing circuits that allow a substantial acceleration of the operations performed by certain types of applications. As an example, Zhang et al. (2020) [11] focused on the realization of a HAR as a Service (HARaaS) model using WiFi Signal in IoT-enabled edge computing environment. In Zhang et al. work, an IoT system has been implemented using a Raspberry Pi, connected to a WiFi network, that performs inference on data collected from IoT devices. In particular, the performances of a CNN model were tested first only using the Raspberry Pi resources, then using the Raspberry Pi and the Intel Neural Compute Stick 2 resources combined, which is a particular device developed by Intel that able to accelerate the execution of the neural models. Thanks to the use of this accelerator, it was possible to obtain more accurate results with a large performance boost and latency times reduction but, on the other hand, the cost of multiple computational devices has to be considered in the realization of this application.

Focusing on the *edge server computation*, instead of performing inference directly on the edge devices, it is possible to delegate the computation to the edge servers, possibly using security protocols for data exchange. In this way, the different nodes will send their data to a nearby server that will take care of their processing, returning the obtained results to them. Therefore, this category contains all the methods that make the sending and receiving of data to edge servers more efficient. An example of pre-processing aimed at sending data to a server is Glimpse [12], a real-time object detection system that allows the processing of the various frames both on the edge server and the edge device, balancing their number according to latency times. In [13], the authors developed Mainstream, a system where an optimal trade-off between *Accuracy* and latency time is calculated in order to choose which layers of the network models to share, using transfer

learning techniques, between the various requests. Despite the results obtained by the presented approaches, the main limitation of the edge server computation methodology regards the high requirements in terms of hardware resources to be realized. Thus, this type of computation is difficult to adapt for certain types of domain such as home automation.

The last category introduced in [2], namely *Computing across edge devices*, includes all the several methods that provide a more efficient management of the computational load between edge devices and edge servers. Indeed, instead of performing all the computation inside the device or the server, it is possible to perform the computation partially on both types of devices, using also an edge Cloud where possible. In [14], the authors developed a framework that allows to execute Deep Learning tasks both locally and remotely with the aim of balancing *Accuracy*, computational complexity and execution time, determining each time the best strategy to follow. In [15], a hierarchical approach to the computation partitioning between devices and servers is presented. In particular, the authors implemented a system with several DDNN (i.e., Distributed Deep Neural Networks) which allow inference with different levels of *Precision* and speed from edge devices to Cloud data centers as needed. One the other hand, this type of approach does not suite so much real-time applications cause the sending time to the Cloud is not negligible.

Taking advantage of the literature dealing with the presented architectures, researchers in the last years started to use the above mentioned Edge Computing approaches to develop Deep Learning applications, even for complex tasks such as Human Activity Recognition. Different types of DL structures were born to let complex AI tasks to be solved on low capabilities hardware. An example is the Li et al. [16] work, that proposed a structure composed by two layers, the *Edge* layer and the *Cloud* layer, to enable video recognition on an IoT application. Li et al. used the *Cloud* layer to train the model that was splitted among both several edge devices and edge servers. The division of the neural network layers was scheduled by a specific algorithm that splitted the lower layers of the model near the input data, i.e. on the edge devices, and the higher layers near the output data, i.e. on the edge servers, in order to reduce the data sending overhead among the servers. This approach reduced the network traffic from IoT devices to the edge servers and to the Cloud servers but it has the limitation that high resources devices are necessaries to execute the neural network model.

Another work that aimed to realize an Edge computing platform to perform complex AI tasks was made by Zhang ed al. (2018) [17]. In this work they proposed a scalable, consistent and low cost Edge computing platform for activity recognition in smart home applications. In particular, they realized a system composed of three layers, i.e. the *Cloud* layer, the *Edge* layer and the *sensor* layer, able to reduce the amount of data sent to the Cloud and to execute a CNN model on low cost devices.

Data collected from the *sensor* layer, which is composed by different sensors such as motion sensors, water sensors and phone sensors, are sent to the *Edge* layer where several Raspberry Pi devices share the execution of a complex Convolution Neural Network to analyze data. Finally, the *Cloud* layer is responsible for managing those Edge nodes and for the initial training of the CNN. This structure, showed in figure 3.2, is able to reduce the amount of data sent through the network and to reduce latency times and privacy issues and is also scalable and easily adaptable to the various sensors of the related layer. The CNN developed in this work achieved also good results on activity recognition. A great limitation of this approach, however, is that the CNN model can not be executed on a single Edge device, requiring the distribution of the different layers of the model among different devices and also requiring coordination efforts for its parallel execution.

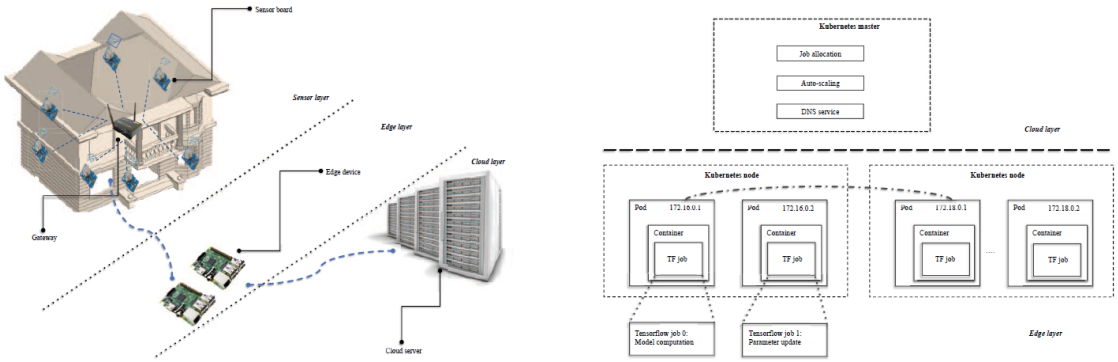

Figure 3.2: Three layer Edge computing platform representation
Image taken from [17]

Finally, the last work related to the execution of complex neural network models on edge devices cited in this thesis work is the one of Aishwarya et al. [18], which realized a light CNN based architecture able to analyze inputs captured from cameras and elaborated by a low complex foreground extraction algorithm and to classify and localize from them base human activities, such as walking, pushing and pulling. To reduce the complexity of the inputs to be passed at the CNN model, Aishwarya et al. used a background subtraction technique able to remove background pixels from images acquired by the cameras, which have to be fixed, to avoid unnecessary elaboration. Despite the lightness of this approach, it has two main issues regarding the above mentioned technique: the first is that the model can recognize only low level activities and the second is that the foreground extraction algorithm is particularly exposed to errors made by environment changes such as illumination variations.

3.2 Human Activity Recognition

Human Activity Recognition, given the growing interest in the scientific community, is object of many studies in the literature that aim to develop a performing neural network model able to solve this task. Over the years, various architectures that produced interesting results have been proposed. In this section, several examples of this architectures will be given, divided according to the type of used approach: sensor-based or image-based.

Sensor-based approach

The HAR task can be associated with the classification task, in fact the first approaches that aimed to solve it used Machine Learning techniques such as the Naive Bayes and Random Forest classification algorithms. These methods, however, do not consider temporal features and are quite limited. More elaborate methods, instead, involve the use of Deep Learning models that are able to analyze both spatial and temporal features for a more accurate and precise classification of the detected actions. Liciotti et al. [1] propose a comparison of different architectures based on Long Short-Term Memory (LSTM) neural networks, both monodirectional and bidirectional, to solve the Human Activity Recognition task in an Ambient Assisted Living scenario, using data collected by several types of typical smart homes sensors. The peculiarity of the LSTM networks is that they use a particular principle called Constant Error Carousel (CEC) which, through memory cells managed by a gating mechanism, allow to learn the temporal dependencies between the data.

All these models were tested on five annotated datasets among those available in CASAS and all the obtained results were then compared with those obtained by one-dimensional Convolutional Neural Networks and by the main Machine Learning techniques used in the literature and tested on the same datasets. The tests showed that LSTM networks lead to a substantial increase in *Accuracy* with respect to both traditional Machine Learning techniques, with an increase of almost six percentage points, and one-dimensional Convolutional Neural Networks, with an increase of over seven percentage points. In addition, among all the LSTM architectures tested, the most performing one was the bidirectional version, which allows time dependencies to be analyzed in both directions rather than in a single direction. Despite the results obtained by proposed approach, instead, a main limitation is the reduced number of activities on which the models were trained. Indeed, with an increasing number of activities, the use of sensor data could generate confusion for human activities sharing the same low level movements, e.g. dry hair and brushing teeth.

Image-based approach

Among the many works in the literature that solve the task of the Human Activity Recognition through the use of images that have influenced the following thesis work, it is mentioned the study of Baccouche et al. [19] who have created a sequential Deep Learning model composed of a Convolutional Neural Network (CNN) 3D, which allows the features extraction from videos, and a Recurrent Neural Network (RNN), composed of an LSTM layer which allows the learning of spatio-temporal features extracted by the previous part of the model to classify the action performed.

Both the training and testing of this network model were carried out on the KTH dataset, which contains videos of six different types of actions, i.e. walking, jogging, running, boxing, hand-waving and hand-clapping, performed in four different scenarios by 25 different people. The network model was tested on two different versions of this dataset: the first version, called KTH1, is composed of the original videos of the dataset containing actions repeated three or four times by the same person separated by some empty frames, while the second version, called KTH2, is composed of shorter videos where the action is performed once by a single person.

The outcomes showed that the proposed network model produced results comparable to those found in the literature on both versions of the dataset. The introduction of the LSTM layer also increases the *Accuracy* of the model compared to the use of a majority voting system on video sequences combined with the 3D CNN by about three percentage points. This confirms the validity of the proposed model and the use of LSTM networks for solving the HAR task. The main limitation of this approach is that the model still uses 3D convolutions, which are complex matrix operations, that makes the entire model too much resource demanding for the execution on low capabilities hardware such as edge devices.

Chapter 4

Materials and Methods

This chapter will discuss all the materials and methods used to solve the Human Activity Recognition task using the Edge Computing computation paradigm and it is divided into seven main sections: in section 4.1 the neural network architecture used for HAR is presented; in section 4.2 is described the training of the model, the parameters used to train it and the dataset on which it was trained; section 4.3 describes the Nvidia Jetson Nano board, that is the hardware on which the neural network model was be used to solve the Human Activity Recognition task; in section 4.4 TensorRT is presented, an SDK that allows the optimization of neural network models to be executed on Nvidia architectures; in section 4.5 is reported the configuration procedure of the operating environment and the deployment of the neural network model on the Jetson Nano board; section 4.6 describes the video cameras used to capture the video streams from which human activities can be recognized; section 4.7, that is the last section of this chapter, describes the functioning of the framework created using the above-mentioned tools and all the different configurations adopted for management of the several video streams collected by the cameras on which inference has to be made.

4.1 Proposed Architecture

The neural network architecture proposed in this thesis to solve the Human Activity Recognition task is the composition of the first part of a Convolutional Neural Network used for the features extraction, called backbone, and a Bi-LSTM Recurrent Neural Network linked to fully connected layers for the classification of the performed action represented in the input data. Three different backbones were used, for which the model architecture differ, which are MobileNetV2 [20], MobileNetV3Small and MobileNetV3Large [21]. The choice of this first part of the model architecture was made taking into account the computational capacities of the board on which they

have to be executed. In particular, the MobileNet architectures were chosen because they are particularly lightweight and because they ensure good performances even on hardware with limited resources. It is also pointed out that, in order to be able to use the Bi-LSTM network and the classifier presented in the following subsection, the default fully connected layers of all MobileNet neural networks have been removed.

4.1.1 Backbones

MobileNetV2

MobileNetV2 is a CNN based on the MobileNetV1 architecture whose working principle is founded on a layer module called *Inverted residual with linear bottleneck* [20]. Sandler et al., for the realization of this architecture, kept the simplicity of the MobileNetV1 structure, increasing at the same time the performances thanks to the introduction of this innovative type of module and replacing the standard convolution operators with two different layers that allow a reduction of the computational complexity. The final architecture of MobileNetV2 is shown in figure 4.1.

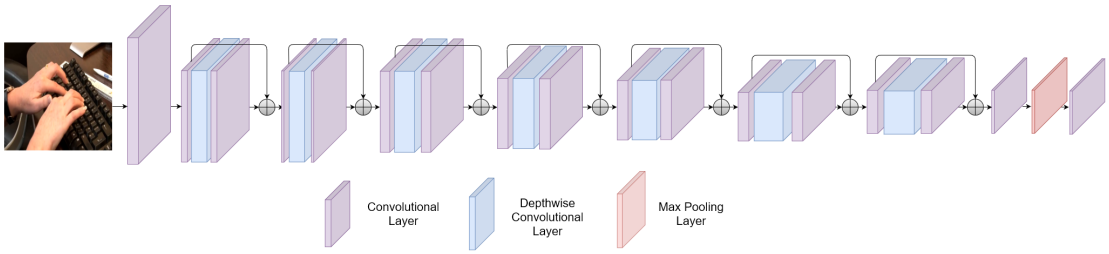

Figure 4.1: MobileNetV2 complete layers architecture

In a CNN, the main operation that allows feature extraction is *convolution*. In this type of networks, the *convolutions* are operations that compare a filter with the input images to return an output based on the similarity between the portion of image under examination and the used filter. An example of output obtained from the comparison between an input image and two different filters is shown in figure 4.2[1].

Standard convolutions, realized through appropriate matrix products, are however computationally onerous, therefore in this architecture they have been replaced by *Depthwise convolutions*, which are composed of a factorized operator that can be

[1] `https://www.nonteek.com/it/machine-learning-parte-iii-reti-neurali-convoluzionali/`

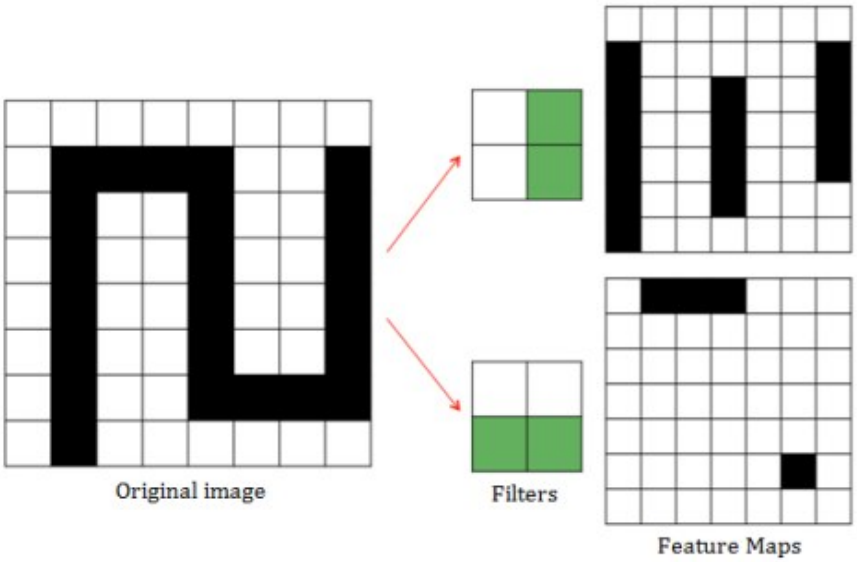

Figure 4.2: Neural networks convolution

realized through two distinct layers: the *Depthwise convolution* layer, which applies a single filter for each input channel, and the *Pointwise convolution* layer, which uses 1×1 convolutions in order to build new features using linear combinations of the input channels. Sandler et al. have shown that this type of operator achieves a computational cost that is about 8-9 times lower than using *Standard convolutions* at the cost of a small reduction in *Accuracy*. As can be seen in figure 4.1, the central blocks of MobileNetV2 architecture are the *Inverted residuals with linear bottlenecks*, realized by previously mentioned layers. To describe them, however, it is first necessary to introduce what *Residual blocks* are: *Residual blocks*, in general, are blocks that allow to connect, by means of skip connections, the beginning and the end of a *Convolutional layer block*[2], as shown in figure 4.3.

This type of structure is adopted mainly for two reasons: to avoid the *Vanishing Gradient* problem and to limit the saturation of *Accuracy* (Degradation)[3].

The blocks used in the MobileNetV2 architecture are called *Inverted residual blocks* because, instead of connecting blocks with a large number of *parameters* bypassing the blocks with a small number of *parameters*, they connect through a skip connection the blocks with a small number of *parameters* bypassing the others, as shown in Figure 4.4.

In order to increase the number of *parameters* in the intermediate layers, a 1 *convolution* is used, followed by a 3×3 *Depthwise convolution*, which allows to

[2]`https://towardsdatascience.com/mobilenetv2-inverted-residuals-and-linear-`

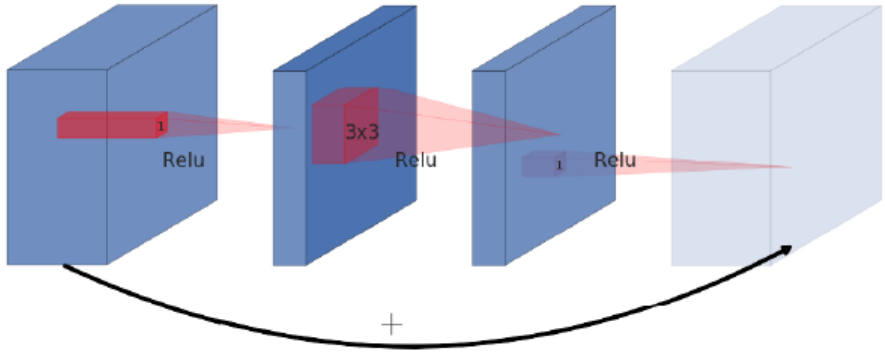

Figure 4.3: Residual block
Image taken from [20]

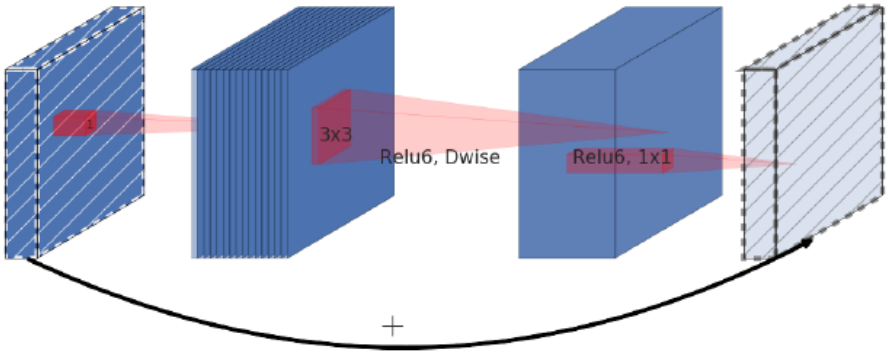

Figure 4.4: Inverted residual block
Image taken from [20]

reduce these *parameters* again. Finally, in order to implement the skip connection between the input of the *Convolutional block* and the output, a further 1×1 convolution is introduced. To avoid losing information during this last operation, it is necessary to introduce a linear activation function, called linear bottleneck, which allows a correct transmission of the information. The final architecture of the MobileNetV2 is therefore described in the table shown in figure 4.5.

This architecture is more advantageous than the previous one because the total number of parameters in the network is reduced, so that the total dimensionality of the network is lower, and at the same time the performance in terms of *Accuracy* are comparable.

---

bottlenecks-8a4362f4ffd5

[3] https://en.wikipedia.org/wiki/Residual_neural_network

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Figure 4.5: MobileNetV2 complete structure
Image taken from [20]

MobileNetV3

MobileNetV3 is a network architecture that evolved from that of MobileNetV2 and MnasNet [21]. What has been done in the MobileNetV3 architectures is to replace the initial and final layers of the MobileNetV2, which are the most computationally onerous, and to introduce the non-linear *Hard swish* activation function which provides several advantages including, among others, a greater processing speed. The modifications made to the initial layer, containing the first set of 32 3×3 filters, include to change the nonlinear activation function used by MobileNetV2 with the *Hard swish* activation function and to reduce the initial number of *convolutional filters* from 32 to 16. This made possible to speed up the operations of the first part of the network and to reduce its computational complexity. The changes to the last block, on the other hand, are even more substantial: first of all, the last layer of 1×1 *convolutions* has been positioned after the 7×7 *Average pooling* layer in order to reduce latency times. This also removes the 3×3 *Depthwise convolution* layers and the subsequent *Normalization layer*. These changes are shown in figure 4.6.

The latest changes to the structure of the original MobileNetV2 involve the introduction of the non-linear *Hard swish* or *h-swish* activation function. This particular activation function replaces the *ReLU6*, i.e. *Rectified Linear Unit 6* used in MobileNetV2, in the second part of the model. Since the output of the *ReLU6* had values contained in the range [0 - 6] in order to achieve good performance in

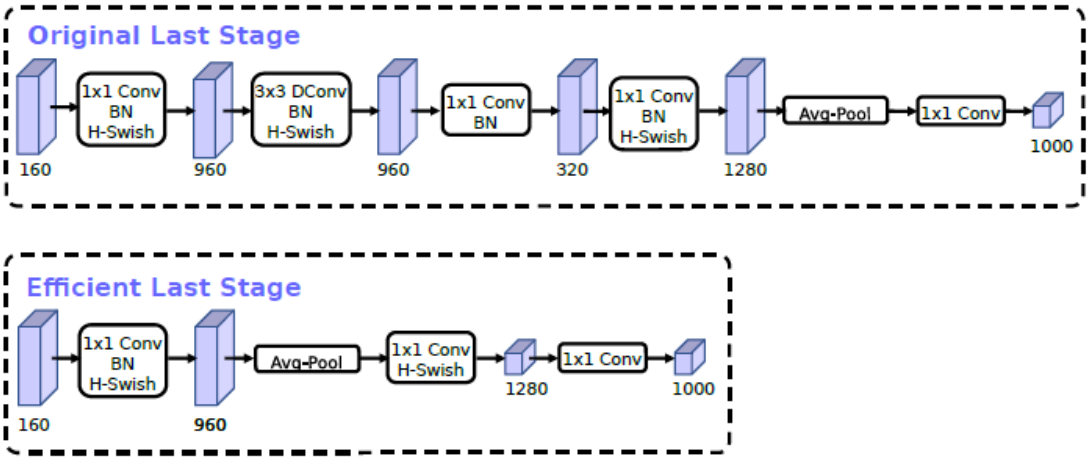

Figure 4.6: MobileNetV3 last layer optimization
Image taken from [21]

terms of *Accuracy*, the *h-swish* function was formally defined as:

$$h\text{-}swish[x] = x\frac{ReLU6(x+3)}{6}$$

The use of this activation function maintains the *Accuracy* levels of the *ReLU6* activation function but introduces great advantages in terms of computational efficiency and in future quantization perspective.

There are two main neural network architectures based on the previous mentioned changes described in [21]: MobileNetV3Large and MobileNetV3Small. The complete structures of this networks are shown in figures 4.7 and 4.8.

Compared to the previous MobileNetV2 architecture, these networks are faster and they have more efficient execution performances. MobileNetV3 Large has also similar *Accuracy* values compared to those of MobileNetV2, while MobileNetV3Small has a greater *Accuracy* loss but gains a lot in terms of lightness and execution speed.

4.1.2 Bi-LSTM and Classifier

The second part of the proposed model consists of a Bidirectional Long Short Term Memory (Bi - LSTM) neural network and a classifier, made up of *Dense* layers able to determine which action is performed on the basis of the features extracted from the previous layers of the model. Figure 4.9 shows the complete composition of this second part of the model.

Excluding the *Pooling*, *Batch Normalization* and *Dropout* layers, which are responsible respectively for reducing the size of the feature maps, normalizing the

Input	Operator	exp size	#out	SE	NL	s
$224^2 \times 3$	conv2d	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	-	RE	1
$112^2 \times 16$	bneck, 3x3	64	24	-	RE	2
$56^2 \times 24$	bneck, 3x3	72	24	-	RE	1
$56^2 \times 24$	bneck, 5x5	72	40	✓	RE	2
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 5x5	120	40	✓	RE	1
$28^2 \times 40$	bneck, 3x3	240	80	-	HS	2
$14^2 \times 80$	bneck, 3x3	200	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	184	80	-	HS	1
$14^2 \times 80$	bneck, 3x3	480	112	✓	HS	1
$14^2 \times 112$	bneck, 3x3	672	112	✓	HS	1
$14^2 \times 112$	bneck, 5x5	672	160	✓	HS	2
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	bneck, 5x5	960	160	✓	HS	1
$7^2 \times 160$	conv2d, 1x1	-	960	-	HS	1
$7^2 \times 960$	pool, 7x7	-	-	-	-	1
$1^2 \times 960$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

Figure 4.7: MobileNetV3Large complete structure
Image taken from [21]

output of the *Bi-LSTM* layer and deactivating a percentage of the neurons that make up the network in order to obtain a better generalization of the model, in this second part of the model there are two main layers: the *Bi-LSTM* and the *Dense*.

Bidirectional Long Short-Term Memory

The *Long Short Term Memory networks* [22], or *LSTM*, are a type of RNN able to learn the temporal dependencies that exist among the input data[4]. As it was reported in Chapter 3, this type of RNNs are widely used for the resolution HAR task, as it is necessary to take into account the dependencies and the interconnections between the *primitives* that compose the various actions. Since these relations can continue for very long times, this type of network is particularly suitable for their memorization since, unlike the standard Recurrent Neural Networks, it does not struggle to learn the long term dependencies.

The operating principle of LSTM networks is based on the concept of *State cells*, whose concatenation gives the network itself. Each state cell is composed of four

[4] `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Input	Operator	exp size	#out	SE	NL	s
$224^2 \times 3$	conv2d, 3x3	-	16	-	HS	2
$112^2 \times 16$	bneck, 3x3	16	16	✓	RE	2
$56^2 \times 16$	bneck, 3x3	72	24	-	RE	2
$28^2 \times 24$	bneck, 3x3	88	24	-	RE	1
$28^2 \times 24$	bneck, 5x5	96	40	✓	HS	2
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	240	40	✓	HS	1
$14^2 \times 40$	bneck, 5x5	120	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	144	48	✓	HS	1
$14^2 \times 48$	bneck, 5x5	288	96	✓	HS	2
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	bneck, 5x5	576	96	✓	HS	1
$7^2 \times 96$	conv2d, 1x1	-	576	✓	HS	1
$7^2 \times 576$	pool, 7x7	-	-	-	-	1
$1^2 \times 576$	conv2d 1x1, NBN	-	1280	-	HS	1
$1^2 \times 1280$	conv2d 1x1, NBN	-	k	-	-	1

Figure 4.8: MobileNetV3Small complete structure
Image taken from [21]

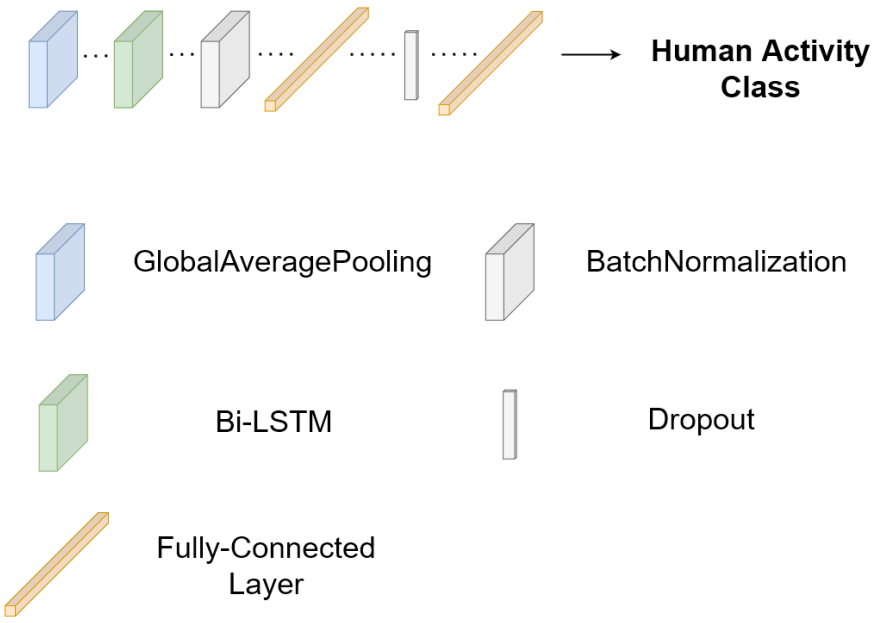

Figure 4.9: Bi-LSTM and Classifier

layers, coloured in yellow in figure 4.10, which allow the realization of a mechanism based on three gates, i.e. the *Input gate*, the *Output gate* and the *Forget gate*,

which allows the manipulation of the state of the single cell and the information that is forwarded to the other *State cells*. The complete representation of a *LSTM cell* is shown in figure 4.10[5], while the caption for the symbols used within it is shown in figure 4.11[6].

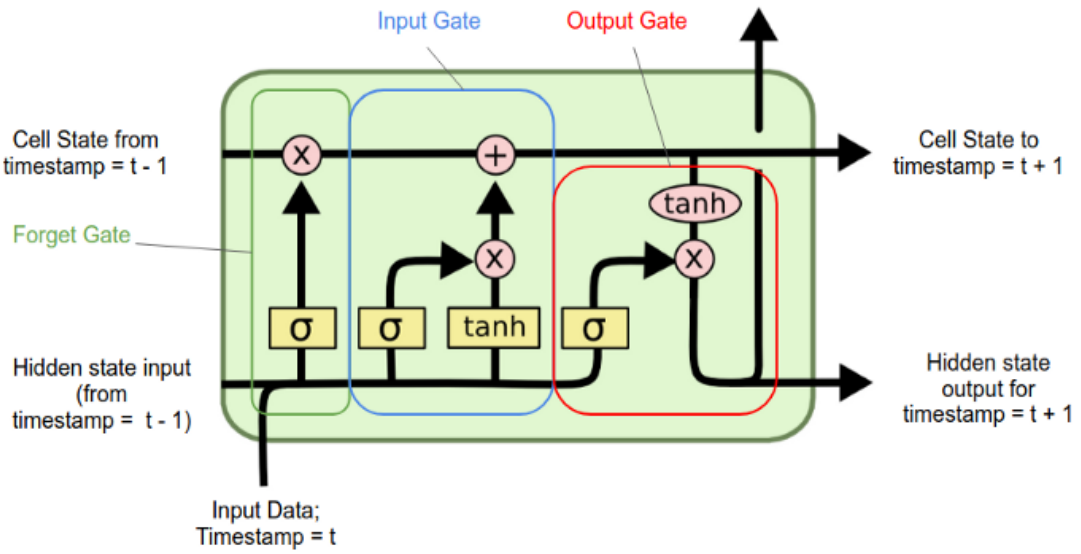

Figure 4.10: Representation of a LSTM Cell

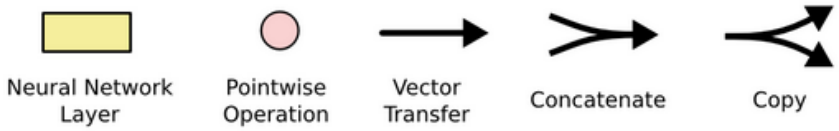

Figure 4.11: LSTM Cell representation legend

The first gate through which information passes, called *Forget gate* and highlighted in green in figure 4.10, allows to define which information to keep and which to forget in their flow from one cell to another. This gate, composed of a layer with a sigmoid σ activation function, evaluates whether an information is useful or not in the current cell and returns in output a number from 0 to 1 according to how much it is necessary to forget or remember the information. This value

[5] https://medium.com/analytics-vidhya/lstms-explained-a-complete-technically-accurate-conceptual-guide-with-keras-2a650327e8f2

[6] https://colah.github.io/posts/2015-08-Understanding-LSTMs/

is multiplied, in fact, to the information by means of a floating point operation, making sure that it is taken into account or not.

The second gate, called *Input gate* and highlighted in blue in figure 4.10, allows instead the insertion of new information and the updating of the cell state. This is done through two layers: one with a *tanh* activation function, which is in charge of creating new information to be added to the state, and one with a sigmoid σ activation function, which takes care of deciding how much to update each piece of information. The output from these two layers is then combined to create a global update of the cell state.

Finally, the last gate, called *Output gate* and highlighted in red in Figure 4.10, defines the information to be passed to the next cells. Not all the information stored in the previous state are passed on, but only a portion. The information collected is therefore processed through the *tanh* operation, to regularize the values from -1 to 1, and filtered by the layer with a sigmoid σ activation function that decides, according on the returned output, which information have to be passed to the next cell. The final representation of a LSTM Recurrent Neural Network composed by three cells is shown in figure 4.12.

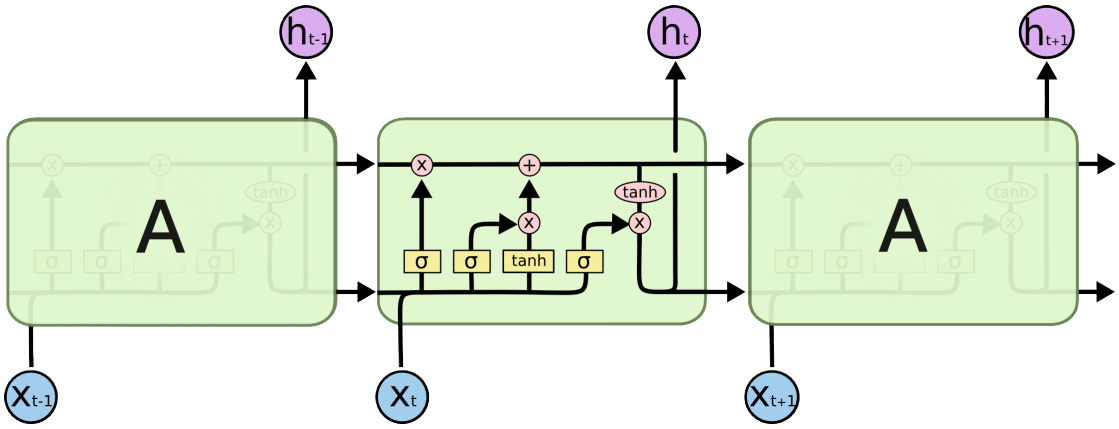

Figure 4.12: LSTM Recurrent Neural Network representation

In this thesis work is used a Bidirectional LSTM network, whose main feature is that the information is not only flowed in one direction through the cells but also in the opposite direction. This allows this type of network to learn the temporal dependencies between data both forwards and backwards, a characteristic that in the HAR task increases the overall performance of the network.

Dense layer

Dense layers are particular layers of a neural network whose *neurons* have connections with all those belonging to the previous layer[7]. This causes that each neuron of the dense layer elaborates all the inputs, that are the features, coming from the previous layer. The name *Dense* derives from the fact that the number of connections in this type of layer is very high. This type of layer allows, therefore, to analyze both the spatial and temporal features extracted from the previous layers and to obtain an output from its own *neurons* based on a particular activation function thus solving the classification task.

The final output of the model, in fact, is constituted by the label of the action associated to the *neuron* that has been most activated, that is the one that identifies the action to which, with greater probability, belongs the set of features detected in the previous portions of the network.

4.2 Model training

The training of the model deriving from the previously proposed architecture, one for each of the backbones presented in the previous section, was carried out on Google Colab, a platform made available by Google that allows the execution of Python code on the Cloud and that offers considerable hardware resources that made the training process easy. The model *training* is a learning process that allows the updating of the internal *parameters* of the neural network, i.e. the *weights*, on the basis of a special function, called *loss function*, which evaluates the improvement or worsening of the performance of the network in carrying out its task calculated on a chosen metric.

To make the model be able to recognize human activities, its training was carried out on the Kinetics dataset using the technique of *Fine Tuning*, a practice that allows the use of models already trained on other datasets in order to have initial *weights* already set. This technique allows the training process to be more efficient, compared to that of untrained models, as randomly set initial *weights* require a greater number of training *epochs*, i.e. the number of times the model examines the entire training dataset at its disposal, to obtain good results. In particular, the pre-trained *weights* on the Imagenet dataset of the three backbones were used, which are already available in the Keras API and can be set through the appropriate parameter[8].

[7] `https://heartbeat.fritz.ai/classification-with-tensorflow-and-dense-neural-networks-8299327a818a`

[8] `https://keras.io/api/applications/mobilenet/`

Kinetics

Kinetics is a set of datasets containing up to 650,000 high-quality videos comprising 400, 600 or 700 classes of human actions[9]. Each video contains a human action performed by one or more people for about 10 seconds.

For the training of the neural network model carried out in this thesis work, a subset of Kinetics comprising a total of 20 classes was selected and divided into two parts: a first part equal to 92% of its size, which was further divided into 80% for the training set and 20% for the test set, and a second part equal to 8% of its size entirely dedicated to the validation set. Since the proposed network architecture requires a set of images as input, a number of frames equal to 10 were extracted from each video in the dataset at regular time intervals. These frames will be then processed by the network together with the label of the action that is carried out within them in the *Supervised learning* process. Some examples of extracted frames from Kinetics dataset are shown in figure 4.13.

Training parameters

The proposed network model, regardless of the type of backbone adopted, uses as input a set of 10 frames of size $224 \times 224 \times 3$ called *batch* and was trained with *parameters* having Floating Point 32 (FP32) precision for a number of *epochs* equal to 80 using the *Adagrad* optimizer, whose *Learning rate*, i.e. the update rate of the *weights*, was set to 0.001.

The *optimizer* is a particular algorithm that determines how the weights of the model should be updated during training in order to reduce a cost function, called *Loss Function* which in this thesis work corresponds to *Validation Accuracy*, specifically chosen to monitor the progress of the model's performance[10]. For the HAR task, given the large variety of actions to be recognized, the *Adagrad* optimizer was chosen. *Adagrad*, whose name is an abbreviation of *Adaptive gradient*, is an optimizer based on gradient descent principle that performs smaller updates on parameters corresponding to very frequent features and larger updates on parameters corresponding to less frequent features. This allows better detection of key features for recognizing certain types of actions and greater robustness in the presence of heterogeneous data.

Further training parameters include the use of the activation function $ReLU$, which stands for *Rectified Linear Unit*, used to manage the activation of neurons, and a *Batch Size* of 16, i.e. 16 sets of 10 images (frames) of size $224 \times 224 \times 3$ were

[9] `https://deepmind.com/research/open-source/kinetics`

[10] `https://towardsdatascience.com/optimizers-for-training-neural-network-59450d71caf6`

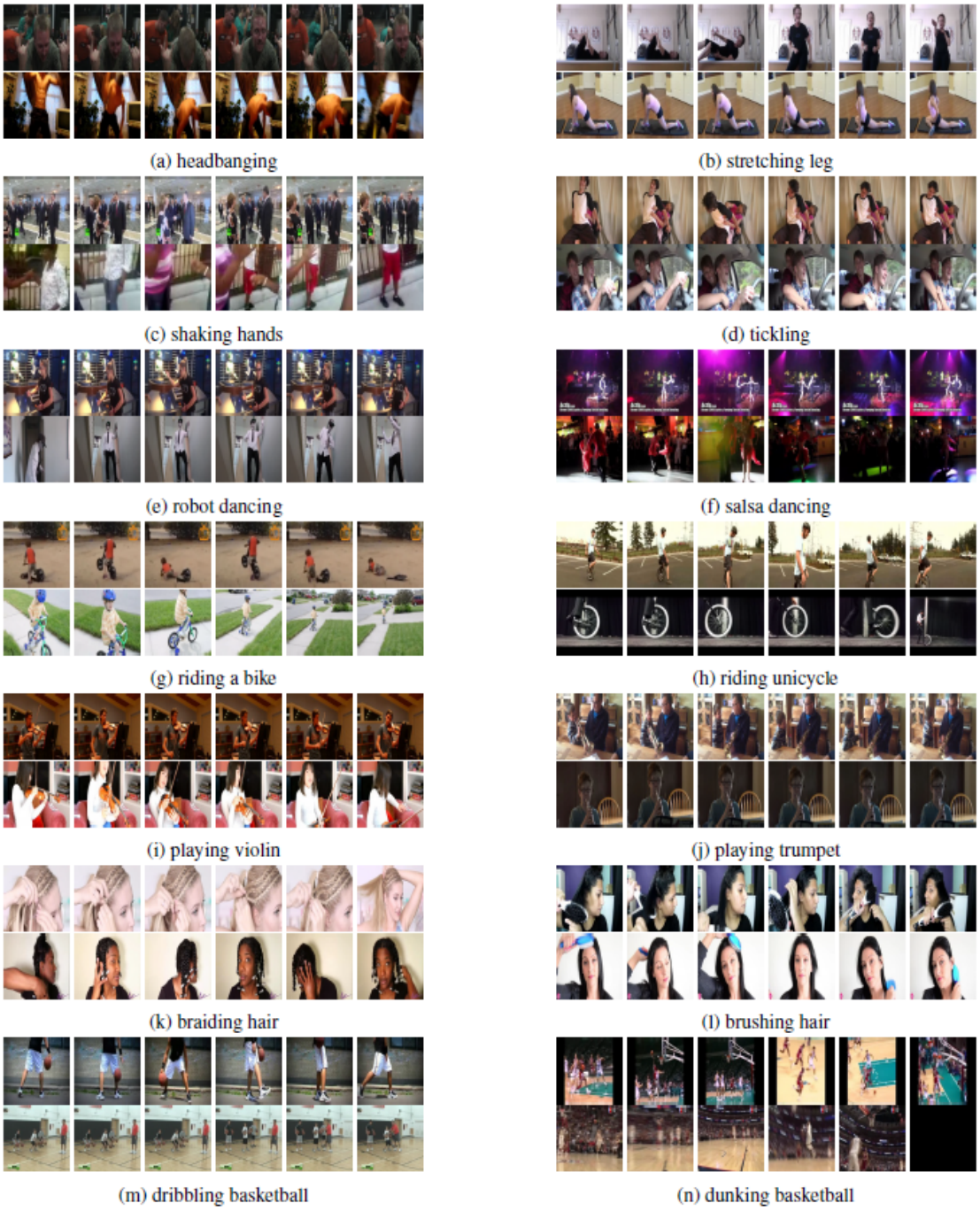

Figure 4.13: Kinetics extracted frames examples
Image taken from [23]

processed by the model at each training iteration. This was possible thanks to the high hardware resources provided by Google Colab, which made possible to train the model in an acceptable time.

Performance metrics

In the literature, there are several metrics that can indicate how effective a neural network is in solving the task for which it is adopted. These metrics, for classification tasks, are calculated as ratios between the different types of prediction made by the classifier, which can be distinguished into four types on the basis of whether they belong to a given class or not: True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN).

The most commonly used metrics to evaluate the classifiers performances are[11]:

- *Accuracy (Acc)*: result of the ratio of all correct predictions made by the classifier to the total number of predictions. In this thesis, *Accuracy* is computed as a global metric so it is calculated as the *average Accuracy* per class, which is the ratio of the number of correct predictions to the total number of predictions. With reference to the prediction types listed above, it is calculated as:

$$Acc = \frac{\sum_i \frac{(TP_i + TN_i)}{TP_i + TN_i + FP_i + FN_i}}{|C|} \tag{4.1}$$

- *Precision (Prec)*: probability of correctly assigning a certain element to a class. It is calculated as:

$$Prec = \frac{TP}{TP + FP}$$

- *Recall (Rec)*: Probability of correct recognition of the elements of a given class by the network. It is calculated as:

$$Rec = \frac{TP}{TP + FN}$$

- *F1-Score (F1)*: harmonic mean between *Precision* and *Recall* calculated as[12]:

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

[11] `http://www.ce.unipr.it/~medici/geometry/node118.html`

[12] `https://it.wikipedia.org/wiki/F1_score`

In this thesis work, all these metrics gave a complete assessment of the performances achieved after the training process by the model and, for every adopted backbone, they were evaluated for each of the 20 classes that composes the previously chosen Kinetics subset. In addition to these metrics, *Support* was also added, i.e. the number of examples of each class that was submitted to the model during testing, to make better considerations on the obtained results.

4.3 Nvidia Jetson Nano

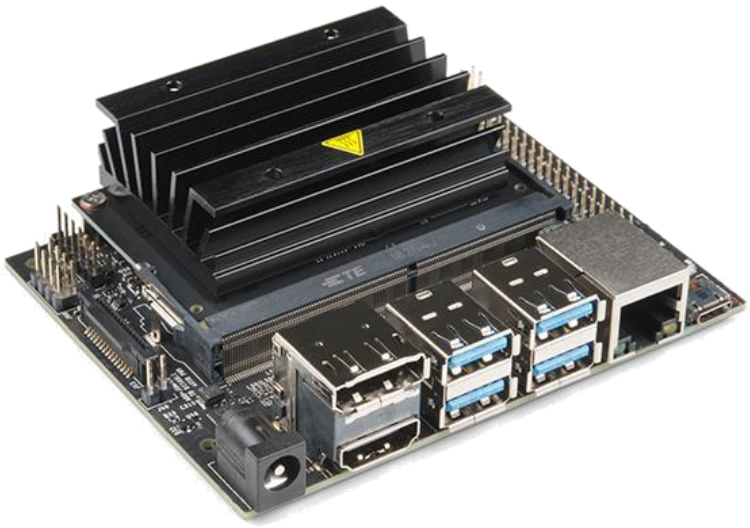

Figure 4.14: Nvidia Jetson Nano board

The Nvidia Jetson Nano is a mini computer designed and built by Nvidia that allows the execution of different neural network models for solving different types of tasks, such as classification and object detection. The discrete hardware resources that this board provides, shown in figure 4.15[13], allow the execution of all the most famous libraries used for Deep Learning, such as Tensorflow, Pytorch and Caffe, allowing the realization of a wide range of applications.

The initial setup of the Jetson Nano is done by flashing an SD Card where to install the JetPack SDK[14], which includes the Linux operating system with the related drivers for the board, the CUDA-X libraries for hardware acceleration and the main libraries and APIs for AI related tasks. Once the initial setup of the

[13] `https://developer.nvidia.com/embedded/jetson-nano-developer-kit`

[14] `https://developer.nvidia.com/embedded/jetpack#install`

GPU	128-core Maxwell
CPU	Quad-core ARM A57 @ 1.43 GHz
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
Storage	microSD (not included)
Video Encode	4K @ 30 | 4x 1080p @ 30 | 9x 720p @ 30 (H.264/H.265)
Video Decode	4K @ 60 | 2x 4K @ 30 | 8x 1080p @ 30 | 18x 720p @ 30 (H.264/H.265)
Camera	2x MIPI CSI-2 DPHY lanes
Connectivity	Gigabit Ethernet, M.2 Key E
Display	HDMI and display port
USB	4x USB 3.0, USB 2.0 Micro-B
Others	GPIO, I^2C, I^2S, SPI, UART
Mechanical	69 mm x 45 mm, 260-pin edge connector

Figure 4.15: Nvidia Jetson Nano technical specifications

board has been completed, for which reference has to be made to the complete guide written by Nvidia[15], it is possible to start working with the device in two different modalities:

- *desktop mode*: mode that allows to work directly with the Jetson Nano, i.e. without the use of an auxiliary PC, simply by connecting a display and input devices such as keyboard and mouse to the board.
- *headless mode*: mode that allows the Jetson Nano to work and to be managed by connecting it to the computer and establishing a connection via SSH protocol.

Given the huge versatility of the Jeston Nano and its accessible price, this device is followed and supported not only by Nvidia, but also by a very active community that contributes continuously by inserting new tutorials, videos and open-source

[15] `https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit#intro`

projects[16] that contribute to making this device a valid solution for the realization of increasingly innovative applications.

4.4 TensorRT

TensorRT is a support development kit for inference acceleration designed and developed by Nvidia that enables the optimization of deep neural networks to achieve reduced latency times and faster execution of Deep Learning tasks, both on high-performance devices and on devices with low computational resources[17]. TensorRT is based on CUDA technology, acronym of Compute Unified Device Architecture, which is the parallel programming model developed by Nvidia for its hardware architectures. Given its effectiveness, many Deep Learning frameworks, including Tensorflow which was used to implement the network model presented earlier in this chapter, have integrated TensorRT within them, allowing it to be used just as a normal library. Given an already trained neural network model, TensorRT allows to perform different types of operations, as shown in figure 4.16, that allow the creation of an *Optimized inference engine*, which allows a faster inference and a more efficient management of the available hardware resources.

The operations performed by TensorRT, with reference to the enumeration shown in figure 4.16, are:

1. *Reduce mixed precision*: quantization operation that reduces the bit number used to represent the model's internal *parameters* in order to speed up the related calculation operations and to reduce the overall size of the model.

2. *Layer and tensor fusion*: merge of the neural network nodes (referred to the graph representation of the model) in order to save the use of GPU resources. Some graph representation nodes of the initial model, in fact, are used only for training and are no longer useful during the inference phase, so this operation allows to remove them and save hardware resources during the inference with the engine.

3. *kernel auto-tuning*: choice of the best algorithms and layers to execute on the available hardware architecture. Depending on the power and characteristics of the board, appropriate layers are chosen to optimize execution.

4. *Dynamic tensor memory*: optimized management of the memory used by the tensors during engine execution.

[16] `https://www.nvidia.com/it-it/autonomous-machines/embedded-systems/jetson-nano/`

[17] `https://developer.nvidia.com/tensorrt`

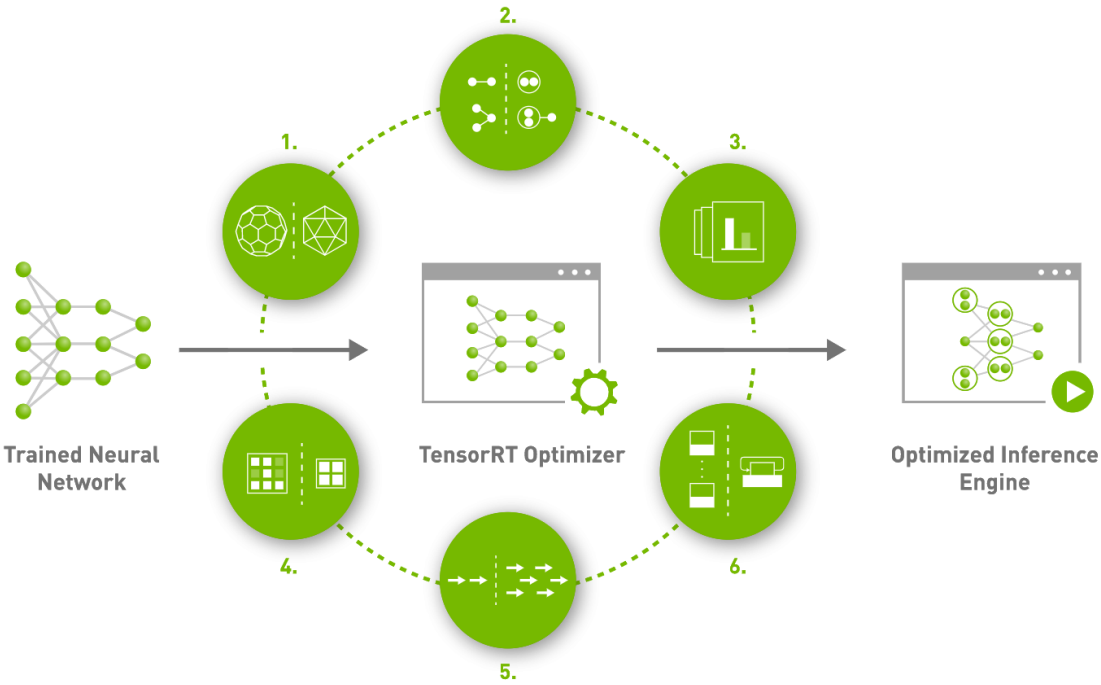

Figure 4.16: TensorRT Optimized inference engine creation

5. *Multi-stream execution*: use of a scalable design for the engine that allows it to be used in parallel with multiple input data streams.

6. *Time fusion*: operation that optimizes time steps of Recurrent Neural Networks used by the engine.

Thanks to the *Optimized inference engine* created by this process, it is possible to get the maximum performance for data inference out of the graphics card based on the chosen model type.

4.5 Model deploy on Jetson Nano

Environment configuration

Before deploying and optimizing the trained model inside the Nvidia Jetson Nano, it was necessary to install the several software components needed for its use and for the development of the Deep Learning framework used to solve the HAR task.

After the initial setup of the Nvidia Jetson Nano, made through the official

Nvidia guide[18], the next step was the installation of Python programming language and Tensorflow platform. Python 3 was installed using the linux terminal command:

```
~$ sudo apt install python3-pip
```

while for Tensorflow, two different versions were installed: Tensorflow 1.15.4 and Tensorflow 2.4.0. With reference to the guide[19], in order to install several versions of Tensorflow within the same device, a special tool for virtualizing environments was used: *virtualenv*.

The used proceeding to install and to create a virtual environment, called *venvtf2.4* on which Tensorflow 2.4.0 was installed, was composed by three steps: the first step was the installation of *virtualenv* on Nvidia Jetson Nano. This has been done with the following command:

```
~$ sudo apt-get install virtualenv
```

The second step was the creation of a virtual environment to install other packages without compromising the ones already installed. The used environment creation command was:

```
~$ python3 -m virtualenv -p python3 venvtf2.4
```

Finally, the third step was used to activate the virtual environment and to start working in it has been accomplished with the following activation command:

```
~$ source venvtf2.4/bin/activate
```

At this point, it was possible to install the several software components within the virtual environment that has just been activated, independently of those installed outside it. To install Tensorflow 2.4.0 within the virtual environment were also needed all the associated packages required for its operations. These packages have been installed using the following commands:

```
~$ sudo apt-get install libhdf5-serial-dev hdf5-tools
libhdf5-dev zlib1g-dev zip libjpeg8-dev liblapack-dev
libblas-dev gfortran
```

```
~$ pip3 install -U pip testresources setuptools==49.6.0
```

[18] https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit

[19] https://docs.nvidia.com/deeplearning/frameworks/install-tf-jetson-platform/index.html?fbclid=IwAR0zkN8bmvXZwr8hF58faofoUp1DmjhRhEQSEjHu_BwreRC65Cv6RQx4NYE

```
~$ pip3 install -U numpy==1.19.4 future==0.18.2
mock==3.0.5 h5py==2.10.0 keras_preprocessing==1.1.1
keras_applications==1.0.8 gast==0.2.2 futures protobuf
pybind11
```

The *h5py* component was quite problematic to install as it required a *wheel* build which failed. To solve this issue it was used the deprecated legacy installation method to install it. This installation method was called automatically with the previous command once the *wheel* build failed.

Once all the previous packages (including *h5py*) have been installed, it was finally possible to install Tensorflow using the following command:

```
~$ pip3 install -extra-index-url
https://developer.download.nvidia.com/
compute/redist/jp/v45 tensorflow==2.4.0+nv21.02
```

In order to work with the *.ipynb* files created and/or downloaded from Google Colab, it was necessary to install the *jupyter-notebook* program within the virtual environment, as well as the *pandas* and *natsort* libraries needed for certain operations used within these files.

All these programs and libraries were installed in the same way as the previous software components using the following terminal commands:

```
~$ pip3 install jupyter

~$ pip3 install pandas

~$ pip3 install natsort
```

The same procedure was followed for the installation of Tensorflow 1.15.4 and all its dependencies, paying close attention to the selection of the correct version of Tensorflow when writing the installation command, as described in the Nvidia guide.

Once the configuration of the working environment and the installation of all the previously mentioned components have been completed, it was possible to deploy the model on the board.

Model optimization

The optimization of the pre-trained model using TensorRT, or the corresponding version already pre-implemented within Tensorflow called TF-TRT which was used

in this thesis work, can be done in two different ways depending on the format in which the model was saved once the training was completed[20]:

- *SavedGraph*: This first modality uses the saved model in SavedGraph format from which it is possible, after a series of proper intermediate steps, to create a *Frozen graph*, i.e. a data structure containing both the graph representing the model and the *parameters* obtained from the training process[21], from which it will be possible to create the *Optimized inference engine*.

Figure 4.17: TensorRT optimization from SavedGraph format

- *SavedModel*: This second modality uses the saved model in SavedModel format, which can be optimized directly using a special procedure. This format involves saving a directory containing a file in *.pb* format, i.e. *Protocol Buffers* format, which stores the actual Tensorflow model, a subdirectory called *assets* which contains files useful for the Tensorflow graph, and a subdirectory called *variables* which contains the training *checkpoints*, i.e. the exact values of the model *parameters*[22].

Figure 4.18: TensorRT optimization from SavedModel format

Since the model was created using Tensorflow version 2.4.1 on Google Colab and, once trained, it was saved in SavedModel format, the modality used for the creation of the *Optimized inference engine* was the second one, whose Python code is available online[23]. Since this procedure, however, was carried out within the

[20] https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html

[21] https://cv-tricks.com/how-to/freeze-tensorflow-models/

[22] https://www.tensorflow.org/guide/saved_model

[23] https://colab.research.google.com/github/vinhngx/tensorrt/blob/vinhn-tf20-notebook/tftrt/examples/image-classification/TFv2-TF-TRT-inference-from-Keras-saved-model.ipynb?hl=en

previously created *venvtf2.4* virtual environment, in order to be able to use all the hardware resources of the Nvidia Jetson Nano without visibility problems, it was necessary to export the values of some determined environment variables using the following three commands to be executed on the linux terminal just after activating the virtual environment using the *source* command:

```
~$ export LD_LIBRARY_PATH=/usr/local/cuda/lib64:
$LD_LIBRARY_PATH

~$ export LD_PRELOAD=/usr/lib/aarch64-linux-gnu/
libgomp.so.1

~$ export CUDA_VISIBLE_DEVICES=0
```

Thanks to these *export* commands, it was possible to solve the problems of visibility of environment variables caused by the use of *virtualenv* and to carry out the conversion procedure successfully.

4.6 Cameras

Figure 4.19: IP Bullet and IP Eyeball network cameras

The cameras used to capture video for inference are the *IP Bullet Network Camera* and the *IP Eyeball Network Camera*, created by Dahua Technology and both shown in figure 4.19. Both camera types have a maximum resolution of 2560×1440 pixels, are equipped with H265 and H264 encoding standards, have a secondary stream and are powered by PoE technology, i.e. *Power over Ethernet* technology. In this thesis work, two IP Bullet cameras and two IP Eyeball cameras were used, for a maximum of four different video streams, and connected to the same network as the Jetson Nano. The configuration of each camera, which is the same for all the camera types, is:

- Codec: H264
- Resolution: 1280×720 pixels
- Fps: 25
- Bit Rate: 2048

Since the neural network model uses input images of size $224 \times 224 \times 3$, the native resolution of the camera was lowered in order to occupy less memory, as well as the number of *Fps* was reduced since there is no advantage in analyzing images too close together for the Human Activity Recognition task. Further settable parameters not mentioned in the previous list were left with their default values. Finally, the frames captured by the cameras were retrieved during the execution by means of the RTSP protocol, an acronym for *Real Time Streaming Protocol*, which uses port 554 as a standard for communication and using different management configurations that are described in the next section.

4.7 Multi-camera handling with thread-based infrastructure

In the previous sections of this chapter, all the hardware devices and software tools used in this thesis work for the realization of a Deep Learning framework that solves the HAR task on edge devices have been presented. In this section it is described in detail the functioning of the developed framework and the different ways of managing the components previously introduced. Specifically, two main multi-threading configurations have been identified, which both allow different interactions with the *Optimized inference engine*, of which there will be only one loaded in RAM memory due to the limited hardware resources available, and different managements of the inputs created from the video streams transmitted by the cameras. Each of these configurations, moreover, has been implemented in two different versions that differ according to the retrieval method of the video streams frames. Thereafter, for each configuration will be described the operating principle on which it is based and the types of threads that compose it, comprehending all the functions they perform. The evaluation of the performance obtained by each configuration will be discussed, instead, in chapter 6.

4.7.1 Configuration 1

The first implemented configuration is based on the use of a single thread associated to a single camera, both for the inference and for the acquisition of the input from

the video stream, which includes all the procedures for the creation of the input *batch* to be analyzed by the model, starting from the single captured frames. The first version of this configuration, named CF1v1, is composed of three main types of threads, namely the *Main thread*, the *Acquisition and Inference thread* and the *Capturing thread*, which allow the execution of all the necessary operations to fulfill the Human Activity Recognition task. A first representation of how this configuration works is shown in figure 4.20, where the main interactions between all the main types of threads and the software objects are present with the exception of the *Main thread*, which has been omitted from the workflow representation as it only has functions that concern the initialization of the other threads and the on-screen results display, thus being of minor relevance in the explanation of the workflow that leads to the resolution of the task. A second representation is also shown in figure 4.21, which reports the operations sequence and all the interactions between the several threads and the software objects of this configuration.

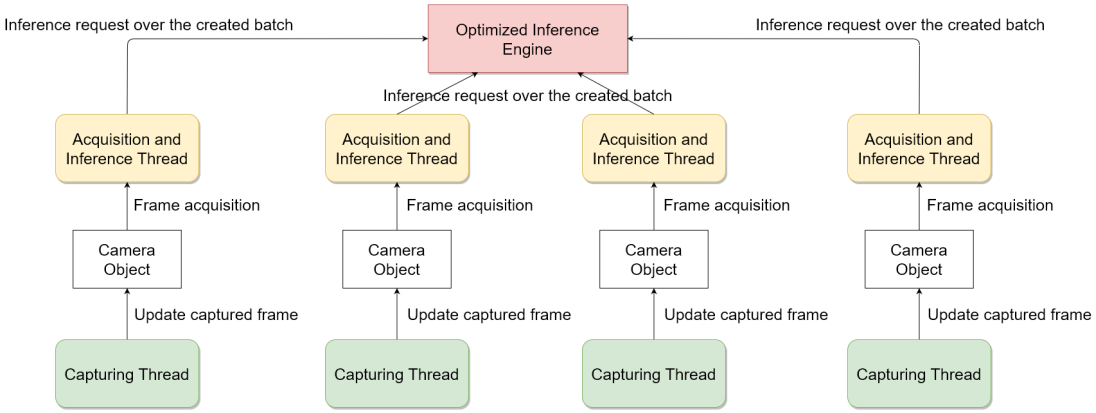

Figure 4.20: Multi-camera handling configuration 1 version 1

The main functionalities of all these components, or *actors*, are:

- *Main thread*: this thread is the starting point of the whole workflow and takes care of the initialization of all the other *actors*. First of all, it loads the *Optimized inference engine* in RAM memory and runs a test on a default input *batch* to allow the initial loading of all the libraries needed for inference. Once the engine is loaded, this thread allows the user to input all the access information about the cameras connected to the same network of the Jetson Nano, in order to establish a communication for accessing the video streams. For each camera, if the access data are correct, a *Camera object* will be instantiated and a dedicated *Acquisition and Inference thread* will be created to interact with it. Finally, once all these components have been created, the *Main thread* will be in charge of displaying the results produced on screen.

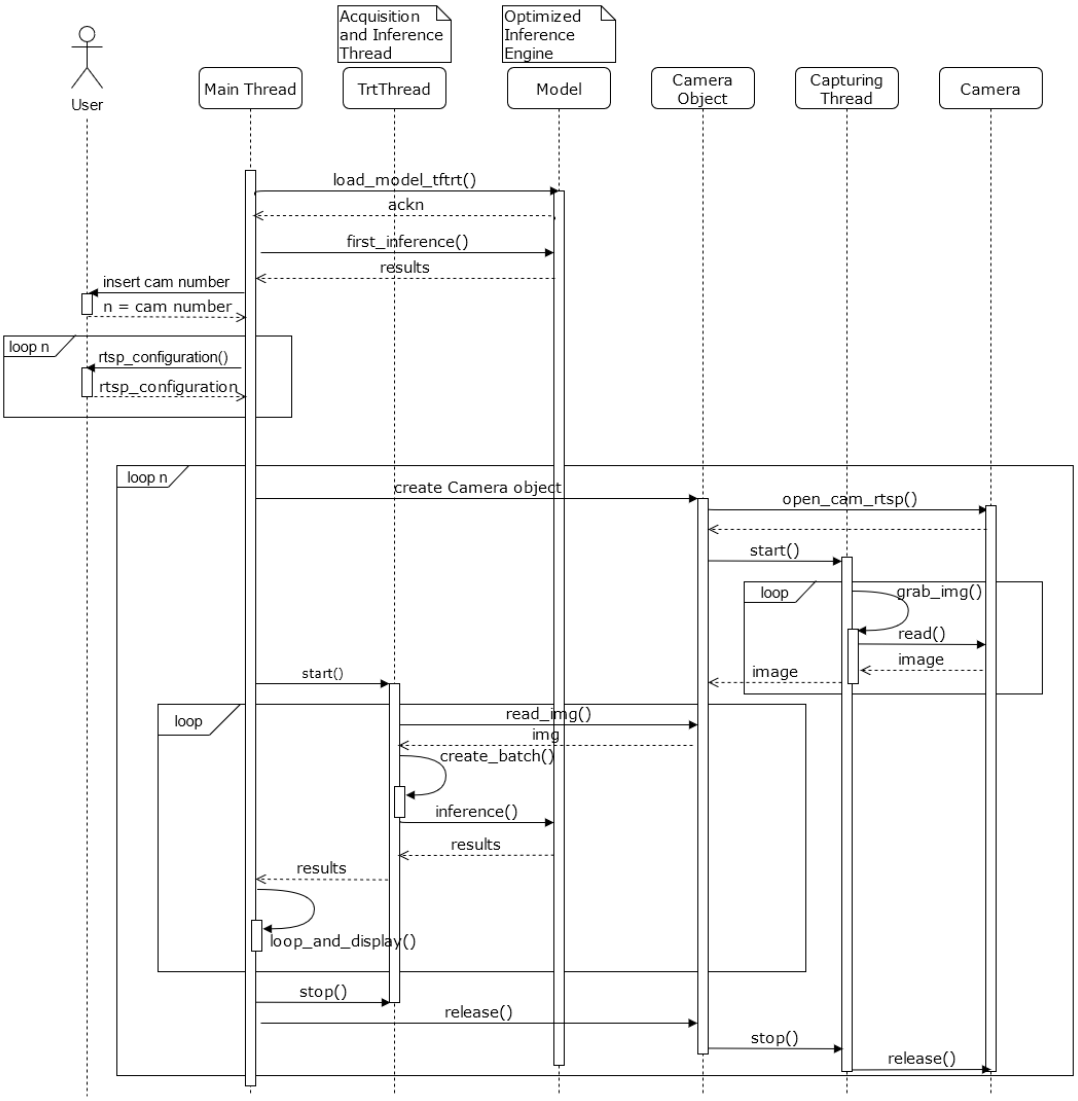

Figure 4.21: Multi-camera handling CF1v1 sequence diagram

- *Optimized inference engine*: copy of the engine in RAM memory that is in charge to make predictions on the input data. In this particular configuration, it is used by every single *Acquisition and Inference thread* in an asynchronous way to analyze each time the related single *batch* of input acquired by that thread, and to return the label associated to the prediction.

- *Camera object*: this object represents the camera and contains all the attributes and methods for interacting with it. When it is created, this object also takes care of creating a corresponding *Capturing thread*, which retrieves the frames

captured by the associated camera.

- *Capturing thread*: thread created by the *Camera object* that plays the role of updating the attribute of the last captured frame where the image is stored. This thread runs continuously in the background and works in a completely transparent manner to all other threads.

- *Acquisition and inference thread*: this type of thread is the one that contains the founding principle of this configuration and performs two particular types of functions: the first is the acquisition of the various frames from the *Camera object* and the creation of the input *batch* to be analyzed; the second, instead, concerns the use of the engine to make the prediction on the created *batch*. Each frame taken from the Camera object is normalized in order to standardize the values of the pixels and is inserted into an array. Once the ten frames that make up an entire *batch* have been collected, the array containing all of these images is sent to the engine to be inferred.

In this first version of configuration 1, therefore, for each camera connected to the network are created two threads in addition to the *Main thread*. The second version of this first configuration, named CF1v2, does not use *Capturing Threads* instead. The frame capture function that the *Capturing threads* are in charge to perform, with the relative updating of the related Camera object attribute, is carried out within the *Acquisition and Inference thread*. A representative diagram of this second version of configuration 1 is shown in figure 4.22 and, as can also be seen from the figure, for each camera connected to the network there is a single thread, in addition to the *Main thread*, associated with it. In figure 4.23 is shown, instead, the operations sequence of this second version to better comprehend the entire workflow of this multi-threading configuration.

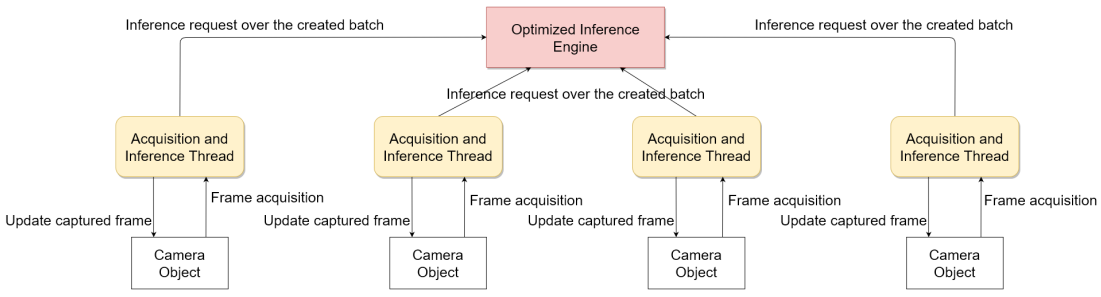

Figure 4.22: Multi-camera handling configuration 1 version 2

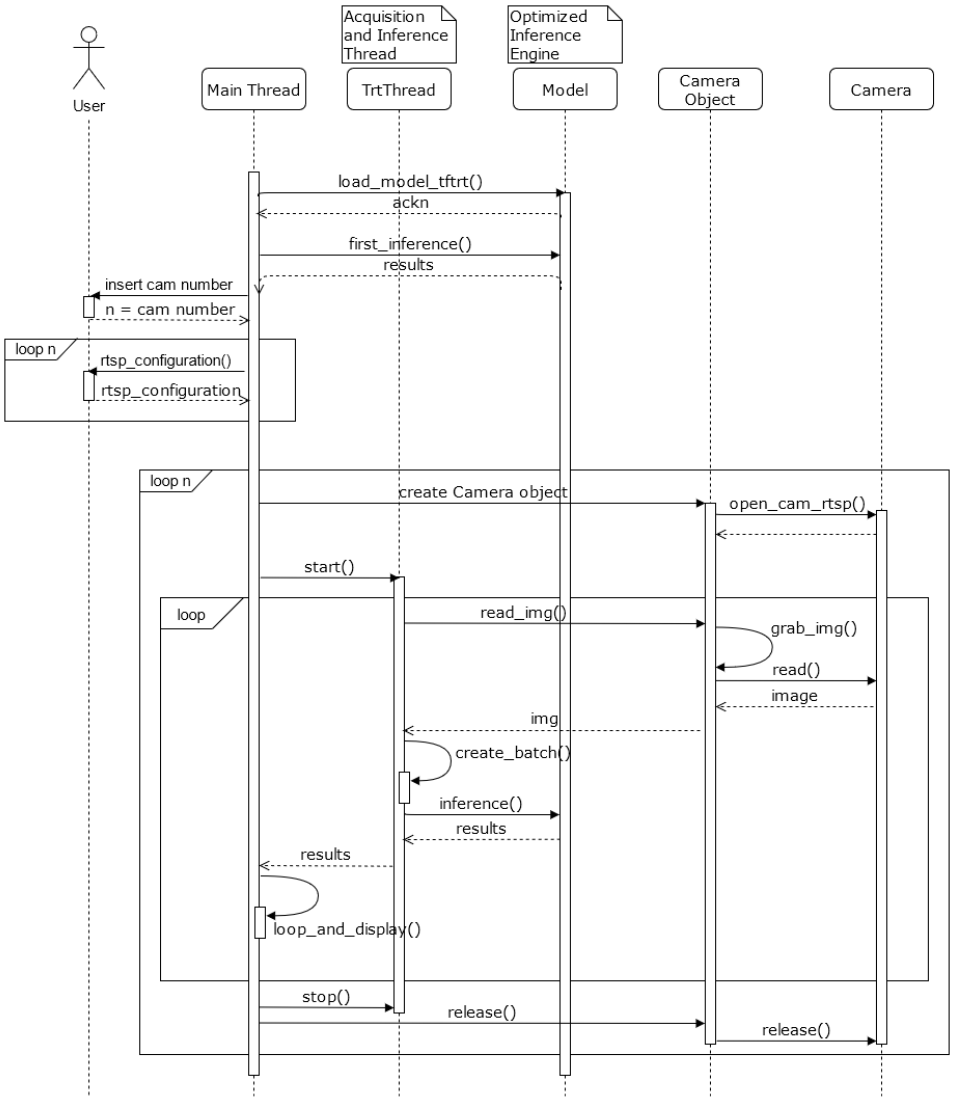

Figure 4.23: Multi-camera handling CF1v2 sequence diagram

4.7.2 Configuration 2

The second configuration implemented is based on the use of two types of threads that make it possible to separate the acquisition and inference functions seen in the previous configuration, with the aim of performing the prediction of the *batches* collected by the various cameras in a synchronous manner. Thanks to this second multi-threading structure it is possible, in fact, to analyze the various *batches* collected in block, thus solving some of the management problems of the engine, which is no longer contended by the several threads.

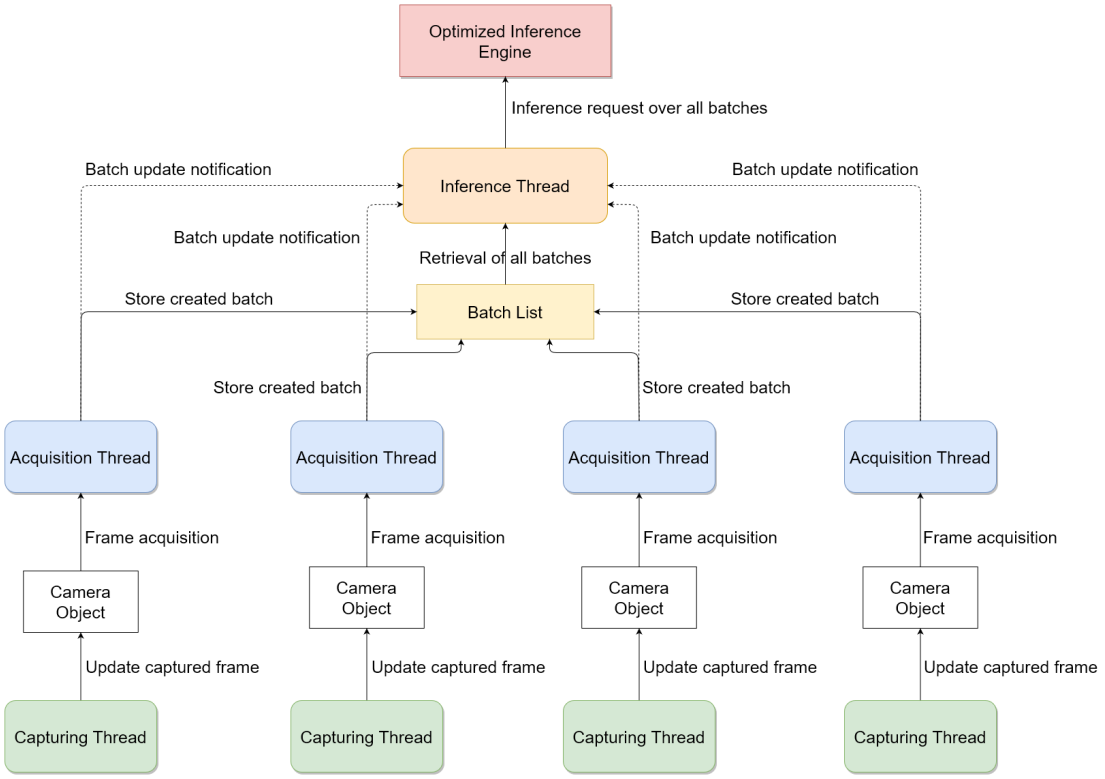

Figure 4.24: Multi-camera handling configuration 2 version 1

The main *actors* of this second configuration are the *Main thread*, the *Acquisition threads*, the *Inference thread* and the *Capturing threads*. All these components are represented in figure 4.24, with the exception of the *Main thread* for the same reason mentioned in the previous configuration. A complete sequence diagram of this second configuration is also reported, as for the previous one, in figure 4.25. The functionalities performed by each *actor* are described below with the exception of the *Capturing thread* and the related *Camera object* as they have already been listed in the previous configuration:

- *Main thread*: as in configuration 1, this thread takes care of the creation and initialization of all the other *actors*. The main difference compared to the one presented previously is that it creates an *Acquisition thread* for each camera connected to the network and a single *Inference thread* that will take care of the interaction with the engine. It is also responsible for the creation of a list, called *Batch List*, in which all the *Acquisition threads* will insert their *batches*.
- *Batch List*: list used by each *Acquisition thread* to store its own created input *batch* while waiting for it to be analyzed by the engine. Only one *batch* for

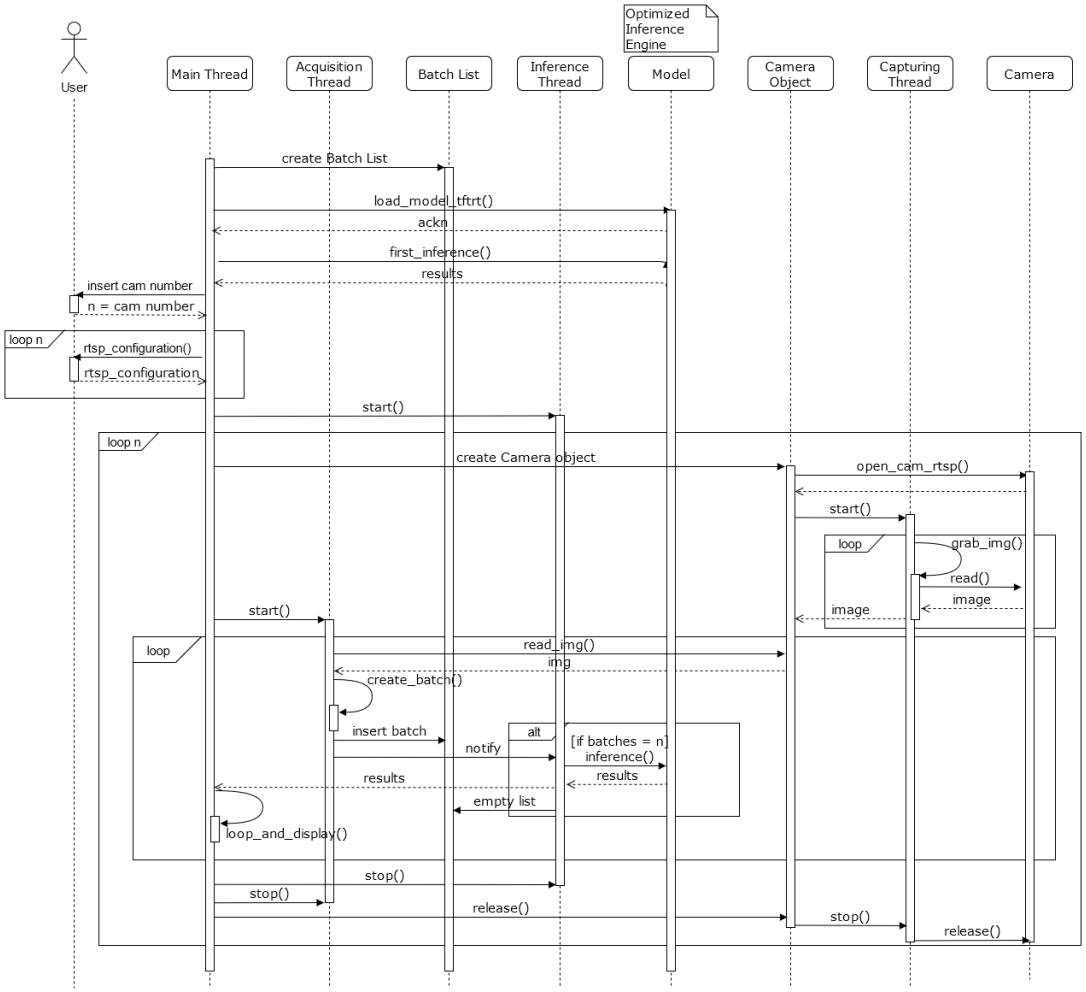

Figure 4.25: Multi-camera handling CF2v1 sequence diagram

Acquisition thread is stored within this list, so that if an *Acquisition thread* had already stored an input *batch* that has not yet been analyzed by the engine, it can not insert another.

- *Acquisition thread*: thread that carries out the function of frames acquisition from the camera object, which also includes the normalization process, in order to create the input *batch* to be submitted to the engine. Since the interaction with the engine is managed by the *Inference thread*, the acquisition function is suspended until the created *batch*, saved in the *Batch List*, is examined by the engine. To notify the *Inference thread* that the captured input has been saved into the dedicated list, each *Acquisition thread* will notify the *Inference thread* each time it stores an input *batch* in the *Batch List*.

- *Inference thread*: thread in charge of the communication with the engine for the inference on the input *batches* saved inside the *Batch List*, which will be analyzed in block. The *Inference thread* analyzes all the input acquired by means of the engine only when all the *Acquisition threads* have added an input *batch* to the *Batch List*. Each time an *Acquisition thread* inserts an input *batch*, the *Inference thread* receives a notification followed by a check of the number of elements in the *Batch List*. If the number of *batches* saved in the list matches with the number of the *Acquisition threads*, i.e. if all of them have inserted their own *batch*, the *Inference thread* starts the prediction procedure and, once accomplished, empties the list in order to start again the process of new frames acquisition of the *Acquisition threads*.

The first version of configuration 2 just analyzed, named CF2v1, has a total number of threads running in parallel of two, i.e. *Main thread* and *Inference thread*, plus another two for each camera in the network. A second version was also implemented for this configuration, named CF2v2 and shown in figure 4.26, where the *Capturing threads* were removed and their functionalities were added to the *Acquisition threads*, thus reducing the total number of running threads by one thread per camera. Even for this version, finally, the entire operations sequence diagram is reported in figure 4.27.

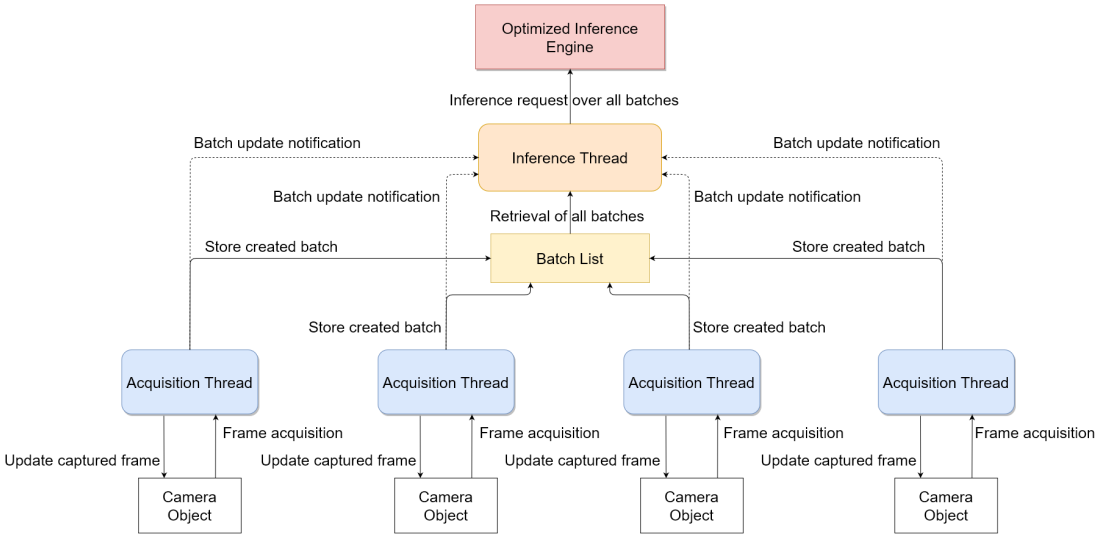

Figure 4.26: Multi-camera handling configuration 2 version 2

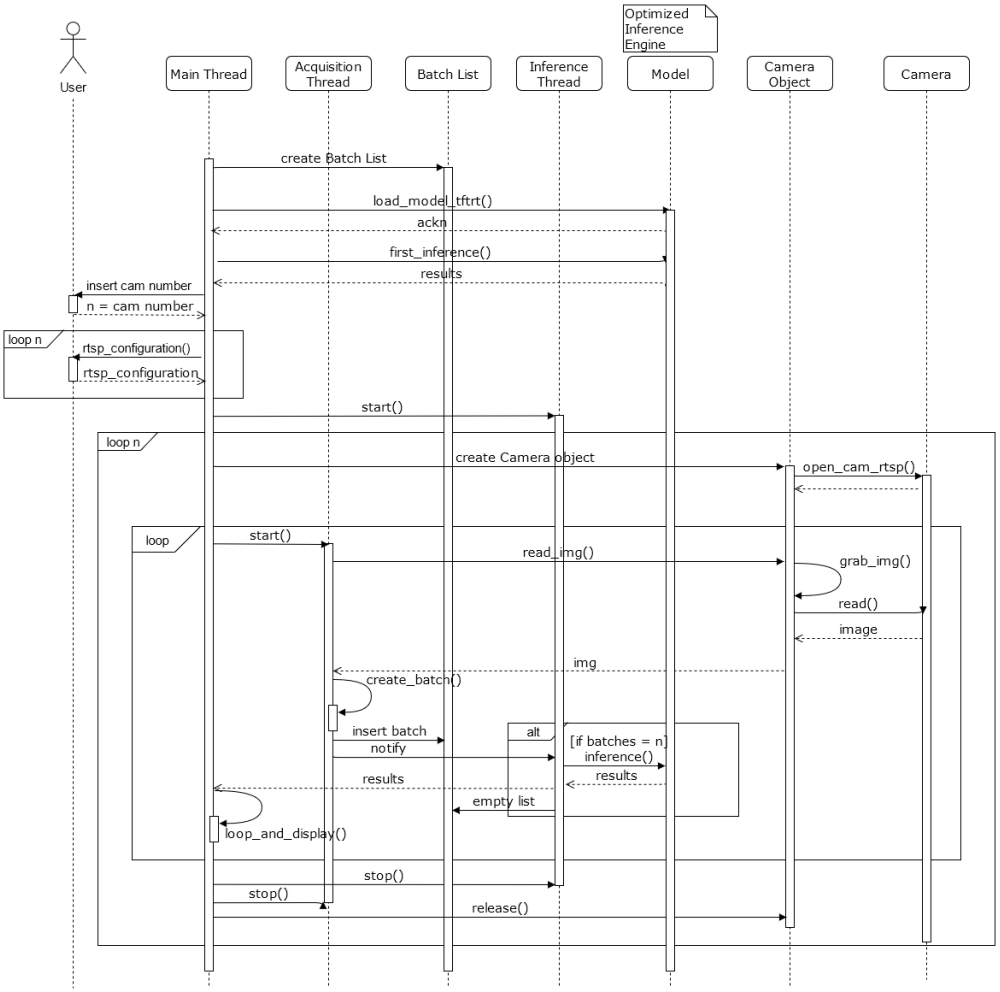

Figure 4.27: Multi-camera handling CF2v2 sequence diagram

Chapter 5

Results

In this chapter are listed the results obtained from the training of the model for all the adopted backbones, as well as the performances of the different configurations presented in section 4.7, varying the used backbone and the number of cameras in the edge devices network. To evaluate the performances achieved by the model after the training process, *Precision*, *Recall*, *F1-score* and *Accuracy* metrics were used, as reported in section 4.2. To analyze the effectiveness and the efficiency of the multi-camera handling configurations, instead, *Fps*, *Inference time* and the *Total time* required to analyze a single input *batch* were used. All the obtained results will be discussed in Chapter 6, where all the evaluations about the implemented solutions will be reported.

5.1 Model performances

A complete evaluation of all the model performances achieved after the training process using MobileNetV2, MobileNetV3Small and MobileNetV3Large backbones respectively, are shown in tables 5.1, 5.2 and 5.3. All these results were obtained on a dedicated test set, extracted from Kinetics dataset, composed by a total of 2140 videos. The number of video submitted to the model for each class is reported in the *Support* column of each table.

In table 5.1 are shown the performances achieved by the model adopting MobileNetV2 backbone, which is the one with the best performances among the three. The resulting average *Prec* is 0.76, the average *Rec* is 0.74 and the average *F1* is 0.75, while the weighted averages of this metrics, as well as the global *Acc*, reach the value 0.77.

In table 5.2 are shown the performances achieved by the model adopting MobileNetV3Small backbone. Among all the three models, it has the lowest performances since it has 0.68 average *Prec*, 0.66 average *Rec* and 0.66 average F1, while

the global accuracy and the weighted values of the above mentioned metrics are 0.69.

Finally, in table 5.3 are shown the performances achieved by the model with MobileNetV3Large backbone, which are slightly worse than the performances achieved by MobileNetV2 backbone. All the average values of *Prec*, *Rec* and *F1* are 0.74, while the weighted *Prec* is 0.77 and the weighted *Rec*, as well as the weighted *F1* and the global *Acc* are 0.76.

Table 5.1: Model performances achieved with the MobileNetV2 backbone. For each class, the corresponding *Precision*, *Recall* and *F1-score* values are reported. On the bottom of the table, instead, the average value of *Precision* and *Recall* and the global *Accuracy* value are reported.

MobileNetV2 model performances				
Class	Precision	Recall	F1-Score	Support
brushing_teeth	0.78	0.80	0.79	183
cleaning_floor	0.75	0.88	0.81	128
cleaning_toilet	0.90	0.86	0.88	70
cleaning_windows	0.80	0.81	0.81	95
crawling_baby	0.83	0.90	0.87	183
dining	0.75	0.86	0.80	95
doing_nails	0.81	0.80	0.80	129
drinking	0.64	0.47	0.54	77
hugging	0.47	0.36	0.41	64
ironing	0.79	0.82	0.81	66
kissing	0.56	0.80	0.66	59
making_bed	0.85	0.74	0.79	91
opening_bottle	0.65	0.55	0.60	98
playing_cards	0.95	0.93	0.94	102
reading_book	0.84	0.77	0.80	177
setting_table	0.84	0.69	0.76	55
using_computer	0.94	0.93	0.93	135
washing_dishes	0.71	0.83	0.77	157
washing_hair	0.67	0.50	0.57	44
washing_hands	0.61	0.60	0.61	132
accuracy			0.77	2140
macro avg	0.76	0.74	0.75	2140
weighted avg	0.77	0.77	0.77	2140

Table 5.2: Model performances achieved with the MobileNetV3Small backbone. For each class, the corresponding *Precision*, *Recall* and *F1-score* values are reported. On the bottom of the table, instead, the average value of *Precision* and *Recall* and the global *Accuracy* value are reported.

MobileNetV3Small model performances				
Class	Precision	Recall	F1-Score	Support
brushing_teeth	0.59	0.73	0.66	183
cleaning_floor	0.71	0.81	0.76	128
cleaning_toilet	0.74	0.79	0.76	70
cleaning_windows	0.81	0.75	0.78	95
crawling_baby	0.78	0.85	0.81	183
dining	0.70	0.85	0.77	95
doing_nails	0.81	0.81	0.81	129
drinking	0.33	0.34	0.33	77
hugging	0.38	0.36	0.37	64
ironing	0.70	0.59	0.64	66
kissing	0.63	0.53	0.57	59
making_bed	0.83	0.71	0.77	91
opening_bottle	0.49	0.44	0.46	98
playing_cards	0.88	0.91	0.89	102
reading_book	0.74	0.72	0.73	177
setting_table	0.75	0.65	0.70	55
using_computer	0.94	0.83	0.88	135
washing_dishes	0.66	0.69	0.67	157
washing_hair	0.45	0.34	0.39	44
washing_hands	0.58	0.47	0.52	132
accuracy			0.69	2140
macro avg	0.68	0.66	0.66	2140
weighted avg	0.69	0.69	0.69	2140

Table 5.3: Model performances achieved with the MobileNetV3Large backbone. For each class, the corresponding *Precision*, *Recall* and *F1-score* values are reported. On the bottom of the table, instead, the average value of *Precision* and *Recall* and the global *Accuracy* value are reported.

MobileNetV3Large model performances				
Class	Precision	Recall	F1-Score	Support
brushing_teeth	0.76	0.79	0.77	183
cleaning_floor	0.81	0.81	0.81	128
cleaning_toilet	0.87	0.79	0.83	70
cleaning_windows	0.81	0.79	0.80	95
crawling_baby	0.89	0.87	0.88	183
dining	0.81	0.92	0.86	95
doing_nails	0.89	0.78	0.83	129
drinking	0.64	0.45	0.53	77
hugging	0.42	0.55	0.47	64
ironing	0.78	0.80	0.79	66
kissing	0.53	0.68	0.59	59
making_bed	0.77	0.82	0.80	91
opening_bottle	0.56	0.53	0.54	98
playing_cards	0.90	0.90	0.90	102
reading_book	0.77	0.77	0.77	177
setting_table	0.75	0.69	0.72	55
using_computer	0.96	0.92	0.94	135
washing_dishes	0.70	0.80	0.75	157
washing_hair	0.55	0.52	0.53	44
washing_hands	0.68	0.58	0.62	132
accuracy			0.76	2140
macro avg	0.74	0.74	0.74	2140
weighted avg	0.77	0.76	0.76	2140

5.2 Multi-camera configurations performances

The performances evaluation of the two different configurations described in section 4.7 for the management of multiple video streams was carried out on two different execution modes: the execution with the on-screen display of the images collected by the cameras comprehending the prediction result, called *camera visualization* mode, and the execution without the images screen display, i.e. *without camera visualization* mode where only the label of the prediction produced by the engine was shown.

This distinction was made because in a resource-limited environment such as Jetson Nano, the sub-processes in charge of displaying the video streams on the screen can not be assumed to be computationally negligible. These sub-processes, in fact, lead to a much greater performances decay, for the same number of cameras, compared to those obtained from the execution of the same configuration without video stream visualization.

To evaluate all these performances three different metrics were used:

- *Average inference time (AIT)*: average time required to perform inference on a single *batch* using the *Optimized inference engine*.

- *Total time (TotT)*: sum of the time needed to acquire the ten frames that compose a single *batch* and the inference on them. This type of time is only calculated in *without camera visualization* execution mode because the number of *Fps* can not be calculated in this execution mode as the number of frames showed on the screen. Indicating with $AcqT$ the single *frame Acquisition time*, the *Total time* is calculated as:

$$TotT = AIT + 10 \cdot AcqT$$

- *Fps*: acronym for *Frame per second*, this metric defines the number of frames that can be processed per second. In the case of using the *camera visualization* execution mode the calculation of *Fps* was carried out on the basis of the number of frames shown in the window dedicated to the single camera, while in the case of *without camera visualization* mode, a mathematical calculation was used based on the *frame Acquisition time*, i.e. the time needed to save the image on the Camera Object, and the AIT of a single *batch*. The number of *Fps*, in this second case, can in fact be calculated as the reciprocal of one tenth of the $TotT$ or, similarly, by making the reciprocal of the sum of the $AcqT$ and one tenth of the AIT, i.e. the time that it would theoretically take the model to infer a single frame:

$$Fps = \frac{1}{\frac{TotT}{10}} = \frac{1}{\frac{AIT}{10} + AcqT}$$

All those metrics values are reported in the following tables, denoting with the × symbol the impossibility of the framework execution using the related model and number of cameras.

The results obtained from the two versions of configuration 1 are shown in tables 5.4 and 5.5.

Table 5.4: Multi-camera configuration 1 version 1 performances achieved using the *Optimized inference engine* obtained from all the three model variants and for a number of cameras varying from 2 to 4.

Configuration 1 version 1						
Model	N. of cameras	Camera visualization		Without camera visualization		
		Average inference time	Fps	Average inference time	Total time	Fps
MobileNetV2	2	×	×	~0.48s	~0.56s	~18
MobileNetV2	3	×	×	~0.73s	~0.86s	~12
MobileNetV2	4	×	×	×	×	×
MobileNetV3 Small	2	~0.10s	~18	~0.17s	~0.26s	~39
MobileNetV3 Small	3	~0.25s	~11	~0.30s	~0.46s	~23
MobileNetV3 Small	4	~0.36s	~8	~0.37s	~0.59s	~17
MobileNetV3 Large	2	~0.38s	~16	~0.40s	~0.49s	~20
MobileNetV3 Large	3	~0.60s	~12	~0.60s	~0.77s	~13
MobileNetV3 Large	4	×	×	~0.76s	~0.97s	~10

For the two versions of the configuration 2, since the inference was carried out in block on all the *batches* memorized by the *Acquisition threads* in the *Batch List*, the previous metrics vary slightly in meaning since the *AIT*, although it corresponds to a single call of the *Optimized inference engine* by the *Inference thread*, no longer corresponds to the analysis of a single *batch* but corresponds to the analysis of a number of *batches* equal to the number of cameras in the edge devices network. The results obtained from the two versions of configuration 2 are shown in tables

Table 5.5: Multi-camera configuration 1 version 2 performances achieved using the *Optimized inference engine* obtained from all the three model variants and for a number of cameras varying from 2 to 4.

Configuration 1 version 2						
Model	N. of cameras	Camera visualization		Without camera visualization		
		Average inference time	Fps	Average inference time	Total time	Fps
MobileNetV2	2	~0.36s	~4	~0.35s	~0.43s	~23
MobileNetV2	3	×	×	~0.68s	~0.78s	~13
MobileNetV2	4	×	×	×	×	×
MobileNetV3 Small	2	~0.09s	~17	~0.09s	~0.17s	~58
MobileNetV3 Small	3	~0.22s	~13	~0.21s	~0.31s	~32
MobileNetV3 Small	4	~0.33s	~7	~0.30s	~0.43s	~23
MobileNetV3 Large	2	~0.38s	~15	~0.25s	~0.33s	~30
MobileNetV3 Large	3	×	×	~0.54s	~0.65s	~15
MobileNetV3 Large	4	×	×	×	×	×

5.6 and 5.7.

Since the neural network model with MobileNetV3Small backbone adopted, which was the lightest, was well supported by all the different configurations even with four cameras, the performances with the limit number of cameras that the board could handle with this model were tested. Considering also that, in a real video surveillance use case, all those configurations have to be executed in *without camera visualization* mode in order to save computational resources, for this test were considered only the performances achieved with this execution modality by the best version of each configuration. The obtained results are shown in table 5.8.

Table 5.6: Multi-camera configuration 2 version 1 performances achieved using the *Optimized inference engine* obtained from all the three model variants and for a number of cameras varying from 2 to 4.

Configuration 2 version 1						
Model	N. of cameras	Camera visualization		Without camera visualization		
		Average inference time	Fps	Average inference time	Total time	Fps
MobileNetV2	2	×	×	×	×	×
MobileNetV2	3	×	×	×	×	×
MobileNetV2	4	×	×	×	×	×
MobileNetV3 Small	2	~0.18s	~24	~0.18s	~0.27s	~37
MobileNetV3 Small	3	~0.26s	~15	~0.26s	~0.39s	~26
MobileNetV3 Small	4	~0.38s	~10	~0.42s	~0.61s	~16
MobileNetV3 Large	2	~0.42s	~13	~0.39s	~0.49s	~20
MobileNetV3 Large	3	×	×	~0.60s	~0.74s	~13
MobileNetV3 Large	4	×	×	×	×	×

Table 5.7: Multi-camera configuration 2 version 2 performances achieved using the *Optimized inference engine* obtained from all the three model variants and for a number of cameras varying from 2 to 4.

Configuration 2 version 2						
Model	N. of cameras	Camera visualization		Without camera visualization		
		Average inference time	Fps	Average inference time	Total time	Fps
MobileNetV2	2	×	×	~0.52s	~0.59s	~17
MobileNetV2	3	×	×	×	×	×
MobileNetV2	4	×	×	×	×	×
MobileNetV3 Small	2	~0.17s	~22	~0.15s	~0.23s	~43
MobileNetV3 Small	3	~0.27s	~20	~0.24s	~0.34s	~29
MobileNetV3 Small	4	~0.36s	~19	~0.40s	~0.52s	~19
MobileNetV3 Large	2	~0.40s	~19	~0.42s	~0.49s	~20
MobileNetV3 Large	3	×	×	×	×	×
MobileNetV3 Large	4	×	×	×	×	×

Table 5.8: Best configuration versions limit performances without camera visualization

Best configuration versions limit performances					
Configuration version	Model	Number of cameras	Average inference time	Total time	Fps
Configuration 1 version 1	MobileNetV3 Small	9	~1.18s	~1.52s	~6
Configuration 2 version 2	MobileNetV3 Small	7	~0.93s	~1.21s	~8

Chapter 6

Discussion

In this chapter are reported all the evaluations made on the proposed solutions, focusing on the obtained results and possible improvements regarding the trained neural network model and the several multi-camera handling realized configurations.

6.1 Proposed architecture evaluation

As it can be noticed from the tables 5.1, 5.2 and 5.3, the best performances were achieved by the model that uses MobileNetV2 backbone, which reach a global *Accuracy* of 77% as well as weighted *Precision*, *Recall* and *F1-score*. The model trained with MobileNetV3Large backbone had comparable performances with the one trained with MobileNetV2 while the last type of model, the one with the MobileNetV3Small backbone adopted, had the worst performances, reaching the 69% of global *Accuracy* as well as weighted *Precision*, *Recall* and *F1-score*. This was due to the number of trainable *parameters* of the different architectures, which are a measure of how much a neural network can learn. The model trained with MobileNetV2 had, in fact, 2,257,984 total *parameters*, the one that uses MobileNetV3Large had 2,667,688 *parameters* and the one that uses MobileNetV3Small had 1,031,848 *parameters*. Since the architectures with MobileNetV2 and MobileNetV3Large had a comparable number of these indicators, their performances looked similar, while the model that have used the MobileNetV3Small backbone, which had less than the half of the other two architectures trainable *parameters*, had not reached the same metrics score with the same training modalities.

There was also a considerable difference between the several classes, for example the *hugging* class and the *using_computer* class, in the model recognition capabilities regardless of the backbone used. This may be due to the video composition of the Kinetics subset and to the chosen optimizer. Several classes, in fact, were easily identified when certain features were detected from the network, such as

the presence of a keyboard for the *using_computer* class, while some other classes did not have specific features and were more difficult for the neural network to identify. Since the *Adagrad* optimizer enhanced specific features parameters updates compared to those of more common features during the training process, in order to increase the recognition performances of certain types of classes some additional examples could be included in the training set.

In addition to this, there was also another problem relating to the recognition of the human activities in large environments. As it is shown in figure 6.1, which represents an example of a correct prediction made by the model during the execution of the Deep Learning framework using the *camera visualization* mode, the trained model worked well in recognizing activities on which the camera was focused, while if the images showed a wider overview of the surrounding environment, as it is shown in figures 6.2 and 6.3, the *Accuracy* of the model significantly dropped.

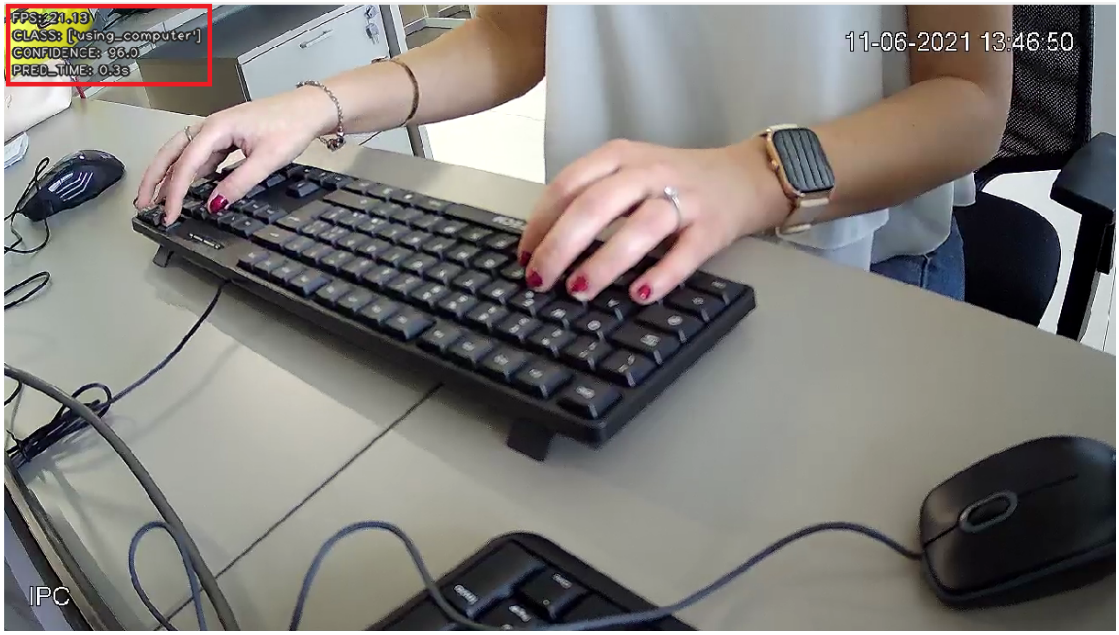

Figure 6.1: Correct prediction example in the framework's camera visualization execution mode

This happened because most of the videos of the Kinetics dataset were focused directly on the performed action without considering the surrounding environment. This leaded to significant inference errors in all the cases where the cameras captured wide spaces beyond the performed action, such as in figure 6.2 where the presence of the windows, correctly identified by the neural network model, prevailed on the performed action that was, instead, focused on the keyboard.

In order to avoid the previously mentioned problem, more videos with a wider

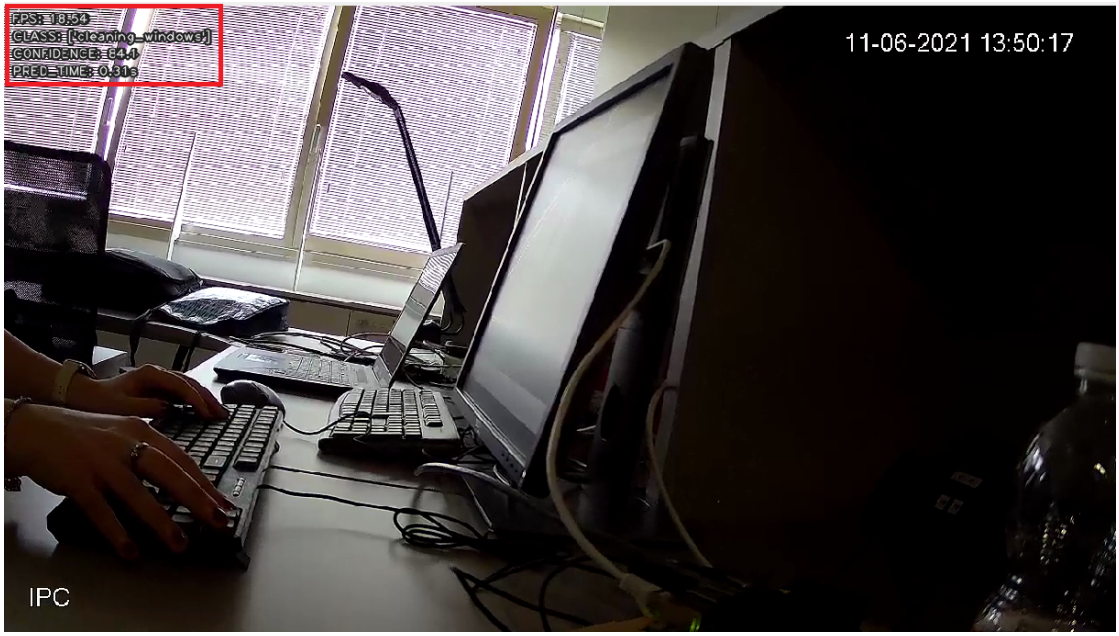

Figure 6.2: First wrong prediction example in the framework's camera visualization execution mode

Figure 6.3: Second wrong prediction example in the framework's camera visualization execution mode

camera range could be utilized for training the model, giving to it a better comprehension of which portion of the captured frames contains the performed action.

6.2 Multi-camera handling configurations evaluation

As it can be noticed from tables 5.4, 5.5, 5.6 and 5.7 the performances of the various configurations varied drastically depending on which type of visualization mode was adopted, i.e. whether the results of the predictions were to be presented with the camera images also displayed on screen or not.

Of all the backbones that have been used, the only one that worked correctly for all configurations and for a number of managed cameras ranging from two to four, was MobileNetV3Small. This backbone, in fact, was lighter than the others and requires less RAM memory as well as being faster in the inference of the input *batches*. The use of the other backbones, with reference to the *camera visualization* execution mode, did not allow the execution of the entire framework for a number of managed cameras equal to three for the MobileNetV3Large backbone or even with two cameras in the case of the MobileNetV2 backbone, which resulted as being the most onerous backbone from the computational point of view.

Of all the tested configurations, the one that provided the greatest stability with respect to varying the number of cameras was CF1v1, whose performance is shown in table 5.4. This type of configuration allowed, in fact, the execution of almost all the different architectures of the network model in the *without camera visualization* execution mode, the only exception of which was the case in which there were four cameras and the model used the MobileNetV2 backbone, while in the *camera visualization* execution mode it was the only one that manages to execute the entire framework using three cameras and the MobileNetV3Large backbone.

On the other hand, the second version of configuration 1, i.e. CF1v2, with reference to table 5.5, did not manage to achieve the stability of the first version even though it achieved a considerable performance boost in terms of *Fps*, especially when using the model with MobileNetV3Small backbone. This was due to the lower number of threads that constitute the infrastructure, which entailed a reduction of the context switch times between the various sub-processes, given that the *Capturing threads* that competed for hardware resources were no longer present. At the same time this second version leaded to a weighting of the functions carried out by the *Acquisition and Inference threads* that justifies the lower number of cameras that have been managed.

Configuration 2, instead, was the heaviest configuration in terms of computational cost for the Jetson Nano and also the most performing as far as the *Inference times* were concerned. In this configuration, in fact, the *AIT* was relative

to the analysis of multiple input *batches* (one for each single *Acquisition thread*) therefore, the times reported in tables 5.6 and 5.7 were much lower than those of the configuration 1. Among the two versions of this second configuration, the most performing was CF2v2, since it allowed a slight increase in the number of *Fps* compared to the first version and a smaller analysis *TotT*. Given the large number of sub-processes that this second configuration had to manage, the model that used the MobileNetV2 backbone could almost never be used, except in a single case corresponding to the *without camera visualization* execution mode and with a maximum of two cameras identified in CF2v2. The neural network architecture that used the MobileNetV3Large backbone was rather problematic to execute as the one with MobileNetV2 as shown in the previously mentioned tables.

The best infrastructures identified were, therefore, CF1v1 for its stability of execution and CF2v2 for the achieved performances. A test was carried out for both this multi-threading configurations on the maximum number of cameras that they could manage in the *without camera visualization* execution mode using the MobileNetV3Small backbone architecture, the results of which are shown in table 5.8. As expected, CF1v1 managed a greater number of cameras, i.e. nine, although the inference and acquisition times increased drastically not allowing, therefore, to maintain the time constraints defined by the real-time use of this framework. CF2v2, on the other hand, maintained much lower times compared to the previous one, also considering the fact that the *AIT* relates to the analysis of the seven different *batches* collected, which made it the best configuration of all those identified within this thesis work.

Chapter 7

Conclusions and future work

Human Activity Recognition is an Artificial Intelligence task that is becoming more and more popular in everyday life and that can bring numerous advantages in various fields such as security, health care and home automation. Today, more and more techniques are being developed to ensure that this type of task can be performed in any environment, acquiring data from different types of sensors. In this context, the use of the Edge computing computation paradigm is particularly suitable, as it allows all these data to be processed in real-time near the place from which they are collected.

In this thesis work, a Deep Learning framework that makes use of this computation paradigm has been realized to solve the task of Indoor Human Activity Recognition through the real-time analysis of images captured by cameras. In particular, a neural network architecture has been proposed to solve this task, realized with three different backbones used for feature collection, and has been distributed and optimized on hardware with limited computational capacity that is the Nvidia Jetson Nano. In addition, two different multi-threading configurations were identified, each available in two different versions, for the management of the video streams captured from cameras connected to the edge network. Of the three used backbones, the most accurate was MobileNetV2, while the lightest was MobileNetV3Small, which had even the lowest data inference time. The neural network model with all these different backbones was tested within the Jetson Nano to analyze images captured from different cameras, ranging from two to four, for all the two different configurations, both with and without display of captured images on screen. The tests showed that the architectures with MobileNetV2 and MobileNetV3Large backbones were very resource demanding, allowing the correct management of a low number of cameras, which varies according to the

configuration adopted, that is smaller than the maximum limit chosen for the tests. The model that used the MobileNetV3Small backbone, on the other hand, allowed the correct management and analysis of all data sent by the cameras regardless of the configurations used.

Of all the configurations tested, the most stable was CF1v1, which allowed the management of a maximum of nine cameras using the model with MobileNetV3Small backbone and the *without camera visualization* execution mode, while the best performing was CF2v2, which reached a maximum of seven cameras using the same settings but keeping acceptable inference times for a real-time application unlike the previous one.

Further developments of this thesis work, as far as the improvement of the neural network model is concerned, include the choice of a more specific dataset for the training of the model in order to recognize activities belonging to a specific application domain, for example the fall detection in elderly care or the detection of break-ins and violence in the security field, and the use of a training dataset containing videos with a large environment view in order to allow the model to better recognize the activity carried out also in wide spaces. It would also be possible to vary the training *parameters*, such as the optimizer used or the number of training *epochs*, to obtain better results in *Accuracy*.

As regards the use of Edge computing by deploying the model on the Jetson Nano, it would be possible to lower the model complexity by lowering its *parameters* precision from Float Point 32 to Float Point 16, in order to reduce its size and increase the inference speed at the cost of a *Precision* and *Accuracy* loss. It would also be possible to use the H265 codec standard for videos captured by the cameras in order to reduce the RAM memory occupied by the images acquired by the different acquisition threads.

As a final consideration, it would be possible to examine other and more powerful boards than the Nvidia Jetson Nano on which to deploy the neural network model. Having greater computational resources, in fact, would allow to process more video streams transmitted through the edge devices network and to deploy more complex models able to solve the Human Activity Recognition task.

These remarks can be a good starting point for the development of a more specific and better performing framework for real-time Indoor Human Activity Recognition based on the Edge Computing computation paradigm.

Bibliography

[1] L. Liciotti and M. Bernardini et al. «A sequential deep learning application for recognising human activities in smart homes». In: *Neurocomputing* 396 (2020). DOI: https://doi.org/10.1016/j.neucom.2018.10.104 (cit. on pp. 1, 10, 17).

[2] J. Chen and X. Ran et al. «Deep Learning With Edge Computing: A Review.» In: *Proceedings of the IEEE* (2019). DOI: 10.1109/JPROC.2019.2921977 (cit. on pp. 6, 12, 13, 15).

[3] Jagwinder Kaur Dhillon, Chandni, and Alok Kumar Singh Kushwaha. «A recent survey for human activity recoginition based on deep learning approach». In: (2017), pp. 1–6. DOI: 10.1109/ICIIP.2017.8313715 (cit. on p. 7).

[4] Cem Ersoy Ozlem Durmaz Incel Mustafa Kose. «A Review and Taxonomy of Activity Recognitionon Mobile Phones». In: *BioNanoScience* (2013), pp. 145–171. DOI: 10.1007/s12668-013-0088-3 (cit. on pp. 8–10).

[5] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. «Mobilenets: Efficient convolutional neural networks for mobile vision applications». In: *arXiv preprint arXiv:1704.04861* (2017) (cit. on p. 13).

[6] Joseph Redmon and Ali Farhadi. «YOLO9000: better, faster, stronger». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271 (cit. on p. 13).

[7] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. «SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size». In: *arXiv preprint arXiv:1602.07360* (2016) (cit. on p. 13).

[8] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. «On-demand deep model compression for mobile devices: A usage-driven model selection framework». In: *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. 2018, pp. 389–400 (cit. on p. 14).

[9] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek Abdelzaher. «Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework». In: *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 2017, pp. 1–14 (cit. on p. 14).

[10] Loc N Huynh, Youngki Lee, and Rajesh Krishna Balan. «Deepmon: Mobile gpu-based deep learning framework for continuous vision applications». In: *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 2017, pp. 82–95 (cit. on p. 14).

[11] Jin Zhang, Bo Wei, and Jun Cheng. «HARaaS: HAR as a service using wifi signal in IoT-enabled edge computing». In: *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*. 2020, pp. 681–682 (cit. on p. 14).

[12] Tiffany Yu-Han Chen, Lenin Ravindranath, Shuo Deng, Paramvir Bahl, and Hari Balakrishnan. «Glimpse: Continuous, real-time object recognition on mobile devices». In: *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*. 2015, pp. 155–168 (cit. on p. 14).

[13] Angela H Jiang et al. «Mainstream: Dynamic stem-sharing for multi-tenant video processing». In: *2018 Annual Technical Conference*. 2018, pp. 29–42 (cit. on p. 14).

[14] Xukan Ran, Haolianz Chen, Xiaodan Zhu, Zhenming Liu, and Jiasi Chen. «Deepdecision: A mobile deep learning framework for edge video analytics». In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE. 2018, pp. 1421–1429 (cit. on p. 15).

[15] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. «Distributed deep neural networks over the cloud, the edge and end devices». In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 328–339 (cit. on p. 15).

[16] He Li, Kaoru Ota, and Mianxiong Dong. «Learning IoT in edge: Deep learning for the Internet of Things with edge computing». In: *IEEE network* 32.1 (2018), pp. 96–101 (cit. on p. 15).

[17] Shaojun Zhang, Wei Li, Yongwei Wu, Paul Watson, and Albert Zomaya. «Enabling edge intelligence for activity recognition in smart homes». In: *2018 IEEE 15th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE. 2018, pp. 228–236 (cit. on pp. 15, 16).

[18] D Aishwarya and RI Minu. «Edge computing based surveillance framework for real time activity recognition». In: *ICT Express* 7.2 (2021), pp. 182–186 (cit. on p. 16).

[19] Moez Baccouche, Franck Mamalet, Christian Wolf, Christophe Garcia, and Atilla Baskurt. «Sequential deep learning for human action recognition». In: *International workshop on human behavior understanding*. Springer. 2011, pp. 29–39 (cit. on p. 18).

[20] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. «Mobilenetv2: Inverted residuals and linear bottlenecks». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520 (cit. on pp. 19, 20, 22, 23).

[21] Andrew Howard et al. «Searching for mobilenetv3». In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 1314–1324 (cit. on pp. 19, 23–26).

[22] Sepp Hochreiter and Jürgen Schmidhuber. «Long short-term memory». In: *Neural computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 25).

[23] Will Kay et al. «The kinetics human action video dataset». In: *arXiv preprint arXiv:1705.06950* (2017) (cit. on p. 31).