UNIVERSITA' POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Architetture di Deep Learning per Riconoscimento di Attività Umane Indoor da stream video RGB in ambienti a risorse illimitate e limitate

Deep Learning architectures for Indoor Human Activity Recognition from RGB video stream in unlimited and constrained resource settings

Relatore:

Prof. Emanuele FRONTONI

Correlatore:

Sara MOCCIA, PhD

Tesi di Laurea Magistrale di:

Sara GIAMBERINI

Matr. 1090962

Anno Accademico 2020-2021

Sommario

Il progresso tecnologico, abbinato alla recente disponibilità di un'elevata potenza di calcolo, sta determinando un interesse della comunità scientifica sempre maggiore verso il campo dell'Intelligenza Artificiale. 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. Oltre all'introduzione di tecniche di Intelligenza Artificiale nella quotidianità, il progresso tecnologico ha anche permesso di sviluppare nuovi paradigmi di computazione in grado di supportare al meglio l'esecuzione di questo tipo di task, tra i quali si distingue l'Edge Computing.

Lo scopo di questo lavoro è lo studio di molteplici architetture di Deep Learning per il riconoscimento di attività umane indoor di alto livello con il fine di realizzare un framework che permetta di classificare le immagini provenienti da uno stream video real-time.
Un caso reale di applicazione orientata all'Edge Computing sarà poi affrontato con il framework sviluppato.

Questo lavoro di tesi è articolato come di seguito riportato.
Nel **primo capitolo** verranno presentate le tematiche trattate e gli obiettivi che il progetto si pone.
Nel **secondo capitolo** verrà data una panoramica in merito alla problematica relativa all'HAR ed all'Edge Computing. Riguardo all'HAR ci si focalizzerà sulle fasi che costituiscono la procedura di riconoscimento delle attività, sugli approcci sensor-based ed image-based proposti per risolvere questo task, nonché sui molteplici ambiti applicativi che ne hanno determinato il notevole aumento di interesse da parte della comunità scientifica. In merito al modello di computazione Edge Computing, dove l'elaborazione dei dati risiede in prossimità della fonte da cui questi ultimi sono stati generati, verranno elencate le proprietà ed i vantaggi nell'adozione di questo paradigma e le applicazioni più frequenti.
Nel **terzo capitolo** verranno presentati i diversi approcci, sensor-based ed image-based, proposti fino a questo momento in letteratura che si prefiggono di risolvere il

task dell'HAR, e verranno illustrate anche le possibili soluzioni di Edge Computing adottabili per il task stesso.
Nel **quarto capitolo** verranno discussi gli algoritmi e le architetture che sono state oggetto di studio. In particolare saranno analizzate le Convolutional Neural Network (CNN) e le Recurrent Neural Network (RNN), ponendo il focus sulla Rete Neurale Ricorrente LSTM, che sono state utilizzate per la creazione del modello utilizzato per il riconoscimento di attività umane.
Verrà presentata l'architettura del modello di rete SingleBranch usato per la classificazione delle attività umane, verrà data una breve descrizione delle differenti backbone testate per l'estrazione delle feature spaziali e sarà presentata l'architettura della rete DoubleBranch come tentativo di miglioramento del modello precedentemente creato.
Infine sarà presentato l'experimental protocol, descrivendo il dataset utilizzato per allenare le reti, i training settings e le performance metrics adottate per valutare la bontà dei modelli.
Nel **quinto capitolo** sarà discussa la procedura di ottimizzazione e deployment dei modelli ottenuti all'interno di un hardware a basso costo, evidenziando le problematiche relative a questa procedura e proponendo un meccanismo di single camera handling.
Nel **sesto capitolo** verranno presentate le performance raggiunte dai modelli creati nell'ambiente a risorse illimitate (Google Colab), ed i risultati del processo di ottimizzazione ed inferenza real-time nell'hardware a risorse limitate (NVIDIA Jetson Nano).
Infine, nel **settimo** e nell'**ottavo capitolo** saranno esposte le discussioni sui risultati ottenuti e le conclusioni tratte da questo lavoro e saranno proposti eventuali sviluppi futuri.

Summary

Technological progress, coupled with the recent availability of high computing power, is leading the scientific community to take an increasing interest in the field of Artificial Intelligence. One of the various tasks of this discipline is Human Activity Recognition (HAR), which, thanks to its versatility, is becoming increasingly popular in various fields of application such as home automation, surveillance systems and health care for the elderly. In addition to the introduction of Artificial Intelligence techniques in everyday life, technological progress has also led to the development of new computational paradigms that can best support the execution of this type of task, such as Edge Computing.

The aim of this work is to study multiple Deep Learning architectures for the recognition of indoor high-level human activities in order to create a framework that allows the classification of images obtained from a real-time video stream.
A real case of Edge Computing oriented application will be approached with the developed framework.

This thesis is structured as follows.
In the **first chapter**, the issues and objectives of the project are presented.
In the **second chapter**, an overview will be given of the problems related to HAR and Edge Computing. With regard to the HAR we will focus on the phases that constitute the procedure of recognition of the activities, on the sensor-based and image-based approaches proposed to solve this task, as well as on the many application areas that have determined the considerable increase of interest from the scientific community. With regard to the Edge Computing paradigm, where the data processing is performed close to the source from which it was generated, the properties and advantages of adopting this paradigm and the most frequent applications will be listed.
In the **third chapter** we will present the different approaches, sensor-based and image-based, proposed in the literature up to now, which have the objective to solve the HAR task, and we will also illustrate the possible solutions of Edge Computing that can be adopted for the task itself.
In the **fourth chapter** the algorithms and architectures that have been studied

will be discussed. In particular, the Convolutional Neural Networks (CNN) and the Recurrent Neural Networks (RNN) will be analyzed, with the focus on the LSTM Recurrent Neural Network, which were used for the creation of the model used for the recognition of human activities.
The architecture of the SingleBranch network model used for the classification of human activities will be presented, a brief description of the different backbones tested for the extraction of spatial features will be given and the architecture of the DoubleBranch network will be presented as an attempt to improve the previously created model.
Finally, the experimental protocol will be presented, describing the dataset used to train the networks, the training settings and the performance metrics adopted to evaluate the quality of the models.
In the **fifth chapter**, the procedure of optimization and deployment of the obtained models in a low cost hardware will be discussed, highlighting the problems related to this procedure and proposing a single camera handling mechanism.
In the **sixth chapter**, the performances achieved by the models created in the unlimited resources environment (Google Colab), and the results of the real-time optimization and inference process in the limited resources hardware (NVIDIA Jetson Nano) will be presented.
Finally, in the **seventh** and **eighth chapters**, discussions on the results obtained and the conclusions derived from this work will be presented and possible future developments will be proposed.

Acknowledgements

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, HAR has received considerable attention from the scientific community due to its many application domains, thanks also to the Internet of Things (IoT), which has made it possible to connect billions of intelligent physical objects of different categories to the Internet.
Within the IoT, in fact, a task of particularly importance is the HAR[1], which aims is to detect and classify high-level human activities executed in real-world contexts by processing spatial and temporal features extracted from data coming from different sources.
HAR techniques can be grouped according to the type of source, i.e. sensor, used for the acquisition of the data that will then be processed to allow the classification of actions. The sensors that are mainly adopted in HAR can be divided into three categories: sensors that can be connected to objects, sensors that can be worn directly by the individual and finally sensors that can be inserted into the environment.
The first category includes those sensors able to acquire logs from the use of certain objects, the second category includes sensors such as accelerometers and gyroscopes and finally the third category includes, among others, video cameras [1].
Two main types of approaches can be distinguished: those that exploit optical data, such as images from the camera stream, and those that use non-optical data, such as motion and proximity sensors.
The approach adopted in this work falls into the first type, the image-based one, and allows to exploit the stream coming from RGB cameras to analyze frames from which it is possible, through processing, to classify human actions.

When video streams have to be analyzed, it is essential to adopt solutions capable of reducing processing latency in order to produce real-time responses.

[1] https://www.redhat.com/it/topics/internet-of-things/what-is-iot

The optimal solution in this scenario is to use the computation paradigm called Edge Computing which, by bringing data processing close to where it is collected, reduces processing latency, always guaranteeing real-time responses.

The objective of this thesis is to develop and implement a framework able to offer a solution to the Indoor HAR task through the use of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) able to process the spatial and temporal features extracted from RGB images coming from real-time camera stream.
A real case of Edge Computing oriented application will be approached with the developed framework.
This work was carried out in collaboration with the company Inim Electronics S.r.l. of Monteprandone (AP), which made it possible to have and use the hardware instruments necessary to achieve the goal.

Chapter 2

Human Activity Recognition and Edge Computing

2.1 Human Activity Recognition

The term HAR defines the discipline that aims to recognize the activities, both simple and complex, of one or more agents in real world contexts starting from a series of observations, on the actions of the agents, extrapolated from sensory data of multiple nature[1].
Several approaches to HAR have been proposed by the scientific community and each one differs depending on the type of sensor, and thus the type of data, used. We distinguish between sensor-based approaches that use non-optical data and image-based approaches that use images extracted from cameras.
HAR can also be applied in many domains, such as smart homes and monitoring and surveillance systems.
In order to better understand this particular discipline, it is essential to spend some time on the semantics of what are the fundamental concepts through which the recognition of the activities is made possible.
Different categorizations are provided in the literature and two of them will be analyzed below [2].
A first categorization involves concepts such as *Action Primitives*, *Action* and *Activity*.
An *Action Primitive* can be defined as an atomic movement that can be described

[1] https://en.wikipedia.org/wiki/Activity_recognition

at the limb level.
An *Action* can be defined as the set of several *Action Primitives* that can describe a movement, possibly cyclic, of the whole body.
The term *Activity*, on the opposite, refers to a number of successive *Actions* capable of providing an interpretation of the movement performed by the agent.
To give an example, think of "Left leg forward" as an *Action Primitive* and "Running" as an *Action*. "Jumping over obstacles" at this point can be considered as an *Activity* as it contains the *Actions* of starting, jumping and running.
A second categorization is that there is a hierarchy in which activities are classified according to their complexity. This hierarchy consists of four levels: *Gestures*, *Actions*, *Interactions* and *Group Activities*.
Gestures can be considered as elementary movements of a single part of an agent's body, and therefore as atomic components able to describe the movement of a subject. "Extending the arm" and "Lifting a leg" are two examples of gestures.
The term *Action* refers to the activity of a single person, which may consist of several temporally organized single *Gestures*, such as "Running", "Walking" or "Jumping".
Interactions, on the other hand, are nothing more than *Activities* involving two or more persons and/or objects. An example of a person-person *Interaction* is "Two people kissing", while an example of a person-object *Interaction* is "A person using a mobile phone".
Finally, the term *Group Activity* refers to *Activities* which, as the name suggests, are carried out by groups of several people and/or objects, such as "A group of people walking".
Activity Recognition is therefore the task that is responsible for recognising and labelling the input data with the identified classes of human activities.

2.1.1 Stages of Human Activity Recognition

As previously mentioned, HAR is the process by which raw data acquired by sensors are interpreted to classify a given activity.
As shown in Fig.2.1, the activity recognition process can be divided into three main phases: data collection, training of a learning model and the activity recognition [3].

In order to train a HAR model, it is essential to have a collection of data to feed to it in such a way to learn the characteristics of interest for the task.
The initial data obtained from the sensors is called raw data. Due to the different nature of the sensors, this collected data may contain noise that could potentially compromise classification task. For this reason, before using the data, a Preprocessing phase (Fig.2.1 (1)) is performed to remove any noise in the data collection.
Once the data has been collected, the following phase foresees the training of a

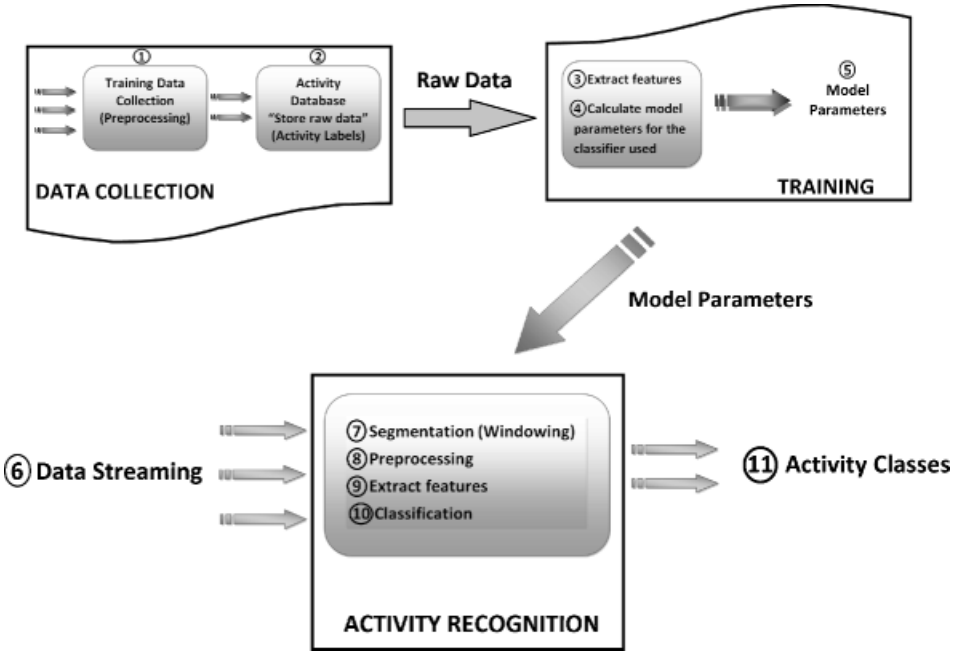


Figure 2.1: Typical steps in the Activity Recognition.
Image taken from [3].

learning model. The learning can be carried out in a supervised or unsupervised way. The first method is based on labelled observations, i.e. associated with a specific class of activity, whereas unsupervised techniques are not based on labelled data. Since a HAR system returns a label associated with a human activity class, supervised or semi-supervised approaches are usually adopted. In semi-supervised approaches a part of the model training data may be without an activity class label.

In order for the learning model to be able to associate a class of activity to the data it is given, it is necessary that the data it processes in the learning phase, is labelled with the class of action (Fig.2.1 (2)). This operation is very often a slow and tedious process and sometimes involves hand-labelling the data with the representative action class.

The learning model performs what is called Feature Extraction (Fig.2.1 (3)), which is nothing more than a form of dimensionality reduction. Through this operation the high dimensional input sensor data is reduced to a smaller set of features in order to limit the memory usage and decrease the complexity of the processing.

During the learning phase the model continuously modifies its parameters (Fig.2.1 (4)) until the optimal parameters are obtained (Fig.2.1 (5)).

Once the best parameters of the learning model have been calculated, the test can be performed on a new data stream (Fig.2.1 (6)).

Analyzing a continuous stream of data is a problem of considerable complexity. Different techniques are applied to reduce the problem: one of these is the segmentation (Fig.2.1 (7)), through which the input signal is divided into smaller time segments. On these temporal segments a Preprocessing (Fig.2.1 (8)) is carried out with which the eventual present noise is eliminated and the Feature Extraction (Fig.2.1 (9)) is carried out to transform the data into useful descriptors for the classification (Fig.2.1 (10)).
The result of the process is the return of the label showing the class of human activity detected by the model adopted (Fig.2.1 (11)).

2.1.2 Approaches to Human Activity Recognition

Several approaches to HAR have been proposed by the scientific community and each of them differs from the others depending on the type of sensor used. The sensors adopted can be worn directly by the agent (wearable sensors), attached to objects in the environment or fused into the environment itself [1].
Therefore, two main categories can be distinguished, which differ from each other in the use of non-optical and optical sensory data: the first category includes sensor-based approaches, which include sensors worn by agents and those attached to objects in the environment, and the second category includes image-based approaches, which involve the use of sensors directly fused within the environment.

Sensor-based approaches

As already mentioned, this category of approaches includes sensors worn by the agent and sensors attached directly to objects in the environment, which detect non-optical data. Wearable sensors are special sensors that are attached to certain parts of the agent's body to capture movements and recognize simple activities such as walking and sitting.
These sensors include accelerometers, which can detect a change in the speed of the body part concerned, gyroscopes, which can detect angular rotation around an axis, and magnetometers which, combined with accelerometers and gyroscopes, can track the orientation of a body with respect to all three axes.
Technological evolution has brought major changes in this respect, making it possible to use smartphones, smart watches and fitness bracelets as wearable sensors [2]. These devices, in fact, are equipped with a wide variety of sensors that can keep track of agent activities, being carried all day long by the agents themselves.
The disadvantages in the use of this type of sensors lie in the fact that, being mobile devices, they need to be powered by batteries that must be periodically

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

recharged. Moreover, given the nature of these devices, the agent is forced to wear them constantly, making the approach rather invasive. The possible forgetfulness of the agent in wearing the devices should also be underlined.
However, this type of devices protects the privacy of the agent, as it does not record audio, video or any physiological information, and the new generation devices are, moreover, equipped with hardware sufficiently able to perform some elaborations internally, obtaining good results in terms of performance and accuracy [4].

Wearable sensors provide important information regarding the movement and physical exercises of the agent, but do not provide any information about the agent's interaction with objects. For this reason, the scientific community has focused its attention on smart homes, in the sense of homes equipped with sensors connected to objects in the environment. This category includes sensors that are connected to, among others, furniture, walls or doors. For example, an accelerometer attached to certain objects or doors to detect the moment when the agent interacts with them.

Image-based approaches

This category includes sensors that allow the extraction of optical data.
RGB or RGBD cameras are examples of sensors that fall into this category as devices fused within the environment.
The advantage of adopting this approach is that, unlike the sensors described in sensor-based approaches, the cameras are directly connected in the environment, making possible a passive analysis of the agent's activities, who does not have to remember to wear the device or to recharge its batteries. These devices are much less invasive than the previous category, allowing agents to perform activities as if the system did not exist.
The use of these sensors, on the other hand, means that a first Preprocessing phase is necessary in order to remove uninteresting information such as, for example, information related to the environment and not to humans.
In this thesis work the image-based approach is followed. In the following chapters, the management modalities of the input stream will be discussed and a real application will be seen through the use of RGB video camera.

2.1.3 Real-world applications

HAR is a discipline that has been enjoying considerable success in recent years thanks to its many application domains.
The application domains are among the most varied, but can be summarized in three main categories: Ambient Assisted Living (AAL) systems for smart homes,

applications for monitoring and healthcare, and monitoring and surveillance systems for indoor and outdoor activities [1].

Ambient Assisted Living for smart home

The term Ambient Assisted Living[3], AAL for short, refers to the set of technological solutions aimed at making the environment in which we live active, intelligent and cooperative, capable of providing greater safety, wellness and satisfaction in carrying out the activities of daily life. The main objectives of AAL include increasing the autonomy and self-sufficiency of people living in that environment, increasing the functional capacity and health assurance of older people, and promoting better and healthier lifestyles for people at risk.
Using HAR techniques, smart homes are able to provide independence and comfort to residents, exploiting different types of technological devices interconnected within the network that are able to communicate and learn through the habits of the user.

Monitoring and healthcare

In recent years, HAR has become very important in the field of care, due to the progressive ageing of the world's population as a result of the development of medical science and technology [1]. This leads to an increase in the demand for medical personnel. The use of HAR systems can increase the autonomy and efficiency of existing monitoring systems and reduce the need for human intervention.
HAR systems used in the care sector typically perform activities such as identifying potentially dangerous activities or situations for the elderly person, e.g. by Fall Detection, and alerting specialized personnel to ensure rapid intervention.

Indoor and outdoor monitoring and surveillance

Nowadays, traditional surveillance systems are characterized by the presence of a human operator who supervises and monitors the environments concerned, signalling any critical situation by means of an alarm.
Usually these operators have to spend a very high number of hours in front of the monitors, leading to a potential loss of attention that could result in a failure to detect a critical situation.
In order to limit potential distractions of the human operator and to make his workload easier, the integration of automatic HAR techniques in video surveillance systems would be a good solution to automate these systems, which would then be able to send autonomous alarm signals to specialized personnel or law enforcement

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

agencies in case critical situations are detected.
These types of systems can be used in industry to supervise production plants and send alarms in the event of breakdowns or malfunctions, but also for security purposes in public places, such as events or concerts, or at a private level by reporting theft or attempted break-ins to the police.

2.2 Edge Computing

Today's most commonly used devices, such as smartphones and Internet-of-Things sensors, generate large amounts of data that in many cases need to be analyzed in real-time using Machine Learning or Deep Learning techniques. However, inference requires significant computing resources to be performed quickly. Edge Computing is a valid solution to satisfy the high computation and low latency requirements of Deep Learning on edge devices and also provides additional benefits in terms of privacy, bandwidth efficiency and scalability. Edge Computing is a distributed computing model in which processing takes place close to where the data is collected. The principle on which it is based is to keep the data collection and processing operations physically close to each other, against the principle of Cloud Computing which aims to perform the processing in one central data center to which all data must be sent.

2.2.1 Properties and advantages of Edge Computing

Through the use of many small data centers, located close to the sensors or devices, capable of collecting and analyzing data autonomously, it is possible to obtain an architecture composed of multiple nodes that guarantees a considerable reduction in network costs and bandwidth constraints, a decrease in latency times 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 greater proximity of service delivery to users and a simpler and more immediate control of the data[4]. Thanks to this decentralized structure and the proximity to the data source, Edge Computing guarantees the following properties[5]:

- Latency: a low response time or real-time inference is essential for certain types of tasks. The adoption of Edge Computing in these cases is the optimal choice to carry out these types of tasks because the presence of local mini-data centres connected directly to the same network of sensor and cameras allows to

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

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

greatly reduce latency and data transmission times. For example, frames from the camera of an autonomous vehicle need to be processed very quickly to detect and avoid obstacles, or a voice-assistance application needs to quickly analyze and understand the user's request and return a response;

- Reliability: Edge Computing systems guarantee different levels of reliability and the activation of alarm mechanisms in the event of a malfunction of data centres. It is possible to detect a device that is not working properly and carry out timely repair or replacement operations without affecting the operation of the others. Edge Computing keeps the various services offered by the systems active even in the event of Internet connection problems. This is possible because the entire processing takes place locally;

- Scalability and Modularity: in a centralized architecture, sending data from sources to the Cloud can introduce scalability issues due to the fact that the access to the Cloud can become a bottleneck as the number of connected devices increases. Edge computing ensures easy expansion of processing capacity and task variety due to the large number of IoT devices and small perimeter data centres that will be additional nodes to add to the existing network[6];

- Security and Privacy: a protection policy must be implemented for each small data centre in the local network or directly on the data collection and processing device. Edge computing allows data to be analyzed close to the source, for example from a trusted local server, avoiding the need for data to travel across the public Internet, thus reducing exposure to possible attacks.

The advantage of having several decentralized processing devices is that a potential attacker would not be able to obtain data from all the sensors, in the event that he tries to break the security of a single device. However, this structure has several disadvantages. Edge devices will have less hardware power availability, which may limit the activities that can be performed. Moreover, each device will have to be provided with its own security methods and protocols.

2.2.2 Applications of Edge Computing

Edge computing is an excellent computing paradigm for all those tasks that cannot be managed efficiently by a centralized approach. In fact, many of the limitations

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

of the centralized approach lie in the need to obtain results in real-time. Some examples of applications that exploit Edge Computing will be presented[7].

Artificial Vision

Image Classification and Object Detection are fundamental tasks of Computer Vision and are applicable in many specific domains, such as video surveillance, vehicle detection or object counting. The data to perform these tasks comes from cameras, so it is necessary to analyze real-time video streams using Artificial Intelligence algorithms. Many of these tasks need to be completed in the shortest possible time in order to ensure that, in the event of anomalies, an alarm signal is generated and the necessary action is taken. An example of a vision system is Vigil [5] which is characterized by the presence of a network of cameras connected via wireless. Within each node of the Edge architecture, algorithms are run to select interesting frames. The selected frames are analyzed, for example, to find missing persons in surveillance cameras or to analyze customer queues in retail environments.

Self-driving Cars

Many of the latest generation of vehicles incorporate Deep Learning algorithms to perform tasks related to autonomous driving. While driving, a lot of data is collected and needs to be analyzed in real-time in order to promptly perform various vehicle operations such as changing the trajectory, avoiding obstacles or performing a forced stop.

Natural Language Processing

Deep Learning has become popular not only with images and videos but also with tasks for Speech Synthesis, Recognition and understanding of different parts of a sentence and Automatic Translation. An example of Natural Language Processing on the Edge are voice assistants, such as Amazon Alexa or Apple Siri [5]. Although these voice assistants perform most of their processing in the Cloud, they typically use on-the-edge processing to detect passwords such as "Alexa" or "Hey Siri". The voice recording is sent to the Cloud to perform further analysis and produce a response only if the keyword is detected. In the case of Apple Siri, keyword processing uses two Deep Neural Networks on the device to classify speech into one of 20 classes.

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

Augmented Reality

In Augmented Reality (AR), Deep Learning can be used to detect objects of interest in the user's field of view and apply virtual overlays. The use of an Edge Computing platform is essential to support Augmented Reality services by providing highly localized data specific to the user's point of interest[8].

Financial Transactions

Real-time analysis is also required in the financial sector. POS financial transactions are captured and analyzed in micro data centres within bank branches in order to identify anomalous transactions that must be intercepted as quickly as possible and blocked.

2.3 Aim of the thesis

The aim of this thesis is to carry out a study on Deep Learning architectures in order to build a model for Indoor HAR. CNNs for feature extraction and a RNN (LSTM) for the analysis of dependencies between frames coming from videos will be analyzed.
Concerning the feature extraction, different types of backbones will be analyzed and their differences in complexity (number of *parameters*) and effectiveness will be compared.
In addition, a two-stream network typology will be introduced to try to improve the performance of the previously created model.
Once the model has been obtained, we will proceed with the testing phase both in an environment with "unlimited" resources (Google Colab) and in an environment with limited resources (NVIDIA Jetson Nano).
A real-case of application oriented to Edge Computing will be handled with the developed framework.
For inference on hardware with limited resources, the procedure of optimization and deployment of the model on the Jetson Nano will be presented, underlining what are the problems with the use of low cost hardware.
Finally, the results obtained in the "unlimited" and limited resource environment will be compared, focusing on the compromises that have to be accepted to work in a limited resource environment.

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

Chapter 3

State of the Art

HAR is one of the main tasks of Computer Vision, due to its wide applicability in everyday domains. For this reason, the scientific community is paying particular attention to this task, proposing several methodologies to solve it.
In section 3.1 sensor-based approaches, that analyze non optical data to detect information useful for the recognition of human activity, will be discussed while in section 3.2 image-based approaches, that use optical data such as images to recognize human activity, will be presented.
Finally in section 3.3 approaches to HAR using microprocessors will be presented. Also in section 3.3 various Edge Computing architectures that can best accomplish the various Deep Learning tasks will be proposed.

3.1 Sensor-based methods

As previously mentioned, sensor-based methods exploit sensory data, such as motion sensors, temperature sensors, door sensors, etc., placed inside a smart home or worn directly by the agent, to detect information useful for the recognition of human activity.

The authors of the paper [6] propose Machine Learning techniques for HAR using sensor data from smart watches. The data obtained comes from sensors such as accelerometer, gyroscope, pedometer and heart rate sensors. The approach suggested in the paper is to apply Principal Component Analysis (PCA) to select the most characteristic features and speed up the classification process. The Machine Learning techniques tested are: Random Forest (RF), Support Vector Machine (SVM), C4.5, K-nearest neighbor (KNN). The different models were trained on a set of 2800 samples of different daily human activities extracted from five participants. In the RF algorithm the number of trees was chosen as 10, in the SVM the kernel function RBF (Radial Basis function) was set as kernel function,

in C4.5 the maximum number of trees was set to 100 and in the KNN the k-value was set to 5.
The best result was obtained with the RF algorithm.
The innovation introduced by the study conducted by this paper is in the use of a hybrid method characterized by the use of PCA, for the creation of higher quality features and for the reduction of the dimensionality of the data, in addition to the RF algorithm.
Furthermore, this study makes use of pedometer and heart rate sensors in addition to the traditional accelerometers and gyroscopes. The study demonstrated that the use of these sensors led to an increase in classification accuracy.
However, one of the limitations of this work is that a real-time application for activity recognition with the proposed method has not yet been implemented.

In [7] two architectures for HAR using data from accelerometers and gyroscopes in smartphones are proposed and compared. A first architecture uses a CNN for feature extraction and SVM as classifier, while the second architecture consists of a CNN and a combination of Fully-Connected layer and Softmax classifier.
The models were trained on a dataset of 7352 examples consisting of sensory data from tri-axial accelerometers and gyroscopes. The data were collected from 30 volunteers who performed six different activities while holding a smartphone in their pocket.
The Deep Learning architecture consisting of CNN and Fully-Connected layer achieved a very high performance with an accuracy level of 99.66%. This architecture proved to be able to classify even similar activities that were considered very difficult to classify. The SVM classifier architecture, on the other hand, performed slightly less well at 94.91% and failed to classify similar assets.
A negative aspect highlighted by the authors of the paper is that the addition of Convolutional layers does not necessarily lead to an increase in performance.

The scientific community has proposed several solutions of Machine Learning that adopt classification algorithms such as Naive Bayes, Random Forest, K-NN and SVM, but they are static models that are not able to evolve and adapt to the changing environment. For this reason the Deep Neural Networks (DNNs), the CNNs, the RNNs and the Long Short Term Memory (LSTM) have become the most used techniques of Deep Learning to solve the task of HAR.
In [1] multiple versions of the LSTM are proposed, which has proved to be one of the best solutions for the HAR in the case in which the input data comes from sensors present in the surrounding environment. A first version analyzed of the LSTM is the Uni-LSTM which is nothing more than an RNN architecture with the addition of a hidden layer of LSTM cells. Then more complex versions are analyzed, such as the Bi-LSTM that foresees the presence of an LSTM architecture with forward and backward connections able to extract information from the past

and the future, modelling at the best the temporal dependencies of the features, the Casc-LSTM whose input layer is a Bi-LSTM cascaded with a Uni-LSTM, the Ensemble2-LSTM which combines the output of a Bi-LSTM and a Uni-LSTM, and finally a CascadeEnsemble which is an architecture consisting of a cascaded Ensemble2-LSTM and a Uni-LSTM. In Fig.3.1 is shown the architecture proposed by the authors. It includes a first phase of Preprocessing, in which operations of Filtering and Aggregation of the input data are carried out, and a second phase of Classification which, through the use of the LSTM and of a *Softmax* layer, is able to predict the class of activity. Thanks to the capacity of this typology of network to capture the long-term temporal dependencies of the human activities, all the versions of the LSTM architecture have proved to be more effective in the prediction of the human activity, both of the existing techniques of Machine Learning and of those of Deep Learning.
The authors of the paper show that the ability of LSTM to automatically extract spatio-temporal informations leads to a significant improvement in accuracy compared to ML approaches.
However, increasing the complexity of LSTM-based models has not always led to a significant improvement in performance.

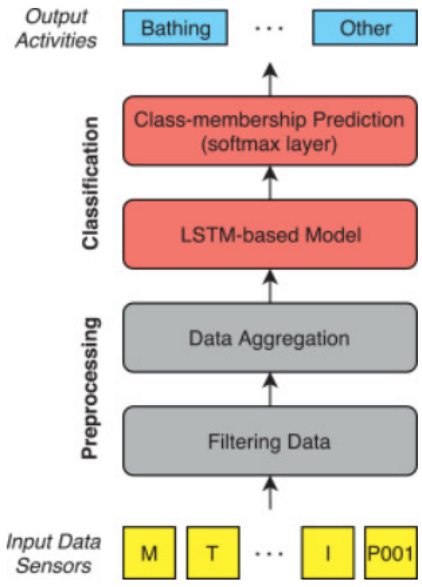


Figure 3.1: HAR Architecture.
Image taken from [1].

3.2 Image-based methods

Differently from sensor-based methods, image-based methods use image analysis to detect human activity. Usually, these images come from video camera streams located within the environment in which the agent is interacting.
Also for this type of methods there are several solutions in the literature that adopt Machine Learning and Deep Learning techniques.

The authors of the paper [8] propose an approach based on the use of the SVM classifier to classify human activities from a video stream. The approach involves analysing consecutive frames. An image is produced as a result of the pixel by pixel difference of the consecutive images obtaining a Region of Interest (RoI). On the RoI a measure for the evaluation of the movement is calculated and only the portions of RoI with the greatest value of the movement measure will be considered. The feature vector will then be passed to the SVM classifier.
Two versions of the SVM were tested: Polynomial SVM and RBF SVM. The training of the models was performed with a subset of the KTH dataset restricted to four scenarios related to 7 persons.
The results show that the RBF SVM performed better than the Polynomial SVM, with an accuracy of 94.58% compared to 87.92%.
This approach was compared with other works in the literature, obtaining comparable performance.
An issue with this approach is that this architecture is not able to recognize similar activities, such as running and walking, with high accuracy.

In [9] and [10] alternative methods are proposed that try to increase the performance of the activity recognition model. In this case the architecture consists of two streams, according to the hypothesis that the human visual cortex contains two pathways: the ventral stream (which deals with Object Recognition) and the dorsal stream (which deals with Motion Recognition).

The architecture proposed in [9] is shown in Fig.3.2. It is based on streams to process data, both implemented as ConvNets: a spatial stream is in charge of performing the recognition of the scene and the objects depicted in the video, while a temporal stream is in charge of recognizing the activity from the motion detected by a dense optical stream. The two streams are then combined through two fusion methods: averaging and a trained multi-class linear SVM.

As can be seen in Fig.3.2, the inputs to this model are both frames and optical flows between consecutive frames. The optical flow (Fig.3.3) can be considered as a set of displacement vector fields between pairs of consecutive frames. In this way the input explicitly describes the movement between video frames. The paper shown that this type of architecture significantly outperforms many state-of-the-art

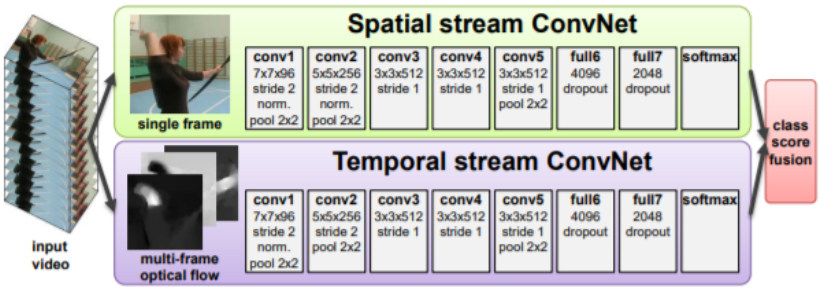


Figure 3.2: Two-stream architecture for video classification.
Image taken from [9].

architectures.
Some negative aspects relate to the fact that using bi-directional flow is not beneficial in the case of ConvNet fusion and that the model was trained on a relatively small data set.

Also in [10] a two-stream network architecture is proposed with the use of a Recurrent LSTM Network for the classification of human activities. This is determined by the fact that a structure capable of extracting both spatial and temporal information has proven to be a competitive approach in dealing with video comprehension problems. In recent years, many studies in the literature have proposed the introduction of optical flow, as a complement to RGB frames, achieving a significant improvement in performance.

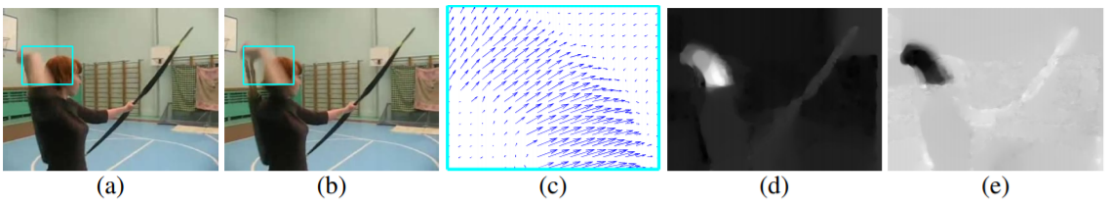


Figure 3.3: Optical flow. (a),(b) pair of consecutive frames with part of the moving body highlighted by a cyan coloured rectangle; (c) dense optical flow of the highlighted area; (d) horizontal component of the displacement vector field; (e) vertical component.
Image taken from [9].

As shown in Fig.3.4 the architecture consists of two flows implemented as ConvNet: the spatial flow is fed by the RGB frames sampled from the video, and the temporal flow is fed by an optical flow that will be calculated from the frames. The resulting feature maps are then merged to obtain feature maps that contain both spatial and temporal information. Finally, the feature maps are fed into a ConvLSTM which is used to further learn spatio-temporal dependencies. A

ConvLSTM is a variant of the traditional LSTM, whose gates are replaced by convolutional gates.
The authors demonstrated that, using two stream network + ConvLSTM, spatiotemporal correlation information is kept through the whole feature map learning process, achieving state-of-the-art accuracy on both UCF101 and HMDB51 without using Kinetics to pre-train the model. On the other hand, large memory and GPU cost needed cause the proposed network is a two-stage network; furthermore, this network may not work well on those dataset whose length of video sequences largely varies.

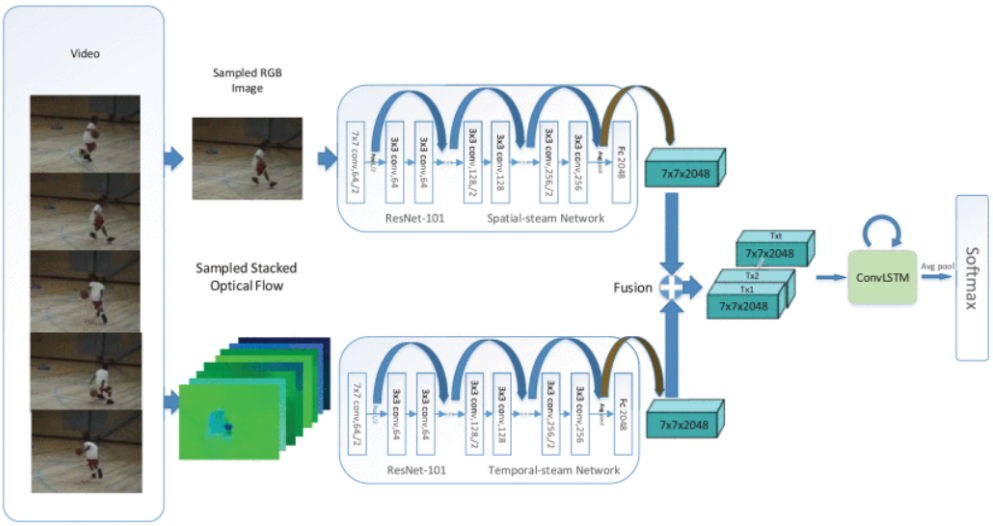


Figure 3.4: Proposed HAR framework.
Image taken from [10].

In [11] a network architecture made of two parts is proposed. As it is possible to observe in Fig.3.5, the input of the architecture is composed by a sequence of frames that is fed to the first part of the architecture, characterized by a CNN and a RNN, that is focused on the extraction of the spatial and temporal features, and a second part characterized by *Dense* layers to classify the human activity.

The classic Convolutional blocks are replaced by the 3D Convolutions, which are nothing else than extensions of the latter, able to learn the spatio-temporal characteristics of a number of consecutive frames (for example less than 15) coming from video inputs. An LSTM is proposed as RNN for its capacity to consider the context using recurrent connections in the hidden layers. In the paper it is also shown how, with the use of the LSTM Classifier, significant performance improvements are obtained with respect to the best correlated results, but it should be remarked that the model was tested on a dataset containing simple actions.

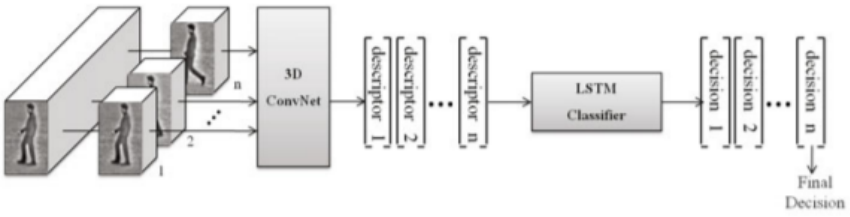


Figure 3.5: Two-steps Neural Network scheme.
Image taken from [1].

3.3 HAR and Edge Computing

The inference of a model of Deep Learning is a computationally expensive operation that requires high resources of calculation in order to be executed in the shortest possible time, due to the high number of calculations that must be performed and the high dimensionality of the input data.
This problem is even more evident when the inference has to be performed on the final devices, which are very often low-cost hardware with limited computing power.
Edge Computing is the best solution to overcome the problems related to high computational costs and low latency for inference on final devices.
Different architectures are proposed in the literature to speed up the inference of Deep Learning models [5]: On-Device Computation, Edge Server-based Architecture and Joint Computation among End Devices, Edge Servers and the Cloud.

On-Device Computation

The paper shown that this architecture requires Deep Neural Networks to be executed directly on the end device in order to reduce latency times (See Fig.3.6). Several aspects must be taken into consideration:

- Model Design: when deciding to make inference within resource-limited devices, Deep Learning models must be designed to reduce memory occupation and execution latency. This can be done by reducing the number of model *parameters*, minimizing the loss of *Accuracy*.
  Several Deep Learning models for resource-constrained devices have been proposed by the scientific community, and among these the best known are MobileNets, Single Shot Detector (SSD) and YOLO. In particular, MobileNets reduce the number of calculations to be performed by decomposing Convolution filters into simpler operations;

- Model Compression: Deep Learning model compression is another technique to perform inference in edge devices. Compression techniques include *parameter* quantization, *parameter* pruning and knowledge distillation.
  Parameter quantization involves compressing model *parameters* from floating point numbers to low bit-width numbers, thus limiting costly floating point multiplications. The pruning is the operation with which the less important *parameters* are removed, for example, those very close to 0. Finally, the distillation of the knowledge foresees the creation of a smaller Deep Neural Network that imitates the behaviour of a larger and more powerful one;

- Hardware: it is important to design the hardware in such a way as to make memory accesses efficient, reduce power consumption and latency.

In [12] a framework for HAR task as a Service using WiFi signal was developed. The on-board intelligence of IoT devices was exploited to perform inference, using a Raspberry Pi connected to a WiFi network, on data collected from IoT devices. The CNN model was tested both using the Raspberry Pi resources and using the Raspberry Pi and the Intel Neural Compute Stick2 resources combined. Neural Compute Stick2 is a device developed by Intel that is able to accelerate the inference of the models. Accurate results with latency time reduction and large performance boost were archieved, but it is necessary to design lightweight Machine Learning and Deep Learning algorithms.

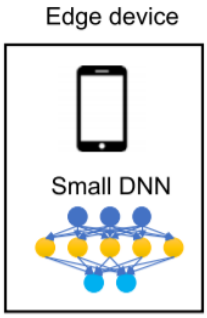


Figure 3.6: On-device Computation.
Image taken from [5]

Edge Server Computation

The paper described the Edge Server Computation architecture that involves sending data from end devices to one or more on-board servers for computation. The end devices then send their data to a neighbouring edge server that processes it and receives the corresponding results.

In this scenario, data pre-processing is required to reduce data redundancy and communication time.

Computing Across Edge Devices

The paper described also the Computing Across Edge Devices. Several strategies can be adopted in this architecture: intelligent Offloading, partitioning of the Deep Learning model, use of edge devices plus the Cloud and distribution of computation:

- Offloading: as can be seen in Fig. 3.7, Offloading is combined with the inference of a more complex Deep Neural Network present on the edge server and a lighter and simpler Deep Neural Network available on the final device;

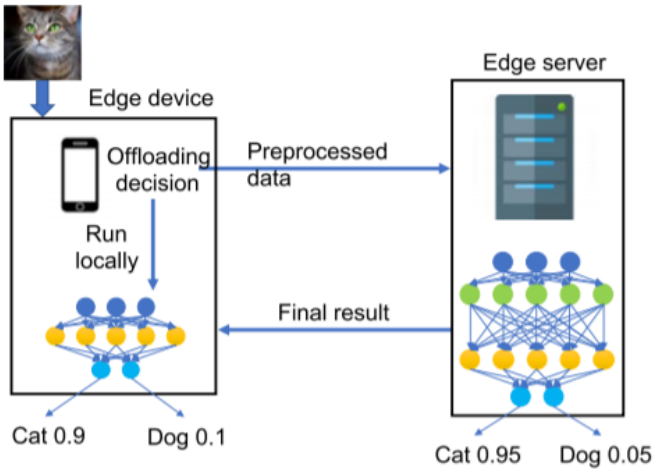


Figure 3.7: Offloading with model selection.
Image taken from [5]

- DNN Model Partitioning: with this strategy the model is partitioned in such a way that some layers are computed directly on the device and others are computed on the Edge Server or in the Cloud (See Fig.3.8). With this approach, significant latency reductions can be achieved by exploiting the computing power of other devices;

- Edge Devices Plus the Cloud: with this strategy the inference of a Deep Learning model can be performed not only on the Edge devices but also in the Cloud, as shown in Fig.3.8. This type of architecture can potentially decrease the total processing time.
  By exploiting the partitioning of the Deep Learning model, some layers can be executed on the end devices, and others on the Edge Server and the Cloud. In this way, the Edge Server receives the input data and performs the processing

of the model layers and then sends the intermediate results to the Cloud which, after processing the upper layers, sends the final results to the end devices;

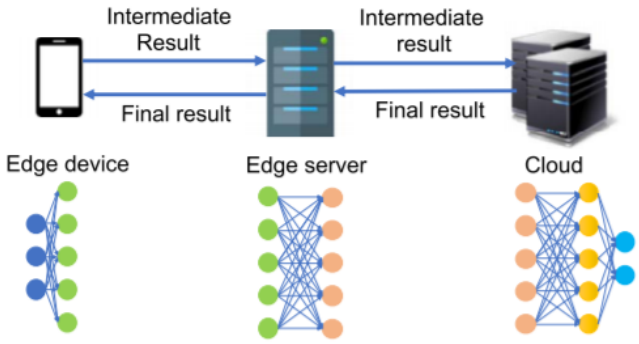


Figure 3.8: Computing Across Edge Devices with DNN model partitioning. Image taken from [5]

- Distributed Computation: this last strategy requires that the computations of the Deep Learning model are distributed over several edge helper devices, as shown in Fig.3.9.

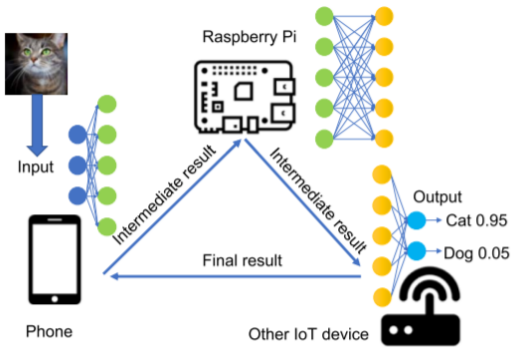


Figure 3.9: Distributed Computing with DNN model partitioning. Image taken from [5]

Chapter 4

Materials and Methods

In this Chapter we will illustrate the Deep Learning techniques adopted that in the literature have proved to be the best solution to solve the task of HAR, focusing on CNNs and RNNs.
The network architecture designed for this thesis work will be presented and the different backbones tested for spatial feature extraction will be illustrated.
A two-stream network architecture will also be proposed as an attempt to improve the previously illustrated network model.
The dataset used for training the previously developed models will be shown, as well as the training configurations adopted and the performance metrics used to evaluate the models during testing.

4.1 Convolutional Neural Networks (CNNs)

CNNs are Artificial Neural Networks used for automatic learning in Deep Learning. The behaviour and architecture of CNNs are inspired by the organization of the connections of the neurons within the human brain and are designed to learn automatically and adaptively the spatial hierarchies of features from low and high level models.
Due to their high performance in the literature, CNNs are used in various fields, from Object Classification to the Detection of lymph node metastases and skin lesions [13].
The objective of CNNs is to analyze images and reduce them to a form that is easy to process without losing those features that are fundamental to obtain a good prediction of the class.

Fig.4.1 shows the architecture of a CNN[1]. As it is possible to observe, the architecture takes in input an image and it is composed by two parts: a first part in which the operation of feature extraction is performed and a second part in which the classification is performed and, through it, the class of the input image is given in output.

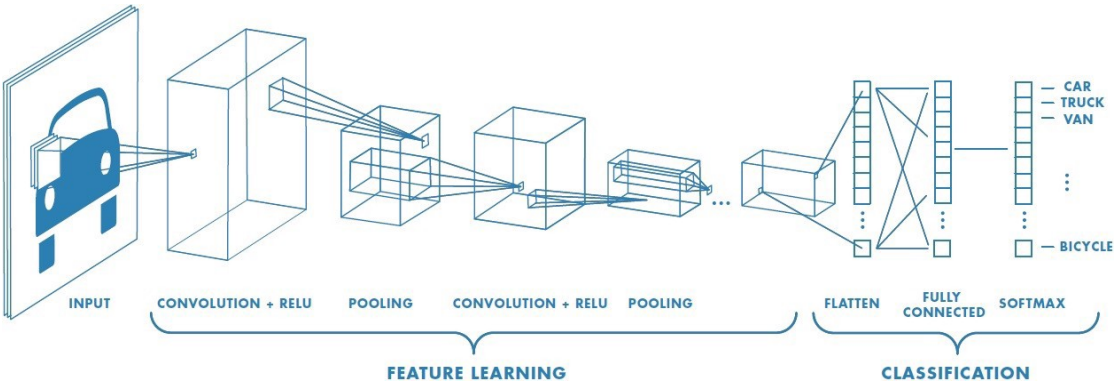


Figure 4.1: CNN architecture.

The input for this architecture is an image in Grayscale or RGB or HSV or CMYK format. In the case of an RGB image, it is divided into the three colour planes, as shown in Fig.4.2, and represents the input tensor in the form of a matrix of pixel values.

Feature Extraction

Feature extraction is the process of converting input images into a set of interesting features. When the input data is large, it is necessary to perform this dimensionality reduction operation to reduce the number of variables involved and to simplify the cost of resources required.
The part of the network that performs the feature extraction is also called backbone and is typically composed of *Convolutional* and *Pooling* layers.
Convolutional layers perform feature extraction through a combination of linear and non-linear operations, respectively *Convolution* and *Activation Function* [13]. With the *Convolution* operation, many small matrices called kernels (Fig.4.3 in yellow) are applied to the input image (Fig.4.3 in green) and, through matrix multiplications and sums, produce many feature maps (Fig.4.3 in red).

The feature maps obtained from the *Convolution* process are then passed through a non-linear *Activation Function* that takes the feature map in output to

[1]https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

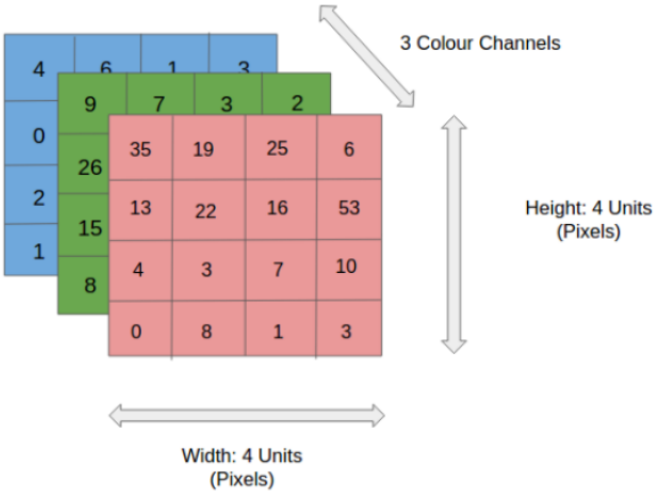


Figure 4.2: A 4x4x3 RGB image.

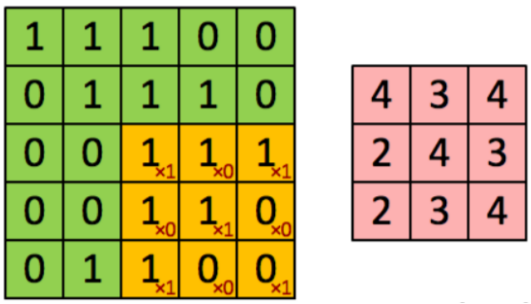


Figure 4.3: *Convolution* operation. On the left, the kernel (in yellow) is applied to the matrix of pixels representing the image (in green), producing the feature map in red on the right.

a convolution and transforms it applying non-linearity. The *Activation Function* helps a Neural Network to learn the relationships and dependencies in the data and allows the output to be limited to a particular range.
There are different types of *Activation Functions*: the *Sigmoid* and the *Hyperbolic Tangent* are the most famous because they have a mathematical representation that imitates the behaviour of a biological neuron, but the most used *Activation Function* is the *Rectified Linear Unit* (*ReLU*), due to its simplicity and its effectiveness with respect to the others.
The layer of *Pooling* instead executes an operation of downsampling that reduces the dimension of the feature maps and therefore the parameters that must be learnt in the training phase. There are two types of *Pooling*: the *Max Pooling* and the *Global Average Pooling*.
As shown in Fig.4.4, the difference between these two types of *Pooling* is that with

Max Pooling, the matrix is divided into blocks and only the highest value of each block is retained, while with *Global Average Pooling*, the average of the values in each block is calculated. Both types of *Pooling* make it possible to reduce noise and the risk of overfitting by keeping only the areas with the highest activation.

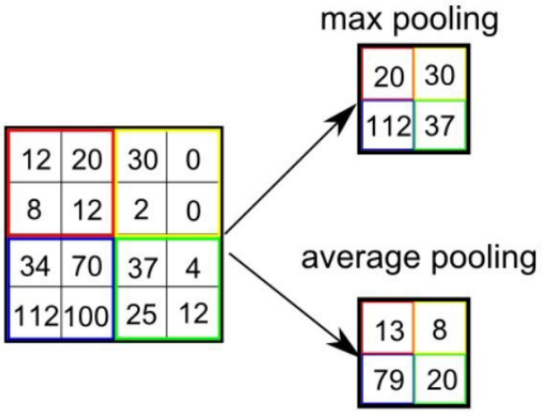


Figure 4.4: *Max* and *Global Average Pooling*.

Classification

After the *Convolutional* and *Pooling* layers, a layer of *Fully-Connected* neurons is added, preceded by a *Flattening* layer. The *Flattening* layer flattens the output feature maps of the previous layers into a one dimensional array and connects them to one or more *Fully-Connected* layers. The *Fully-Connected* layers are also known as *Dense* layers and each input is connected to each output by a learnable weight in the training phase.
The mapping between the feature maps and the output neurons of the network is characterized by the probabilities of each class.
The final *Fully-Connected* layer has typically a number of output neurons equal to that of the classes.
An *Activation Function* is applied to the last *Fully-Connected* layer, which is usually different from those previously discussed and depends on the task to be carried out. If we have a task of multi-class classification, the *Activation Function* applied is usually the *Softmax* that normalizes the real values of the output from the last *Fully-Connected* layer belonging to each class. Each value varies between 0 and 1 and the sum of all values is equal to 1.

4.2 Recurrent Neural Networks (RNNs)

RNN is a type of Neural Network able to analyze time series or sequential data and predict future trends.
This class of Neural Networks is typically used for tasks in which data are temporally ordered, or in general when dealing with sequences of data. Some examples of applications are: Speech Recognition, Natural Language Processing, Language Translation and Activity Recognition.
They can be applied in several popular technologies such as Google Translate and Apple Siri[2].
RNNs differ from Feedforward Neural Networks, such as CNN, in that the data does not flow in only one direction, from the input layer to the output layer, but the architecture allows the data to flow back to the input layer. This means that the output produced at the previous time step will become part of the input for the current time step.
Considering a simple RNN[3] constituted by a single neuron that receives an input, processes it, produces an output and sends it back to itself as a new input (Fig.4.5 on the left).

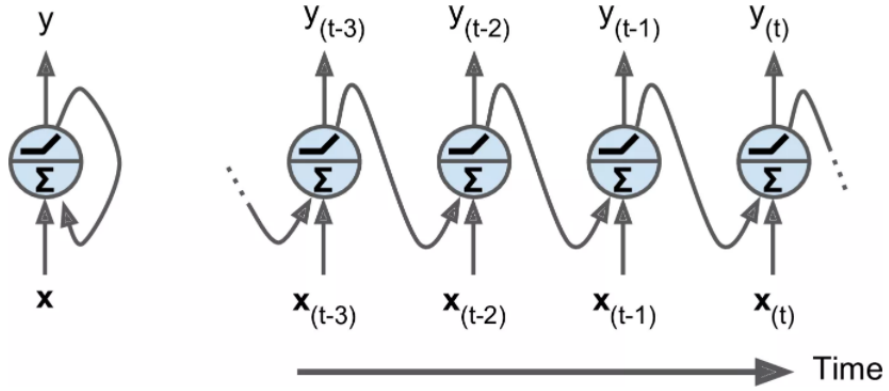


Figure 4.5: A Recurrent Neuron (left), unrolled through time (right).

At each time step (t), defined also with the term *frame*, the recurrent neuron receives as input x(t) and the output produced by itself at the previous time step (y(t-1)).
To better understand the functioning of this network constituted by a single recurrent neuron, we can use a method called *unrolling the network through time*, through which it is possible to observe the flow of input received and output

[2]https://www.ibm.com/cloud/learn/recurrent-neural-networks

[3]https://andreaprovino.it/rnn-recurrent-neural-network/

produced during each time step (Fig.4.5 on the right).
Once the functioning of a single recurrent neuron is understood, modelling a layer of recurrent neurons becomes easier.
This time at each time step t, each neuron receives as input a vector $\mathbf{x}(t)$ and the output vector $\mathbf{y}(t\text{-}1)$ produced in the previous time step.
Two weight sets, or weight vectors, have to be elaborated by each recurrent neuron present in the layer:

- $\mathbf{w}$x, for the input $\mathbf{x}(t)$;
- $\mathbf{w}$y, for the output $\mathbf{y}(t\text{-}1)$ of the previous time step.

To obtain the formula for the computation of the output vector of a layer of the network, we insert the weight vectors in two weight matrices $\mathbf{W}$x and $\mathbf{W}$y, we add a *bias* term $\mathbf{b}$ and the *Activation Function* for example *ReLU* ($\phi(-)$).
The formula obtained is as follows:

$$\mathbf{y}_{(t)} = \phi(\mathbf{W}_x^T \cdot \mathbf{x}_{(t)} + \mathbf{W}_y^T \cdot \mathbf{y}_{(t\text{-}1)} + \mathbf{b})$$

To calculate the output of a single mini-*batch*, all the inputs of a time step t are inserted into an input matrix $\mathbf{X}(t)$, giving the following result:

$$\mathbf{Y}_{(t)} = \phi(\mathbf{X}_{(t)} \cdot \mathbf{W}_x + \mathbf{Y}_{(t\text{-}1)} \cdot \mathbf{W}_y \cdot + \mathbf{b}) = \phi([\mathbf{X}_{(t)}\mathbf{Y}_{(t\text{-}1)}] \cdot \mathbf{W} + \mathbf{b}) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{W}_x \\ \mathbf{W}_y \end{bmatrix}$$

where:

- $\mathbf{Y}(t)$ is an $m \times$ *n-neuron* matrix containing the output of the layer at time step t for the mini-*batch* (m is the number of instances and n is the number of neurons);
- $\mathbf{X}(t)$ is an $m \times$ *n-inputs* matrix containing all instance inputs (*n-inputs* is the number of input features);
- $\mathbf{W}$x is an *n-inputs* $\times$ *n-neurons* matrix containing the weights for the inputs of the current time step;
- $\mathbf{W}$y is an *n-neurons* $\times$ *n-neurons* matrix containing the weights for the outputs of the previous time step;
- b is a vector of size *n-neurons* containing the *bias* term of each neuron;
- The matrices of weights $\mathbf{W}$x and $\mathbf{W}$y are often represented in a vertical concatenation in a single matrix $\mathbf{W}$ of size (*n-inputs* + *n-neurons*) $\times$ *n-neurons*;

- The notation [$\mathbf{X}(t)$ $\mathbf{Y}(t\text{-}1)$] represents a horizontal concatenation of the matrices $\mathbf{X}(t)$ and $\mathbf{Y}(t\text{-}1)$.

The output of the recurrent neuron at time step t is a function of all the inputs of the previous time step. This binding determines the introduction of a form of memory. This is what makes RNNs so successful in dealing with sequential data. The part of the Neural Network able to preserve the state over time is called *memory cell*. In this way a single recurrent neuron or a layer of recurrent neurons is called *basic cell*.
An important aspect of RNNs is that their architecture is very flexible, allowing for different combinations of input and output sequences.
The four most used types of RNN architectures[4] are:

- Many-to-one: it can have input sequences with arbitrary time steps and produces only one output. This architecture is mainly used to classify sequential data, such as for Sentiment Analysis;

- One-to-many: this is the opposite of the Many-to-one architecture. It has only one input which in this case is not a sequence and generates a sequence in output. It is used as a sequence generator, for example to generate a piece of music from an initial note;

- Many-to-many (synced): in this architecture each output is calculated on the basis of its corresponding input and all previous outputs. It is used for the prediction of time series such as the hourly energy consumption of a factory or the daily sales of each product of a company;

- Many-to-many (unsynced): this architecture is used when sequences need to be generated only after processing the entire input sequence. In this case, the length of the output sequence can be different from that of the input, giving us a certain flexibility. The most common application of this architecture is for automatic translation.

The most known types of RNN are the Bi-directional RNN (BRNN), the Gated Recurrent Unit (GRU) and the Long Short-Term Memory (LSTM).
BRNN is an evolution of the classic RNN that, instead of using only the previous inputs to make predictions on the current state, uses also the future data to improve the precision.
GRU is used to handle the problem of the short term memory of the RNN models.

[4]https://medium.com/@ODSC/understanding-the-mechanism-and-types-of-recurrent-neural-networks-6a93ee347e23

It uses hidden states and two ports, a reset port and an update port, to control how much and what information is retained.
A separate section will be dedicated to LSTMs for further study.

4.2.1 Long Short-Term Memory (LSTM)

LSTM is a type of RNN able to learn long-term dependencies.
They are used mainly for the classification and prediction of time series with intervals of unknown duration.
The architecture of an LSTM[5] is characterized by a chain structure of repetitive modules, called *cells*, where each of these modules consists of gates, as shown in Fig.4.6.

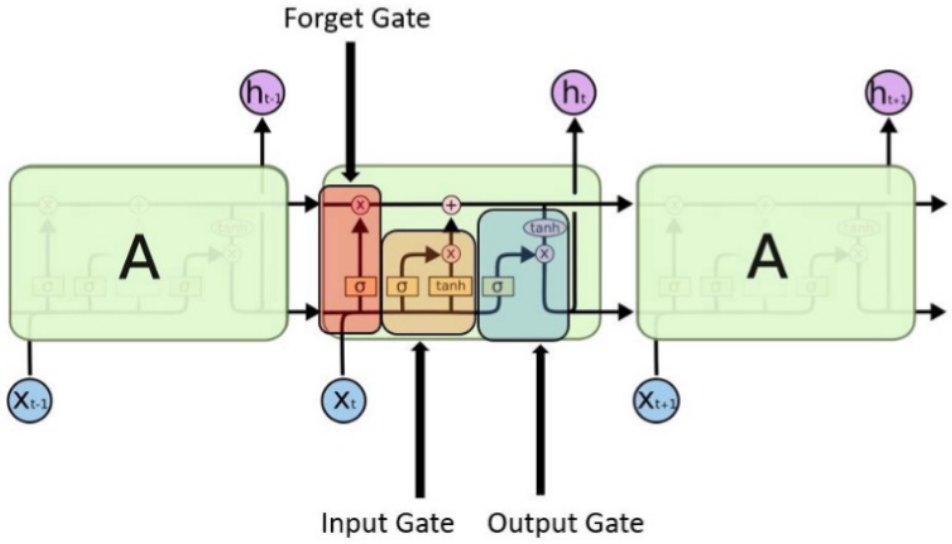


Figure 4.6: Gates in a LSTM *cell*.

There are three types of gates: the Input Gate, the Forget Gate and the Output Gate.
With the Input Gate it is decided which value of the input to keep and use in order to modify the state of the memory. With the Forget Gate it is decided which information should be discarded, and with the Output Gate the input and memory of the block are merged to produce the output.
The architecture shown in Fig.4.6 is composed by different elements shown in Fig.4.7: the yellow boxes indicate the learned layers of the Neural Network, the

[5]https://aditi-mittal.medium.com/understanding-rnn-and-lstm-f7cdf6dfc14e

pink circles represent point operations, the lines indicate the direction of scrolling of a vector, two or more lines that join indicate the operation of concatenation and finally, a line that bifurcates indicates that the content is copied in different positions.

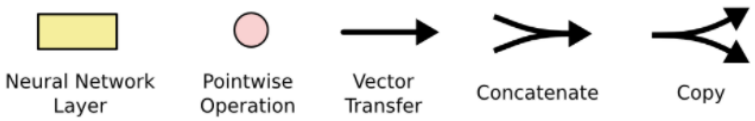
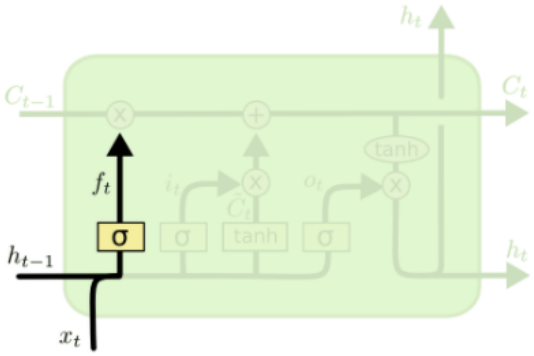


Figure 4.7: Elements in LSTM architecture.

The state of the LSTM[6] *cells* is divided, for performance reasons, into two state vectors h_t and c_t (Fig.4.8). h_t can be seen as the short-term state and c_t as the long-term state.
The first step is to decide which information to discard from the *cell* state. This operation is carried out by the Forget Gate that, through a *Sigmoidal* layer, analyzes the short term state h_{t-1} and the input x_t and produces a value between 0 and 1. If the value produced is 1 it means that the information has to be completely maintained, instead, if the value is 0 the information has to be completely discarded (Fig.4.8).

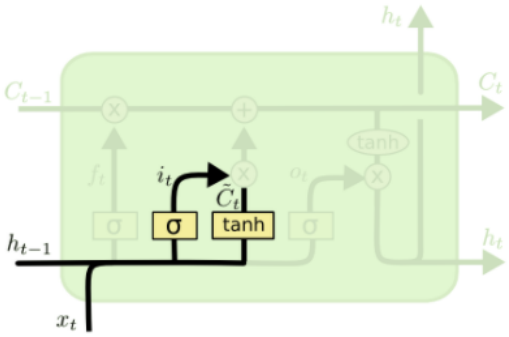


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Figure 4.8: First step.

The second step to be carried out is the analysis of which new information to add to the state of the *cell*. This step, as can be seen in Fig.4.9, is carried out by the Input Gate, and consists of two operations: in the first operation a *Sigmoidal* layer decides which values to update, while in the second operation a *tanh* layer creates the vector of new $\tilde{C}_t$ values that could be added to the state.

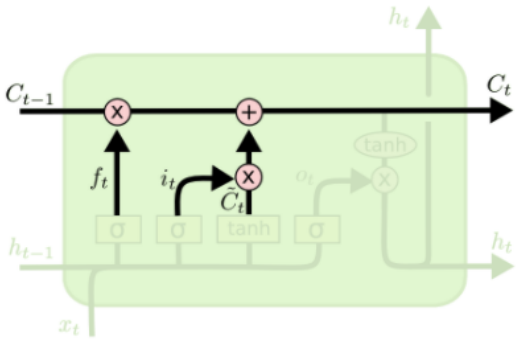
[6]http://colah.github.io/posts/2015-08-Understanding-LSTMs/



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Figure 4.9: Second step.

Now we need to update the old status of the *cell*, $\tilde{C}_{t-1}$, by combining the two values calculated in the previous step (Fig.4.10).
Two point operations are performed: the old state is multiplied by f_t, discarding the information we decided to forget in the first step, and the value calculated in the second step is added.



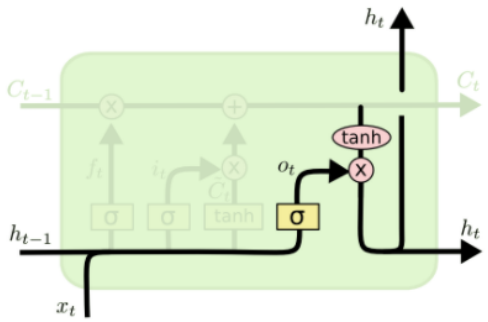
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Figure 4.10: Third step.

The final step involves deciding what we want the output to be. The output will be a filtered version of the *cell* state. This step is performed by the Output Gate and consists of two phases (Fig.4.11): a *Sigmoidal* layer is performed to establish which parts of the state will constitute the output of the *cell*, then a *tanh* layer is performed to ensure that the values of the state are between -1 and 1. Finally, the state of the *cell* is multiplied by the output of the *Sigmoidal* gate in order to output only the information that we have decided to return.

Variants of LSTM

There are different variants of LSTMs. Among these we distinguish the LSTMs with Peephole Connections and the Bi-directional LSTM (Bi-LSTM).



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Figure 4.11: Fourth step.

LSTMs with Peephole Connections[7] are LSTMs that are allowed to analyze not only the input x_t and the previous short term state h_{t-1} but also the long term state in order to give them a bit more context.
The architecture of a Peephole Connection LSTM has extra connections that make sure to add the previous long-term state c_{t-1} as input to the Forget Gate and Input Gate, and make sure to also add the current long-term state c_t as input to the Output Gate.

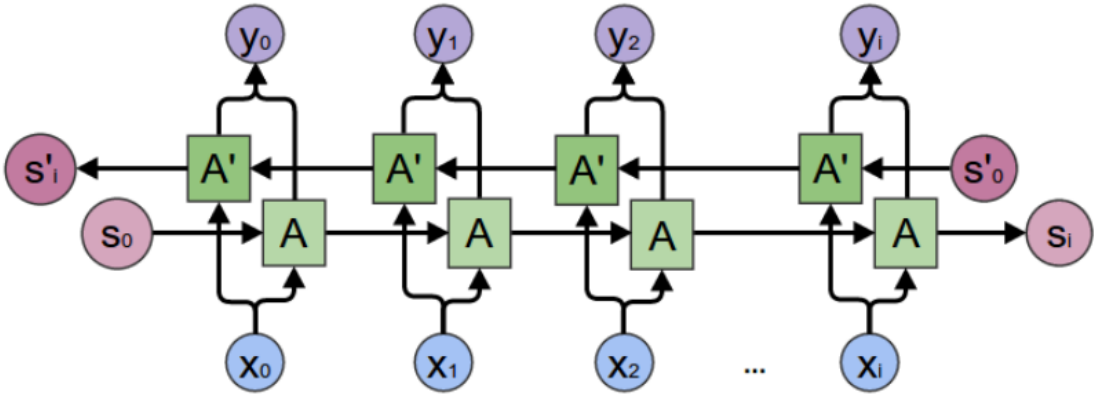


Figure 4.12: Bi-LSTM architecture.

The Bi-LSTM[8] are two independent LSTMs connected together, as it is possible to observe in Fig.4.12. A structure of this type allows to analyze the information both in one direction and in the other on the input sequence at each temporal step,

[7]https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch04.html
[8]https://medium.com/@raghavaggarwal0089/bi-lstm-bc3d68da8bd0

proving to obtain excellent results because they are able to better understand the context.
In this way the input is analyzed in two directions, one from the past to the future and the other from the future to the past, and is therefore able to store information from both the past and the future at any point in time.

4.3 SingleBranch Proposed Architecture

In this section the network architecture proposed to solve the Indoor HAR task is presented.
The network architecture will be called SingleBranch because an attempt of improvement will be proposed with another architecture, called DoubleBranch, characterized by the presence of two streams.
The SingleBranch, observable in Fig.4.13, has an architecture constituted by a backbone coming from a CNN for the extraction of the spatial features, a *Bi-LSTM* layer for the extraction of the temporal features and a classifier.
The input of this architecture is the set of 10 frames, of dimension 224x224x3, extracted from an input video.
The extraction of the spatial features from the input images is carried out by the backbone of a CNN. In Fig.4.13 a generic architecture of backbone is shown because different types of backbones, discussed in the Different Backbone section, will be tested. The performances of the different backbones will be evaluated to identify the most performing network architecture of which we will try to implement an improvement.
Once extracted the spatial features, they will be passed to the second part of the network architecture constituted by a *GlobalAveragePooling2D* that will average each channel of the input feature map to obtain a value, by the *Bi-LSTM* layer that will capture time dependencies, by a layer of *BatchNormalization* that accelerates the training reducing the training *epochs* and the error of generalization providing a certain regularization, by a first *Dense* layer with 512 neurons and *ReLU Activation Function*, by a *Dropout* layer that "switches off" in a random way a certain number of neurons and allows to obtain reduced networks able to better generalize, and finally, by a second *Dense* layer with a number of neurons equal to the number of the human activity classes and *Softmax Activation Function*.
The output of the SingleBranch architecture will then be the class of human activity detected by the SingleBranch on the set of frames passed as input.

TimeDistributed layer

When analyzing a time sequence of frames, it is necessary to adapt certain layers by introducing a *TimeDistributed* layer.

The *TimeDistributed* layer is nothing more than a wrapper that is applied, in this case, to the layers of the different backbones, such as the *Convolutional*, *Pooling* and *Dropout* layers, and to the *GlobalAveragePooling2D* so that they are able to operate on a time sequence. The *TimeDistributed* layer adds an extra dimension to the input that instead of being of size (*sample*, *width*, *length*, *channel*) will become (*sample*, *time*, *width*, *length*, *channel*).

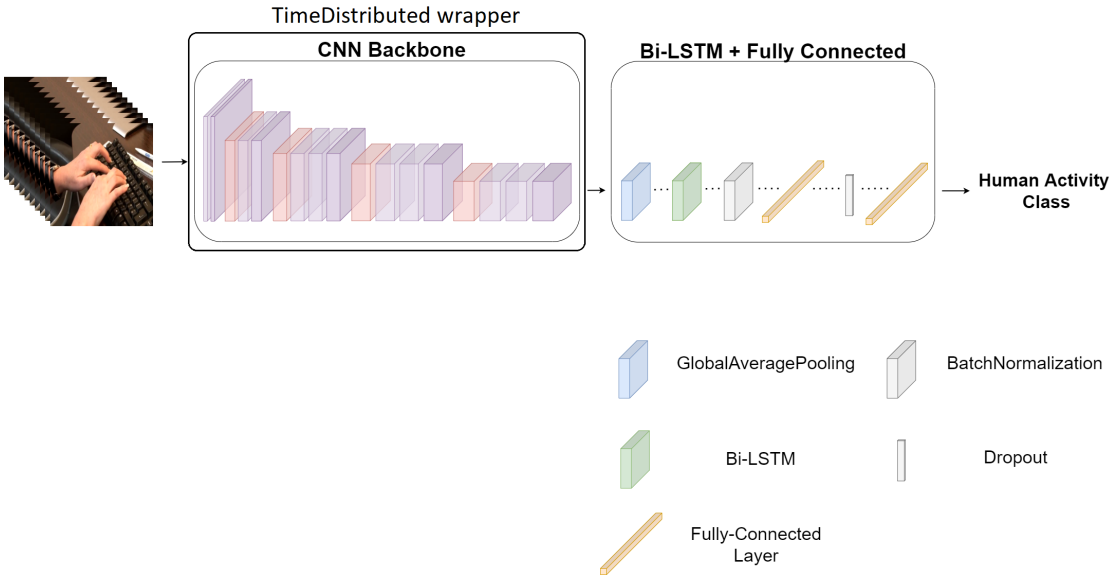


Figure 4.13: SingleBranch architecture.

4.4 Different Backbones

As previously mentioned, the backbone is the part of the Neural Network specialized in the extraction of the spatial features that will be used for the final classification. There are many types of backbones and they differ among them both for the logic of functioning and for the complexity, that is, the depth of the network and the number of *parameters*.
Some backbones that will be used as feature extractors in the previously introduced SingleBranch network architecture will be discussed below.

VGG

The VGG [14] is a CNN architecture for classification and identification tasks.
A VGG CNN is characterized by an architecture that makes use of very small convolutional filters (small receptive fields) of size 3x3. The use of these small

filters has demonstrated a significant improvement over previously implemented configurations.
There are two versions: VGG16 and VGG19.
VGG16 has 16 levels and just over 138 million *parameters*, while VGG19 has 19 levels and just over 143 million *parameters*. VGG19 is more complex but has similar performance to VGG16, which is why many developers are opting to remain with the simpler structure of VGG16.
The two versions of VGG are able to generalize the data, adapting and obtaining good results even on new datasets.

ResNet

The developers realized that Neural Networks with a high depth are more difficult to train. They proposed a Residual Neural Network (ResNet) [15] structure to facilitate the training of deep networks. They reformulate the architecture layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions.
ResNets 8 times deeper than VGG networks were tested on ImageNet.The ResNets achieved comparable performance while maintaining lower complexity.
ResNets are mainly used for Visual Recognition tasks.

Inception

While ResNet focuses on depth, Inception [16][17] focuses on extension. In particular, the developers of Inception were interested in the computational efficiency of training larger networks.
The architecture is characterized by 1×1 *Convolutions* to reduce dimensionality and "filter" the depth of the outputs. These *Convolutions* can extract spatial information and compress it into a smaller dimension. Classification tests have been performed on the ILSVRC 2012 dataset, yielding substantial performance gains over the state-of-the-art [16].
The Inception family includes the InceptionResNet which, by introducing residual connections, produced very good state-of-the-art performance in the 2015 ILSVRC challenge [17]. It would therefore appear that there are benefits to combining the Inception architecture with residual connections.

Xception

Xception [18], as its name suggests, takes the principle of Inception to the extreme. The architecture of Xception is inspired by that of Inception, where however, the Inception modules are replaced by "Depthwise Separable Convolution" which consists of a spatial *Convolution* (*Depthwise Convolution*) performed independently

for each channel, followed by a 1x1 *Convolution* (*Pointwise Convolution*) between the channels.
Correlations are then searched first in a two-dimensional space and then in a one-dimensional space.
The Xception architecture marginally outperforms InceptionV3 on the ImageNet dataset, while significantly outperforming InceptionV3 on a classification dataset with 350 million images and 17,000 classes.
The Xception architecture has the same number of *parameters* as InceptionV3 but performance gains are achieved due to more efficient use of model *parameters*.

MobileNet

MobileNets [19] are efficient network models for mobile and embedded vision applications. They are a simplified version of the Xception architecture, that uses *Depthwise Separable Convolutions* to build light weight Deep Neural Networks. They are mainly used for large-scale Object Detection, classification, Face Recognition and Geolocation applications.
There are different types of MobileNet: MobileNetV2, MobileNetV3Small and MobileNetV3Large.
The architecture of MobileNetV2 uses lightweight *Convolutions* to extract input features and uses an inverted residual structure in which the input and output of the residual block are thin bottleneck layers [20].
The remaining two models, MobileNetV3Small and MobileNetV3Large, are mainly used for Object Detection and Semantic Segmentation tasks on hardware with limited resources [21]. Their lighter architecture has led to very good results. MobileNetV3Large was 3.2% more accurate in classification on ImageNet with a latency reduction of about 20% compared to MobileNetV2.

DenseNet

Recent studies have shown that the CNNs can be more efficient and accurate, increasing their depth, if trained with an architecture that contains shorter connections between the input and output layers.
Dense Convolutional Network (DenseNet) [22] uses the same concepts of *Convolution*, *Pooling* and *ReLU Activation Functions* to work, but the innovation is in the introduction in the network architecture of *Dense* blocks.
Within DenseNet each layer is connected in a feed-forward manner to every other layer. Differently from the traditional CNNs that with N layers have N connections, each one between each layer and the next, in the DenseNet there are N(N+1)/2 direct connections.
The feature maps of all the previous layers are used as input for each layer, and the feature maps of this last one are used as input for all the following layers.

There are several advantages to using DenseNets: they improve feature propagation, encourage feature reuse, reduce the escape gradient problem, and also reduce the number of *parameters*.
There are many types of DenseNet which differ in depth. These include the DenseNet121 with 121 layers and just over 8 million *parameters*, the DenseNet169 with 169 layers and about 14 million *parameters* and finally the DenseNet201 with 201 layers and just over 20 million *parameters*.
The DenseNets have achieved great performance improvements over other state-of-the-art work, requiring fewer calculations.

EfficientNet

EfficientNet [23] is a type of Neural Network that improves model accuracy and computational requirements by efficiently scaling depth, width and resolution. EfficientNet does not require as many computational requirements as CNN, resulting in higher *Accuracy*.
To achieve EfficientNet, the scientific community realized that by carefully balancing the depth, width and resolution of the network, better performance could be achieved. Therefore, a scaling method was proposed that uniformly scales all three dimensions using a simple and very effective compound coefficient.
This type of Neural Network has a higher *Accuracy* and efficiency than traditional CNNs.
There are different types: from EfficientNetB0, with about 5 million *parameters*, to EfficientNetB7 with more than 66 million *parameters*. The last one has exceeded 80% *Accuracy* on ImageNet, being 8.4 times smaller and 6.1 times faster in inference than the best existing CNN.
Tests were also carried out on the CIFAR-100 dataset, obtaining an average *Accuracy* of over 91% and reaching 98.8% *Accuracy* for the Flower class.

NasNet

The NasNet [24] is a typology of Neural Network in which the *Convolution* layer (or "*cell*") has been improved. It is able to generalize very well on different datasets thanks to the introduction of a new technique of regularization called ScheduleDropPath.
Tests were performed on CIFAR-10 achieving an error rate of 2.4%.
Tests were also performed on ImageNet achieving *Accuracy* comparable to other state-of-the-art work. A small version of NasNet is able to exceed the *Accuracy* of other models of comparable size for mobile platforms by about 3%.

4.5 DoubleBranch Neural Network

In this section a new network, called DoubleBranch, will be presented as an attempt to improve the SingleBranch network architecture discussed previously.
The DoubleBranch, as the name suggests, is characterized by the presence of two flows. The architecture of the DoubleBranch is shown in Fig.4.14.
As for the SingleBranch, also the DoubleBranch has an architecture constituted by a first part of extraction of spatial features (backbone) coming from a CNN, then by a *Bi-LSTM* layer for the extraction of temporal features and finally by a classifier.
Also in this case, two generic backbone architectures are shown in Fig.4.14, as they will be replaced by the one that obtained the best performance in the test phase.
The part of the architecture of the DoubleBranch after the extraction of the spatial features is the same discussed already for the SingleBranch: it is then composed by a layer of *GlobalAveragePooling2D*, by the *Bi-LSTM* layer, by the layer of *BatchNormalization*, by a first *Dense* layer, by one of *Dropout* and finally by a last *Dense* layer with number of neurons equal to the number of the classes of human activity and *Softmax Activation Function*.

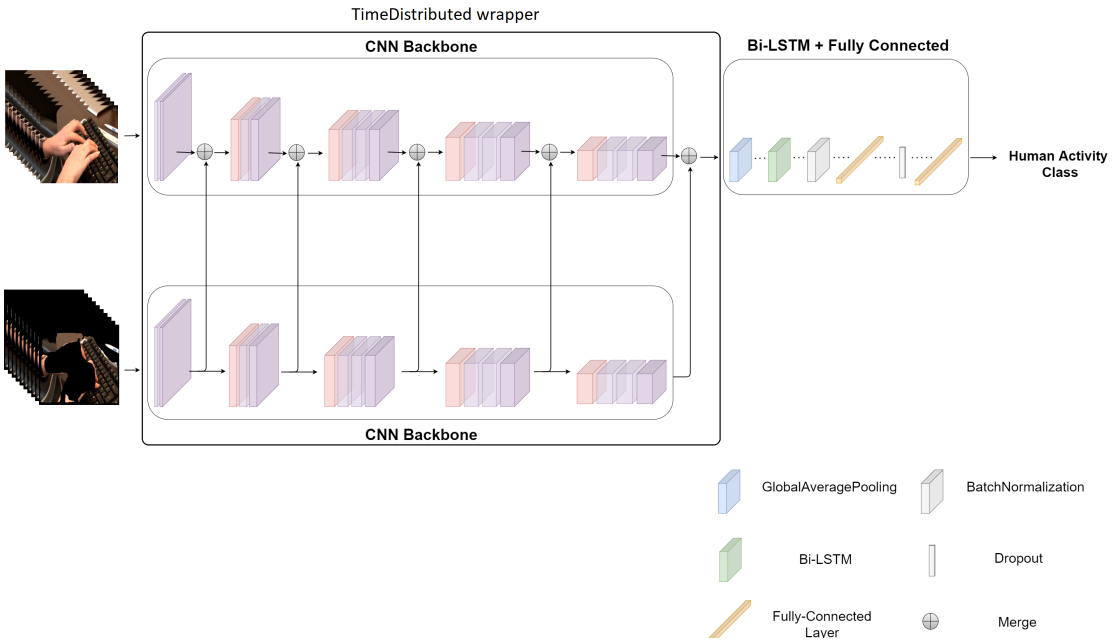


Figure 4.14: DoubleBranch architecture.

What distinguishes the DoubleBranch from the SingleBranch is the presence of two flows and the interconnections between the *Convolutional* layers of the two backbones.

As also proposed in the literature by the scientific community, a second branch has been introduced to improve the performance of the network. As input to the second branch, however, it was decided to use a set of 10 masks of size 224x224x3. One mask, Fig.4.15 on the right, is an RGB image obtained as a result of the Segmentation process carried out by a further Neural Network called MaskRCNN that will be briefly described in the next section.
In the mask the Segmentation process will highlight the interesting parts obscuring all the rest.
Having available both the RGB frames and the relative RGB masks with the interesting parts highlighted, *merge* is then carried out after each *Convolution*. In this way, the aim is to concentrate the network only on certain portions of the images that are considered important.
Also in this case, having to analyze temporal sequences, some layers have been wrapped by the *TimeDistributed* layer in order to be able to operate on frame sequences.

Figure 4.15: Two examples of RGB frames (on the left) and RGB masks (on the right).

4.5.1 MaskRCNN

MaskRCNN[9] [12][13], extension of the Faster R-CNN Neural Network, is a Neural Network used for instance Segmentation problems.
It allows to efficiently detect objects within an image and, at the same time, generate a high quality segmentation mask for each detected instance.
Compared to the Faster R-CNN there is the addition of a third branch that allows the prediction of the mask of the detected object instance within the image.

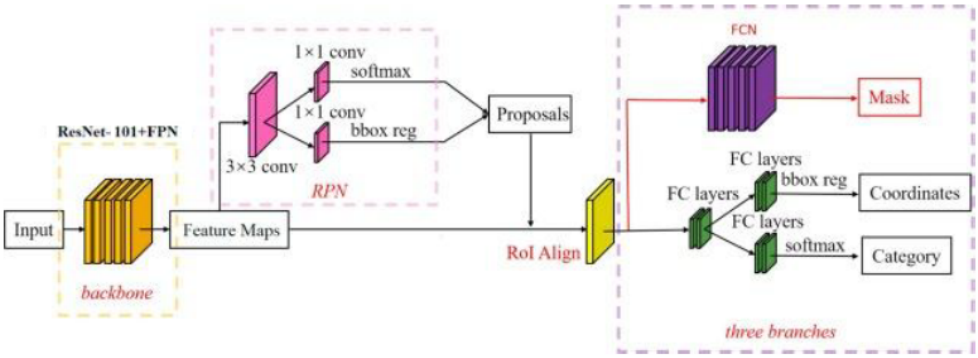


Figure 4.16: MaskRCNN architecture.

The architecture, observable in Fig.4.16, consists of a first extraction of the features performed through a ResNet101 and a Feature Pyramid Network (FPN) that exploits a pyramidal structure for the extraction of the features. After the extraction of the features, a Region Proposal Network (RPN) generates multiple regions of interest (RoI - anchor boxes), shown in Fig.4.17 and Fig.4.18, each characterized by the presence of a score that determines the presence of the object in the region and four coordinates representing the bounding box of the region.

For each RoI the MaskRCNN will generate a mask, observable in Fig.4.19, that will be expanded to adapt it to the dimensions of the corresponding bounding box. For each instance of object identified, MaskRCNN will produce an output like the one observed in Fig.4.20, characterized by the class, the bounding box and an overlapping mask.

4.5.2 DoubleBranch_DenseNet201

Due to the performance obtained from the training and testing phase of the different backbones of the SingleBranch shown in Chapter 6 of Results, the DenseNet201

[9]https://github.com/matterport/Mask_RCNN

Figure 4.17: Anchor sorting and filtering.

Figure 4.18: Bounding box refinement.

was considered to be the backbone with the best results.
It was decided, then, to apply the DenseNet201 as the backbone for the extraction of the spatial features of the two branches of the DoubleBranch.
In this way an architecture was created as an attempt to improve the SingleBranch. Fig.4.21 shows the architecture of the backbone of a DenseNet201. It consists of a first layer of *Convolution* 7 x 7 and *stride* 2, a second layer of *MaxPooling* and a series of *Dense Blocks* and *Transition Layers*. Each *Dense Block* is characterized by two *Convolutions*: a 1 x 1 *Convolution* and a 3 x 3 *Convolution*. The *Transition*

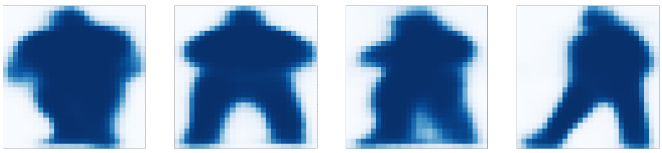


Figure 4.19: Mask generation.

Figure 4.20: Composing the different pieces into a final result.

Layers are characterized by a 1 x 1 *Convolution* and an *AveragePooling*. The architecture is defined by a repetition of the two *Convolutions* of the *Dense Blocks* for 6, 12, 48 and 32 times respectively.

To create the DoubleBranch_DenseNet201, shown in Fig.4.22, the *Convolutions*, the *Dense Blocks* and the *Transition Layers* characteristic of the DenseNet201 have been reproduced for the two branches of the DoubleBranch, with the addition of the *merge* of the results of each *Convolution*. The backbone is also wrapped by a *TimeDistributed* layer that makes possible the processing of temporal sequences. The second part of the network, on the other hand, is the same as that discussed for the SingleBranch. It consists of a *Bi-LSTM* for the extraction of temporal features and the classifier.

4.5.3 DoubleBranch_VGG16

As shown in Chapter 6 of Results, DoubleBranch_DenseNet201 did not produce any results during the learning phase. This is probably due to the complex and deep architecture of the model.
It was decided, therefore, to model a new DoubleBranch with the use of the

Layers	Output Size	DenseNet-201
Convolution	112 × 112	7 × 7 conv, stride 2
Pooling	56 × 56	3 × 3 max pool, stride 2
Dense Block (1)	56 × 56	[1 × 1 conv ; 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv
	28 × 28	2 × 2 average pool, stride 2
Dense Block (2)	28 × 28	[1 × 1 conv ; 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv
	14 × 14	2 × 2 average pool, stride 2
Dense Block (3)	14 × 14	[1 × 1 conv ; 3 × 3 conv] × 48
Transition Layer (3)	14 × 14	1 × 1 conv
	7 × 7	2 × 2 average pool, stride 2
Dense Block (4)	7 × 7	[1 × 1 conv ; 3 × 3 conv] × 32

Figure 4.21: SingleBranch DenseNet201 backbone architecture.

backbone VGG16 which has a structure very easy to manipulate and because it is very used in literature for tasks of classification.
The architecture of the DoubleBranch_VGG16 is shown in Fig.4.23.
The backbones of the two branches have a structure of this type: 3 x 3 *Convolution*, *merge* of results, *MaxPooling*, two 3 x 3 *Convolutions*, *merge* of results, *MaxPooling*, three 3 x 3 *Convolutions*, *merge* of results, *MaxPooling*, a *Dropout* layer, three 3 x 3 *Convolutions*, *merge* of results, *Maxpooling*, another *Dropout* layer, three 3 x 3 *Convolutions*, *merge* of results, *MaxPooling* of the *merge* result and a last *Dropout* layer. To the classic backbone architecture of VGG16, then, *merge* of *Convolutions* and *Dropout* layers were added to improve learning and generalization capability.

4.6 Experimental Protocols

In this section we will discuss the dataset used for training the previously introduced models and the preprocessing related to the sampling process for frame extraction and adaptation to the network input, the various configurations adopted for training, giving a brief description of the *optimizers* tested, and the performance metrics that will be used to evaluate the performance of the models.

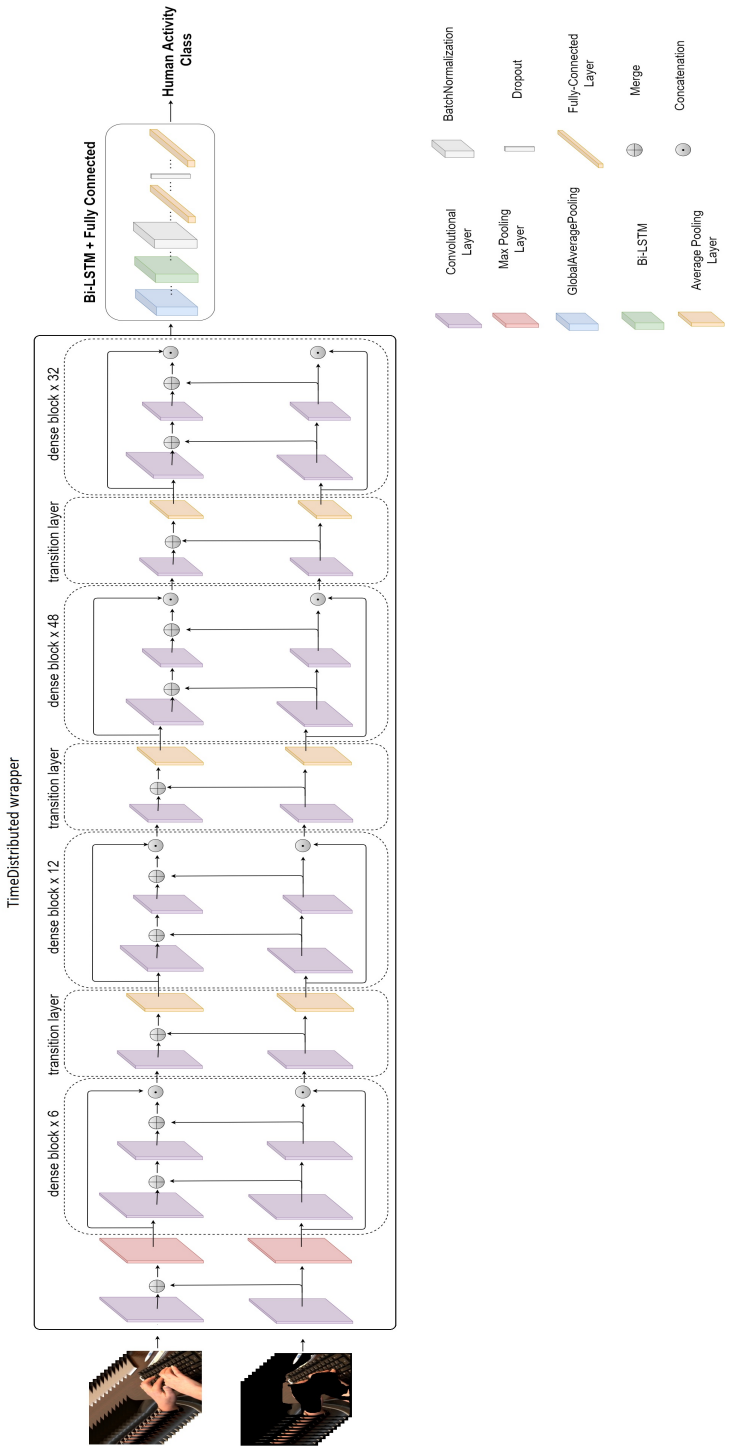


Figure 4.22: DoubleBranch_DenseNet201 architecture.

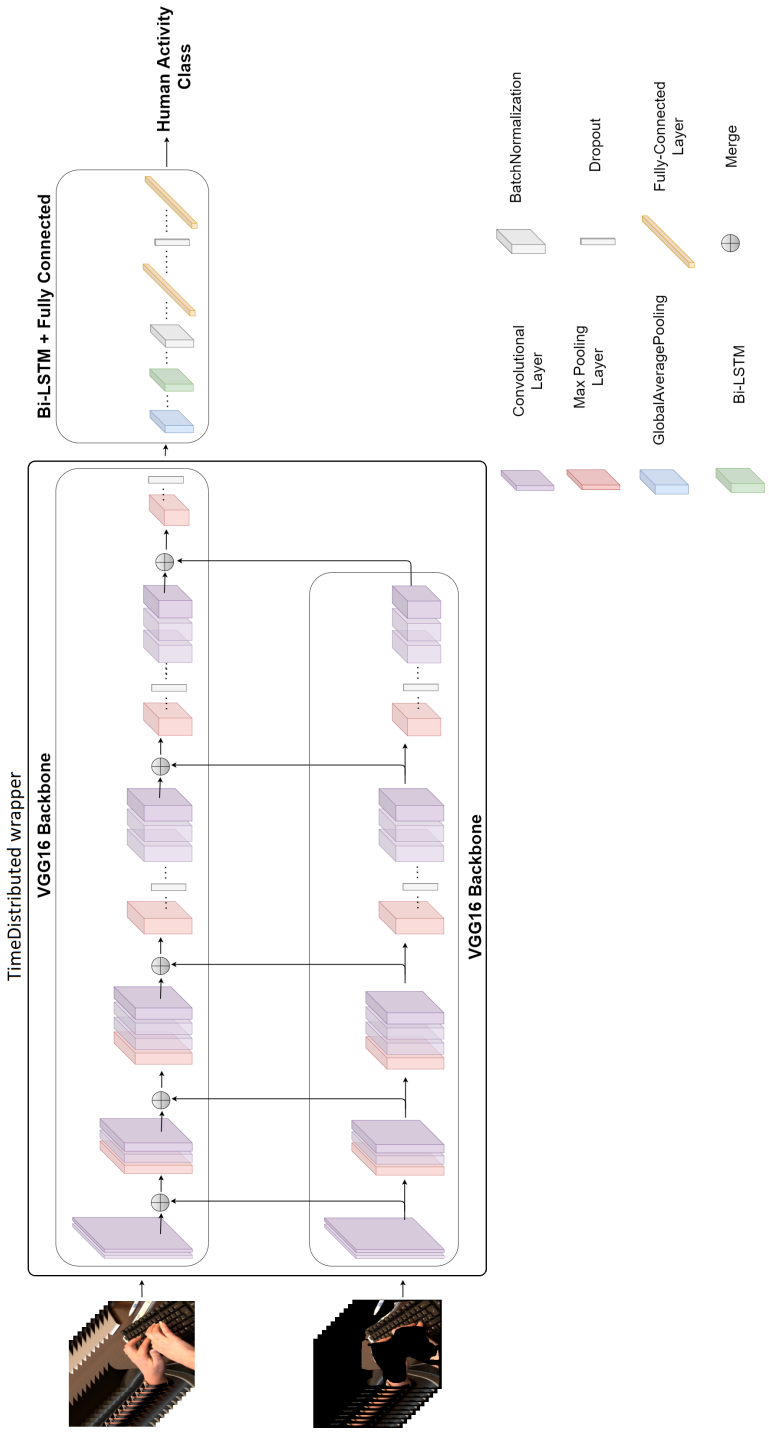


Figure 4.23: DoubleBranch_VGG16 architecture.

4.6.1 Dataset

The dataset that was used for training the network models is a subset of Kinetics.

Kinetics

The Kinetics dataset can be considered as the successor to two well-known reference datasets for HAR: HMDB-51 and UCF101 [25].
The creation of Kinetics is due to the fact that the use of HMDB-51 and UCF101 has decreased considerably because of their small size and variations.
Kinetics is a collection of up to 650,000 YouTube annotated video clips, each of 10s, that cover 400/600/700 human action classes, depending on the dataset version.
The clips come from YouTube and are mainly amateur videos, so they have a variable resolution and frame rate. For this reason, there may be considerable camera movements, lighting variations, shadows, etc. Being amateur videos, there is a great variety of performers and a lot of difference in the way the same action can be performed.
The dataset contains different actions like *Singular Person Actions* (e.g. "*robot dancing*", "*stretching leg*"), *Person-Person Actions* (e.g. "*shaking hands*", "*tickling*") and *Person-Object Actions* (e.g. "*riding a bike*"). There are also actions with same verb but different objects (e.g. "*playing violin*", "*playing trumpet*") and actions with same object but different verbs (e.g. "*dribbling basketball*", "*dunking basketball*").
As can be observed in Fig.4.24, in some cases it is not easy to recognize the human activity, as for the class "*headbanging*", or to distinguish the class, as for "*dribbling basketball*" and "*dunking basketball*".

For the training of the models in this work, a manually created subset of Kinetics was used, containing 11584 video clips of 20 interesting activity classes (*brushing_teeth*, *cleaning_floor*, *cleaning_toilet*, *cleaning_windows*, *crawling_baby*, *dining*, *doing_nails*, *drinking*, *hugging*, *ironing*, *kissing*, *making_beds*, *opening_bottles*, *playing_cards*, *reading_books*, *setting_the_table*, *using_the_computer*, *washing_dishes*, *washing_hair* and *washing_hands*) divided into 92% and 8%. The 92% was further divided into 80% (8556 video clips) for the training set and 20% (2140 video clips) for the test set, while the remaining 30% (888 video clips) was dedicated to the validation set.
When dividing the dataset into training, validation and test set, the *stratified mode* was used to make sure that the number of video clips of each class was balanced between the different sets.
The 20 classes extracted from the original dataset were chosen to focus on Indoor HAR.

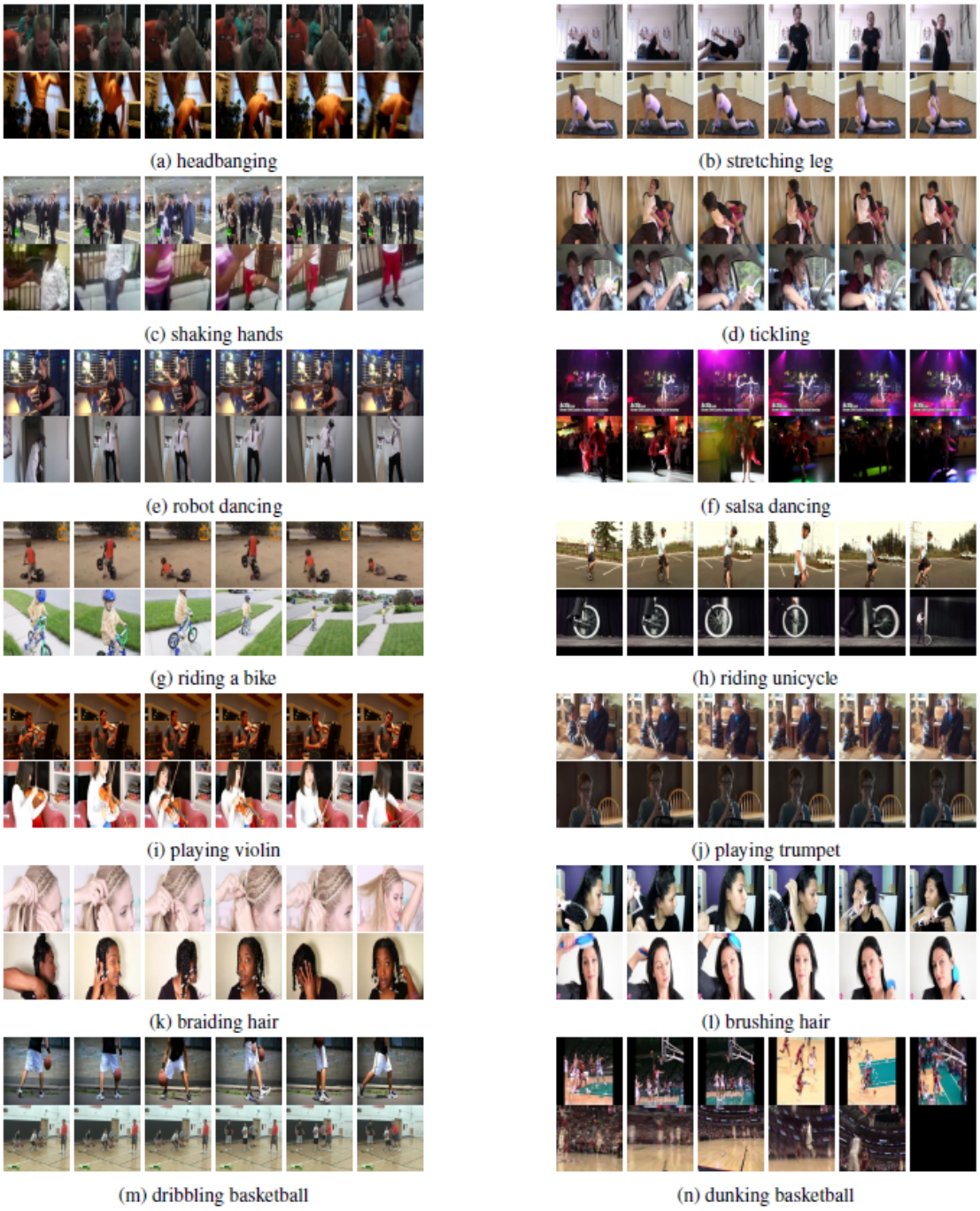


Figure 4.24: Examples of Kinetics dataset classes.
Image taken from [25]

Preprocessing

Once the dataset was acquired, preprocessing had to be carried out in order to adapt the input of the models.
A sampling process had to be performed on the videos for frame extraction. For each video it was experimentally decided to extract a number of 10 frames and to create a folder, containing the 10 frames, for each video.
The sampling process involves the extraction of frames with a frequency determined by the following calculation:

$$SamplingRate = \frac{\text{video length in seconds}}{\text{no. of desired frames}}$$

Once the frames have been extracted, they were resized to 224x224 pixels.
Finally, the average of the values is subtracted from the data set and the normalization is performed.

4.6.2 Training Settings

The training of network models on Kinetics was performed using different configurations.
First of all, the training was performed on an environment with "unlimited" resources: Google Colab. Google Colab is a tool provided by Google that allows Python code to be written directly from the Browser and allows robust and efficient network models to be built and implemented, providing significant computing power.

Fine Tuning was performed to realize *transfer learning* using the pre-trained ImageNet weights.
As can be seen in Chapter 6 of Results, different configurations were tested to find the best configuration to obtain the best model.
The training set was divided into subgroups called *batches* of varying size.
Training was carried out with a variable number of *epochs*: from 30, 60 to 80 *epochs*.
The *loss function* calculated on the training and validation set, used to evaluate the performance of the model, is the *Categorical Crossentropy*. It is mainly used for multi-class classification problems.
Finally, different types of *optimizers* were tested but not all of them produced results:

- *SGD*: acronym for *Stochastic Gradient Descent*, is an *optimizer* used to minimize the *loss function*. It calculates the gradient and updates it after each observation;

- *Adam*: is an *optimizer* that calculates adaptive *learning rates* for each *parameter* by storing an exponential decay average of past square gradients and maintaining an exponentially decreasing average of past gradients;
- *Adagrad*: a gradient-based algorithm that adapts the *learning rate* to the *parameters*, performing larger updates for *parameters* associated with infrequent features and smaller updates for those associated with more frequent features. It is therefore no longer necessary to set the *learning rate* manually, but the default value of 0.01 is often used;
- *Adamax*: is an *optimizer* that is a variant of *Adam* that, in the calculation formula, it uses a different norm;
- *Adadelta*: is an extension of *Adagrad* that, instead of accumulating past squared gradients, limits the window of accumulated past gradients to a fixed size;
- *Nadam*: is an *optimizer* that incorporates *NAG* (*Nesterov Accelerated Gradient*) into *Adam*. It is obtained by modifying the *momentum* term.

4.6.3 Performance Metrics

All network models discussed above will be evaluated using the metrics of *Accuracy*, *Precision*, *Recall* and *F1-score*. To assess the performance of the whole approach, we computed the classification *Accuracy* (Eq.4.1) as a global metric.

Accuracy (*Acc*), as the name suggests, indicates the accuracy of the model and describes how the model performs with respect to all classes. The values of *Acc*, and all other metrics, range from 0 (worst case) to 1 (best case). It is calculated as the average *Acc* per class, which is the ratio of the number of correct predictions to the total number of predictions:

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

where *TP*, *TN*, *FP* and *FN* are *TruePositive*, *TrueNegative*, *FalsePositive* and *FalseNegative* respectively.
Acc is very useful when the dataset is balanced between the various classes, but when we have an unbalanced dataset, we must be careful and avoid giving a wrong interpretation to the values as the calculated *Acc* may not be valid for all classes.

Precision (*Prec*) measures the *Acc* of the model in classifying a sample as positive and is calculated for each class as the ratio of the number of positive samples that were classified correctly to the total number of samples classified as positive both

correctly and incorrectly. Said another way, "*for all instances classified as positive, what percentage were correct?*"[10].

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

Recall (*Rec*) indicates the ability of a classifier to detect all positive instances. For each class it is defined as the ratio between the number of positive samples correctly classified as positive and the total number of positive samples. Said another way, "*for all the instances that were actually positive, what percentage were correctly classified?*".

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

Finally, the *F1-score* (*F1*) is a fusion of *Prec* and *Rec*. It is the weighted harmonic mean of the previous two and is used to compare classification models and not overall *Acc*.

$$F1 = \frac{2 \cdot Rec \cdot Prec}{Rec + Prec}$$

[10]https://www.lorenzogovoni.com/matrice-di-confusione/

Chapter 5

Deployment On Limited Resources Hardware

In this Chapter we will describe the cameras used to obtain the video stream to be analyzed, the microprocessor used (NVIDIA Jetson Nano) and everything related to the deployment of network models on hardware.
We will introduce TensorRT which is a platform for the optimization of inference on hardware and finally we will present the management mechanism of the video stream of the camera for the recognition of human activities.

5.1 Cameras

The cameras used to acquire the video stream on which the Indoor HAR task was performed are presented in Fig.5.1.
Two types of cameras provided by Dahua were used: an IP Bullet Camera (Fig.5.1 on the left) and an IP Eyeball Camera (Fig.5.1 on the right).
For both cameras, the setup and configuration were very simple as it was only necessary to connect the two cameras to the Internet and type in the URL bar of the Browser the IP address 192.168.1.108 (default) which allows access to the configuration interface (Fig.5.2).
It should be noted, however, that by default the address 192.168.1.108 is assigned to each camera, therefore, if tests are to be carried out with several cameras connected at the same time it is necessary to change the address of the cameras from the configuration interface in order to avoid conflicts.
In our case, the default configurations of the various camera parameters have been maintained except for a few parameters listed below:

- Codec (Encode Mode): H.264;

- Resolution: 1280 * 720 pixels;
- Fps: 25;
- Bit Rate: 2048.

In order for the NVIDIA Jetson Nano to receive the video stream from the camera, the RTSP protocol (Real Time Streaming Protocol) was used, which is a network protocol that controls multimedia streaming servers. With the use of this protocol it is possible to establish and manage streaming sessions between clients and servers. The standard communication port used is port 554.

Figure 5.1: IP Bullet Camera (left) and IP Eyeball Camera (right).

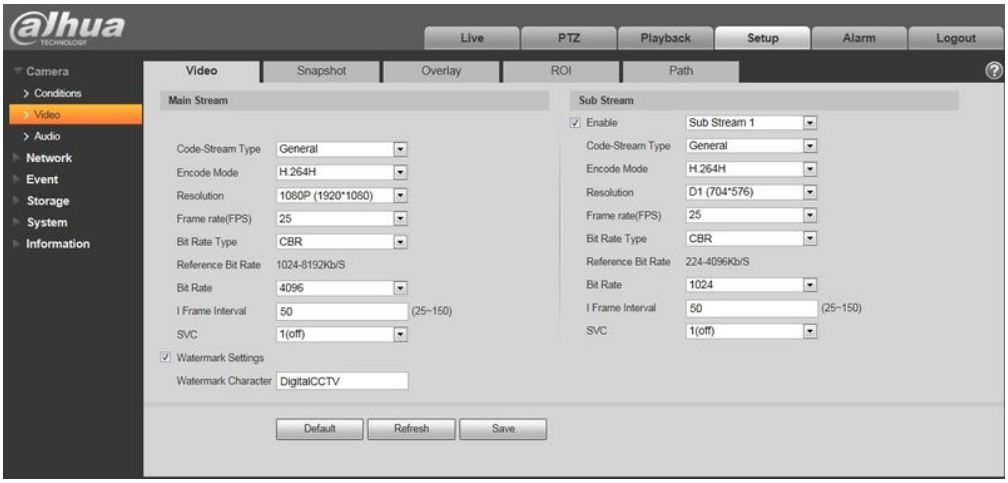


Figure 5.2: Camera configuration interface.

5.2 NVIDIA Jetson Nano

The NVIDIA Jetson Nano[1] (Fig.5.3) is a small, powerful computer designed by NVIDIA dedicated to Edge Computing tasks on mobile or embedded devices.
It is a CUDA-X processor with 472 GFLOPs of power and 4GB of RAM capable of running on just 5 Watts of power.
It is currently used by more than 200,000 developers for the development of drones, security cameras and many other devices that can run without the use of an Internet connection.

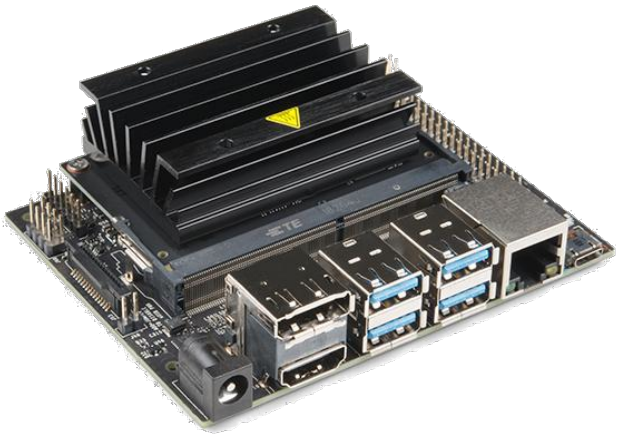


Figure 5.3: NVIDIA Jetson Nano.

The NVIDIA Jetson Nano allows image Classification, Segmentation and Object Detection tasks to be performed by running multiple Neural Networks in parallel. Jetson Nano is supported by the NVIDIA JetPack which includes the latest Jetson Linux Driver Package with Linux operating system and CUDA-X accelerated APIs and libraries for Deep Learning, Computer Vision and Accelerated Computing and Multimedia.
It also includes documentation and tools for developers to create products with increased performance and scalability. Tools include NVIDIA DeepStream and

[1]https://www.nvidia.com/it-it/autonomous-machines/embedded-systems/jetson-nano/product-development/

Transfer Learning Toolkit for video analysis, NVIDIA Clara for imaging and patient monitoring tasks, and NVIDIA Isaac for robotics.
The NVIDIA Jetson Nano can be used in two modes:

- Desktop Mode: in this mode, the Jetson Nano is connected directly to a monitor, keyboard and mouse. No additional PC support is required;

- Headless Mode: in this mode, no monitor, keyboard or mouse are required, but it is necessary to connect the Jetson Nano to a PC and establish a connection via SSH protocol.

The mode of use of the Jetson Nano for this thesis work was Desktop Mode.
Below is the data sheet with all the information about the NVIDIA Jetson Nano (Fig.5.4).

Specifiche tecniche kit sviluppatori	
GPU	NVIDIA Maxwell™ 128 core
CPU	ARM A57 quad-core a 1,43 GHz
Memoria	LPDDR4 4 GB 64-bit 25.6GB/s
Spazio di archiviazione	microSD (scheda non inclusa)
Encoder video	4Kp30 | 4x 1080p30 | 9x 720p30 (H.264/H.265)
Decoder video	4Kp60 | 2x 4Kp30 | 8x 1080p30 | 18x 720p30 (H.264/H.265)
Connettività	Gigabit Ethernet, M.2 Key E
Fotocamera	2 connettori MIPI CSI-2
Display	HDMI e DP
CONNETTORE USB	4 USB 3.0, USB 2.0 Micro-B
Altri	Testina 40-pin (GPIO, I2C, I2S, SPI, UART) Testina 12 pin (alimentazione e segnali correlati, UART) Testina ventola 4-pin
Meccanica	100 mm x 80 mm x 29 mm

Figure 5.4: NVIDIA Jetson Nano data sheet.

Setting Up and Configuration

The NVIDIA Jetson Nano setup[2] was performed following the guide provided directly by NVIDIA. The setup procedure involves writing the Jetson Nano Developer Kit SD Card Image to our microSD card. NVIDIA Jetson Nano uses the microSD card as a boot device and for main storage.
Once the setup was done, a virtual environment was created on which everything needed to set up the working environment was installed. Among others, *Python3*, *Tensorflow2.4.0* and other utilities such as *pandas*, *natsort* and *numpy* were installed.

5.3 TensorRT and Model Optimization

NVIDIA TensorRT[3][4] is a Software Development Kit (SDK) for high - performance inference of Deep Learning models. It contains an inference optimizer and runtime that provides high throughput and low latency for Deep Learning pattern inference applications.
TensorRT is built on CUDA, which is an NVIDIA parallel programming model, and enables optimized model inference through the use of CUDA-X development tools, libraries and technologies for Artificial Intelligence, high-performance computing and autonomous machines.
To optimize the network model, TensorRT performs a process called build phase (Fig.5.5). In the build phase, TensorRT takes the definition of the network model, performs network and platform optimization and generates the inference engine.
In the build phase, dead computations are eliminated, operations are reordered and combined to execute more efficiently on the GPU, floating point computations can be reduced to 32, 16 and 8 bits through the quantization process, multiple implementations of operators can be run to find those that produce the fastest network implementation, where possible the *Convolution*, *bias* and *ReLU* layers are merged to form a single layer and horizontal merges or aggregations of layers are performed to improve model performance.

As shown in Fig.5.5, several operations are performed during the network model optimization process. Mixed precision (1) can be used which maximizes throughput by quantizing models at INT8 while preserving accuracy; layer and tensor merging (2) can be performed by merging nodes into a kernel and optimizing memory usage; the best data layers and algorithms can be selected based on the target

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

[3]https://developer.nvidia.com/tensorrt

[4]https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html

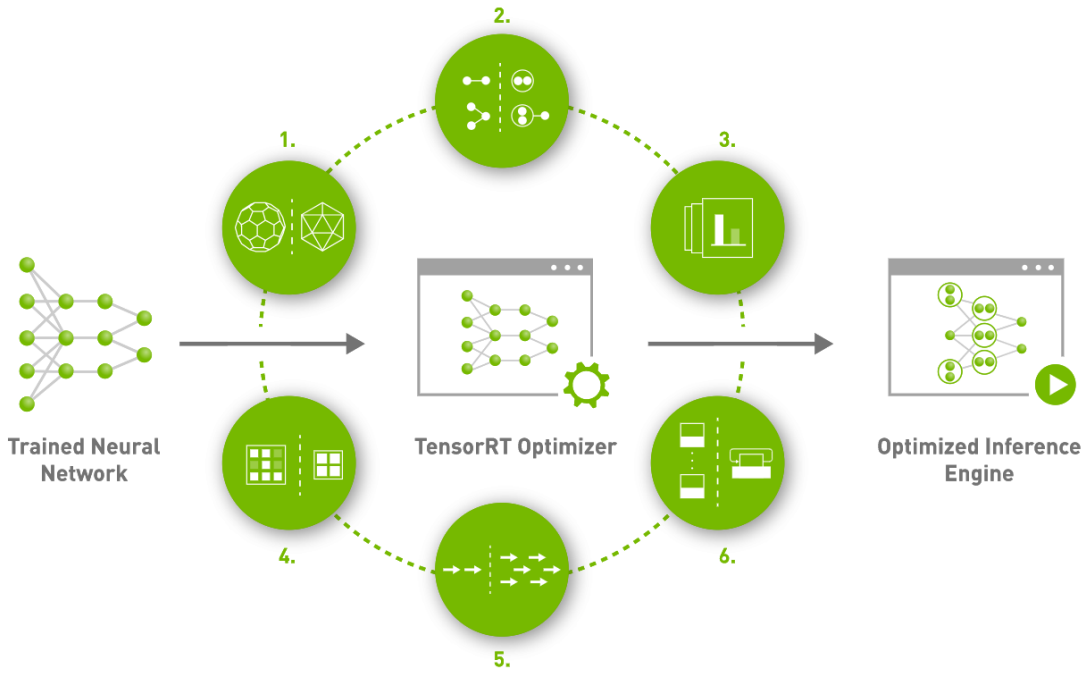


Figure 5.5: TensorRT build phase.

GPU platform (3); a dynamic tensor memory (4) is used which minimizes memory occupancy and promotes memory reuse for tensors efficiently; promotes multi-stream execution (5) using a scalable design to process multiple streams in parallel and finally performs temporal fusion (6) by optimizing RNNs in time steps with dynamically generated kernels.

There is a version of TensorRT integrated directly within Tensorflow: TF-TRT[5]. For the conversion of the network models built in this thesis work, TF-TRT was used as a model optimizer for performing inference within the NVIDIA Jetson Nano.

Figure 5.6: TF-TRT workflow for SavedModel format models.

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

TF-TRT optimizes and executes the compatible subgraphs, allowing Tensorflow to execute the remaining part of the graph.
The model optimization procedure, which can be observed in Fig.5.6 and Fig.5.7, requires that a model saved in SavedModel format (or in regular checkpoints) from a trained Tensorflow model is provided as input. At this point, TF-TRT returns the SavedModel optimized by TensorRT (or the frozen inference graph). TF-TRT will replace each subgraph supported by it with a node optimized by TensorRT (called TRTEngineOp) and produce a new graph on which inference can be performed.

Figure 5.7: TF-TRT workflow for Checkpoints models.

5.4 Single-camera Handling

The management of the camera stream for the execution of inference within the Jetson Nano is summarized in Fig.5.8.
In this mechanism there are six actors: the *Main Thread* that manages the whole program, the *TrtThread* that is the thread that manages the mechanism of creation of the input to give to the model and to start the inference process, the *Model* that is the model optimized in TF-TRT and loaded inside the Jetson Nano, *Camera Handle* that is a class that manages the camera, *CapturingThread* that continuously acquires the frames and *Camera* that refers to one of the two types of IP Camera described previously and from which the video stream will be acquired.
The *Main Thread* first creates a *Camera Handler* object of the class that manages the camera.
The *Camera Handler* establishes a connection, through the RTSP protocol, with the *Camera* and starts the secondary *Capturing Thread*.
The *Capturing Thread* is responsible for continuously extracting frames from the camera stream and inserting them into an attribute of the *Camera Handler* object.
The *Main Thread*, at this point, loads the TF-TRT optimized *Model* and performs a first simple inference on 10 images to load all the libraries and utilities functions needed for inference. This loading process takes a few seconds so it was decided to perform a first single inference to ensure that the libraries and functions were already loaded and therefore there would be no delays in the inference performed in real-time.
Once everything is loaded, the *Main Thread* starts the *TrtThread*.
The *TrtThread* executes in loop the extraction operation, from the attribute of the

Camera Handler object, of 10 frames to create the input tensor of the model and asks the *Model* to execute the inference on the 10 frames. Every time it obtains the results of the inference relative to the predicted class, to the confidence score, to the FPS and to the prediction time, it sends them to the *Main Thread* that will take care of the visualization process.
In the moment in which it is decided to terminate the application, the *Main Thread* will ask the *TrtThread* to stop and will forward to the *Camera Handler* the request to stop the *Capturing Thread* and the closing of the communication with the *Camera*.

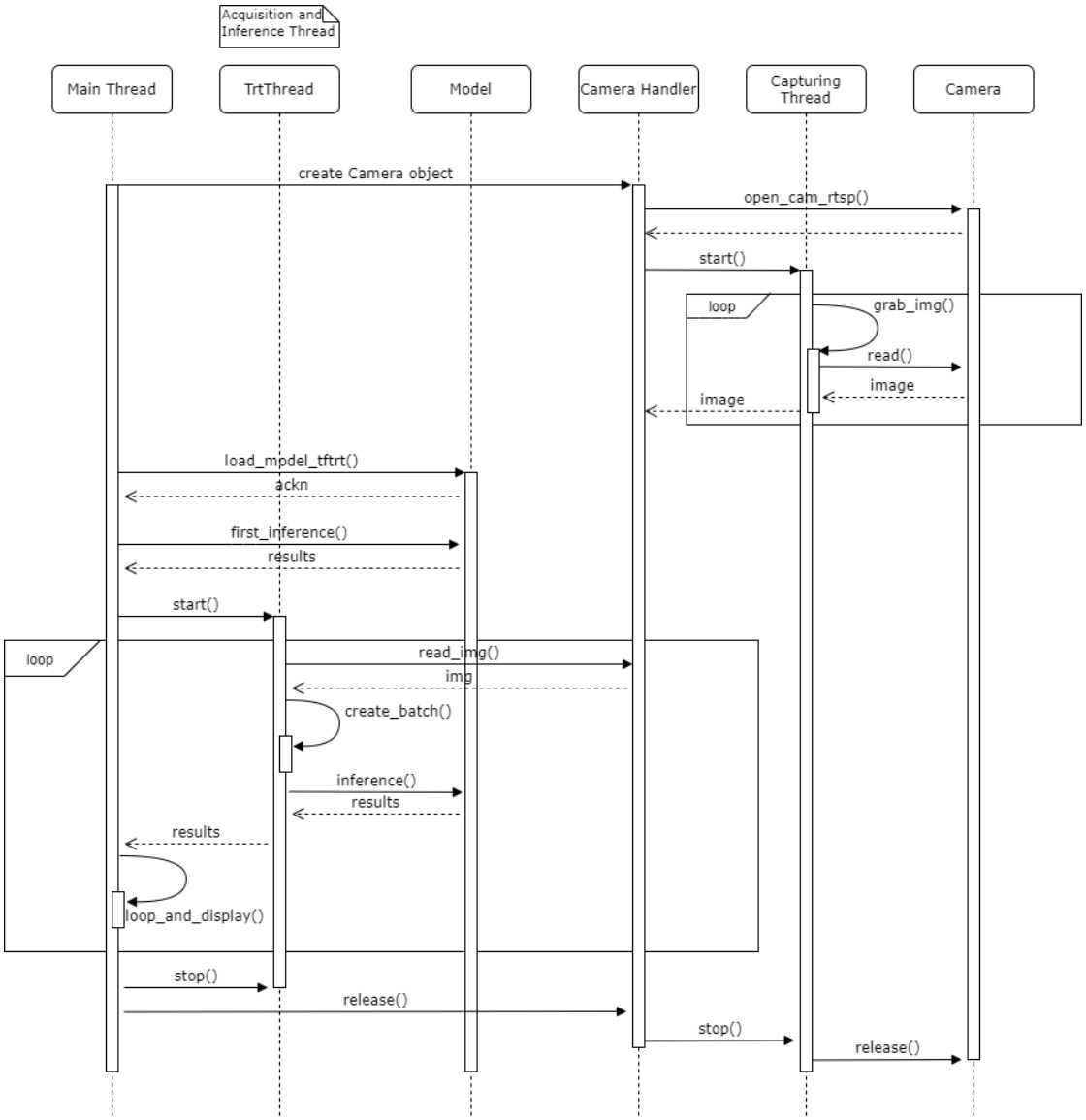


Figure 5.8: Single-camera Handling Workflow.

Chapter 6

Results

In this Chapter, we will present the test results obtained on an environment with "unlimited" resources (Google Colab) and on an environment with limited resources (NVIDIA Jetson Nano).
As regards the environment with "unlimited" resources, the performances achieved in the test phase by the network models discussed in the previous chapters will be presented. As already mentioned, the performance obtained on Google Colab was evaluated by analyzing the performance metrics of *Acc*, *Rec*, *Prec* and *F1*.
An overview of the results obtained from the various backbones on the Kinetics dataset will be presented.
A focus will be given to the performance of the best SingleBranch model and the models with which it was possible to perform inference within Jetson Nano.
The results of the training of the DoubleBranch model on dataset will also be reported.
As concerning the NVIDIA Jetson Nano, the results obtained from the optimization process of the network models and the performances obtained from the real-time inference, on the successfully optimized models, performed on the stream of a video camera will be presented.
Finally, some examples of correct and incorrect prediction of the activity class will be illustrated.

6.1 Unlimited Resource Environment

This section will show the performance achieved by the tested backbones as spatial feature extractors for SingleBranch models and the results obtained by DoubleBranch models.

6.1.1 Backbone Performance Comparison

Table 6.1 shows the performance obtained in the test phase of the SingleBranch with the different backbones. The table shows for each backbone the results of the performance metrics of *Acc*, *weighted avg Prec*, *weighted avg Rec*, *weighted avg F1*, *macro avg Prec*, *macro avg Rec* and *macro avg F1* achieved in the training phase. The reported averages include macro average (averaging the unweighted mean per label) and weighted average (averaging the support-weighted mean per label).
Given the different complexity of the backbones and the variable allocation of hardware resources by Google Colab, it was not possible in some cases to train models with a *batch size* of 16.

Table 6.1: Backbone performance comparison.

Network	Acc	weighted avg Prec	weighted avg Rec	weighted avg F1	macro avg Prec	macro avg Rec	macro avg F1
VGG16	75%	76%	75%	75%	74%	72%	73%
MobileNetV2	77%	77%	77%	77%	76%	74%	75%
MobileNetV3Small	69%	69%	69%	69%	68%	66%	66%
MobileNetV3Large	76%	77%	76%	76%	74%	74%	74%
EfficientNetB2	82%	82%	82%	82%	80%	79%	79%
EfficientNetB1	82%	82%	82%	82%	80%	79%	79%
EfficientNetB0	80%	80%	80%	80%	79%	77%	78%
InceptionResNetV2	82%	82%	82%	82%	81%	79%	80%
NasNetMobile	79%	80%	79%	79%	78%	77%	77%
ResNet152	80%	81%	80%	80%	80%	78%	78%
ResNet152V2	78%	79%	78%	78%	77%	76%	76%
Xception	82%	82%	82%	82%	81%	79%	80%
DenseNet201	82%	83%	82%	82%	81%	80%	80%
DenseNet121	81%	81%	81%	81%	80%	78%	79%

The performances shown in the table were obtained by performing *Fine Tuning*, and therefore taking networks already pre-trained on *ImageNet*, and training the models at 80 *epochs* with the *Adagrad optimizer* and *learning rate* of 0.001 and

ReLU Activation Function.
From Table 6.1, it can be seen that the backbone with the best results is the DenseNet201. With its 18 million parameters it achieved an *Acc* of 82%, a *weighted avg Prec* of 83%, a *weighted avg Rec* of 82% and an *weighted avg F1* of 82%.

Table 6.2, Table 6.3, Table 6.4, Table 6.5 and Table 6.6 show the performances obtained with the DenseNet201, the VGG16, the MobileNetV2, the MobileNetV3Small and the MobileNetV3Large respectively. Only the results obtained with these backbones have been highlighted because the DenseNet201 is the best performing model obtained and together with the VGG16 will be used also for the modeling of the DoubleBranch, while the three MobileNets, due to their lighter architecture, are the only network models with which it has been possible to perform successfully the inference inside the Jetson Nano.

Table 6.2: Model performances with the DenseNet201 backbone.

DenseNet201 model performances				
Class	Prec	Rec	F1	Support
brushing_teeth	0.90	0.77	0.83	183
cleaning_floor	0.82	0.90	0.86	128
cleaning_toilet	0.91	0.87	0.89	70
cleaning_windows	0.81	0.85	0.83	95
crawling_baby	0.93	0.91	0.92	183
dining	0.75	0.87	0.81	95
doing_nails	0.91	0.85	0.88	129
drinking	0.64	0.53	0.58	77
hugging	0.56	0.77	0.65	64
ironing	0.95	0.83	0.89	66
kissing	0.63	0.69	0.66	59
making_bed	0.85	0.82	0.84	91
opening_bottle	0.77	0.63	0.69	98
playing_cards	0.91	0.94	0.93	102
reading_book	0.85	0.85	0.85	177
setting_table	0.86	0.78	0.82	55
using_computer	0.97	0.93	0.95	135
washing_dishes	0.74	0.85	0.79	157
washing_hair	0.81	0.59	0.68	44
washing_hands	0.68	0.77	0.73	132
Acc			0.82	2140
macro avg	0.81	0.80	0.80	2140
weighted avg	0.83	0.82	0.82	2140

For the DenseNet201, having a deeper and more complex architecture, the results were obtained with an 80 *epochs* training, *Adagrad optimizer* with *learning rate* equals to 0.001, *batch size* equals to 4 and *ReLU Activation Function*.
For VGG16 the results were obtained with 80 *epochs* training, *Adagrad optimizer* with *learning rate* equals to 0.001, *batch size* equals to 16 and *ReLU Activation Function*.
Finally, for all three MobileNets the results shown in the tables were obtained with 80 *epochs* training, *Adagrad optimizer* with *learning rate* equals to 0.001, *batch size* equals to 16 and *ReLU Activation Function*.

Table 6.3: Model performances with the VGG16 backbone.

VGG16 model performances				
Class	Prec	Rec	F1	Support
brushing_teeth	0.79	0.77	0.78	183
cleaning_floor	0.83	0.75	0.79	128
cleaning_toilet	0.83	0.79	0.81	70
cleaning_windows	0.85	0.85	0.85	95
crawling_baby	0.85	0.88	0.87	183
dining	0.71	0.92	0.80	95
doing_nails	0.76	0.88	0.81	129
drinking	0.37	0.60	0.46	77
hugging	0.48	0.44	0.46	64
ironing	0.86	0.64	0.73	66
kissing	0.79	0.58	0.67	59
making_bed	0.73	0.79	0.76	91
opening_bottle	0.58	0.52	0.55	98
playing_cards	0.93	0.90	0.92	102
reading_book	0.76	0.81	0.79	177
setting_table	0.73	0.65	0.69	55
using_computer	0.90	0.90	0.90	135
washing_dishes	0.78	0.78	0.78	157
washing_hair	0.54	0.48	0.51	44
washing_hands	0.70	0.52	0.60	132
Acc			0.75	2140
macro avg	0.74	0.72	0.73	2140
weighted avg	0.76	0.75	0.75	2140

The tables show, for each class, the corresponding *Prec*, *Rec* and *F1* values. In the lower part of the tables, on the other hand, there are the average values and the *Acc*. The reported averages include macro average (averaging the unweighted

mean per label) and weighted average (averaging the support-weighted mean per label). Finally, the right side of the table shows the number of samples for each class in the test set.

Table 6.4: Model performances with the MobileNetV2 backbone.

MobileNetV2 model performances				
Class	Prec	Rec	F1	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
Acc			0.77	2140
macro avg	0.76	0.74	0.75	2140
weighted avg	0.77	0.77	0.77	2140

6.1.2 DoubleBranch Performances

The attempt to improve SingleBranch model with the DoubleBranch_DenseNet201, and so with the addition of a further branch to analyze the masks in which only the areas considered interesting for classification were focused, did not succeed. Although several training configurations were tested, DoubleBranch_DenseNet201 was not able to learn.
Not having obtained results with the DoubleBranch_DenseNet201, due to the deep and complex architecture, as well as the scarce quality of the dataset of the RGB

Table 6.5: Model performances with the MobileNetV3Small backbone.

MobileNetV3Small model performances				
Class	Prec	Rec	F1	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
Acc			0.69	2140
macro avg	0.68	0.66	0.66	2140
weighted avg	0.69	0.69	0.69	2140

images and of the masks, it was decided to model a new DoubleBranch with the use of the backbone of the VGG16. The best results obtained during training and testing on the Kinetics dataset are shown in Table 6.7. The results were obtained with the following training setting: 80 training *epochs*, *batch size* of 8, *ReLU Activation Function* and *SGD optimizer* with *learning rate* of 0.0005 and *momentum* of 0.9.

Table 6.8 and Table 6.9 show the comparison between the SingleBranch_VGG16 and the DoubleBranch_VGG16. Table 6.8 shows which model performs better on the Kinetics dataset with the same training *epochs*. Table 6.9 shows the best performing model on the Kinetics dataset for the same training *epochs* and for the same *Adagrad optimizer*, since the *Adagrad optimizer* was the one that gave the best results for the different backbones.

Table 6.6: Model performances with the MobileNetV3Large backbone.

MobileNetV3Large model performances				
Class	Prec	Rec	F1	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
Acc			0.76	2140
macro avg	0.74	0.74	0.74	2140
weighted avg	0.77	0.76	0.76	2140

6.2 Constrained Resource Environment

Regarding the tests carried out on the NVIDIA Jetson Nano, the results that will be reported will refer to the process of optimization and deployment of the models inside the microprocessor and to the performance obtained in the inference phase.

6.2.1 Model Optimization on Jetson Nano

In order to perform inference within the NVIDIA Jetson Nano, the model optimization operation must be performed within the Jetson Nano itself because the procedure takes in consideration the characteristics of the GPU on which the procedure is performed.
It was therefore not possible to perform inference on the Jetson Nano with the

Table 6.7: DoubleBranch model performances with the VGG16 backbone.

DoubleBranch_VGG16 model performances				
Class	Prec	Rec	F1	Support
brushing_teeth	0.70	0.80	0.75	183
cleaning_floor	0.73	0.85	0.79	128
cleaning_toilet	0.74	0.79	0.76	70
cleaning_windows	0.75	0.79	0.77	95
crawling_baby	0.83	0.93	0.87	183
dining	0.81	0.85	0.83	95
doing_nails	0.86	0.79	0.82	129
drinking	0.56	0.47	0.51	77
hugging	0.45	0.44	0.44	64
ironing	0.86	0.73	0.79	66
kissing	0.61	0.64	0.63	59
making_bed	0.77	0.74	0.75	91
opening_bottle	0.79	0.46	0.58	98
playing_cards	0.91	0.91	0.91	102
reading_book	0.79	0.79	0.79	177
setting_table	0.73	0.73	0.73	55
using_computer	0.94	0.89	0.92	135
washing_dishes	0.79	0.75	0.76	157
washing_hair	0.77	0.39	0.52	44
washing_hands	0.55	0.67	0.60	132
Acc			0.76	2140
macro avg	0.75	0.72	0.73	2140
weighted avg	0.76	0.76	0.75	2140

Table 6.8: SingleBranch_VGG16 and DoubleBranch_VGG16 comparison with same *epochs* training.

Epochs	Network Model
30	SingleBranch_VGG16 (*Adagrad*, *batch size* 16)
60	DoubleBranch_VGG16 (*SGD*, *batch size* 8)
80	DoubleBranch_VGG16 (*SGD*, *batch size* 8)

models that have been previously optimized on Google Colab.
The Jetson Nano, as already mentioned, has limited resources as it only has 4GB of RAM. For this reason, many of the models discussed in the previous chapters could not be optimized and used for inference within the Jetson Nano because of the Out Of Memory error due to the complexity and high number of parameters of

Table 6.9: SingleBranch_VGG16 and DoubleBranch_VGG16 comparison with same *epochs* training and *optimizer*.

Epochs	Network Model
30	SingleBranch_VGG16 (*batch size* 16)
60	SingleBranch_VGG16 (*batch size* 16)
80	SingleBranch_VGG16 (*batch size* 16)

the network models.
Regarding the SingleBranch models, the FP32 optimization in TF-TRT was successfully done only for the SingleBranch with MobileNetV2, MobileNetV3Small and MobileNetV3Large backbones. As can be seen in Table 6.1, the computing power of the NVIDIA Jetson Nano is able to successfully support the optimization and loading of network models with less than 3 million parameters. In fact, MobileNetV2 has 2,257,984 parameters, MobileNetV3Small 1,031,848, and MobilenetV3Large 2,667,688.
Regarding the DoubleBranch instead, it was not possible to perform the optimization in TF-TRT because it has a much more complex architecture than the SingleBranch and in particular because there were incompatibility problems between the TensorFlow versions. As we know, the DoubleBranch has one of the two branches that takes in input the masks that are the result of the execution of the MaskRCNN. The MaskRCNN is implemented in version 1.x of TensorFlow, but the TF-TRT methods and functions for optimizing models are implemented in version 2.x of TensorFlow, so incompatibility problems arose that made it impossible to use DoubleBranch.

6.2.2 Model Results and Performances

The results discussed in this section refer to tests performed with only SingleBranch models with MobileNetV2, MobileNetV3Small and MobileNetV3Large backbones.
Once the model has been optimized, the next step is to load it into memory.
From the tests carried out, despite the three previous models being quite light, it appears that only one model at a time can be loaded into RAM.

Below will be an overview of the results obtained for the three MobileNets. The average loading times of the models, libraries and utility functions and the first inference are shown, and the average inference times and FPS are also reported. The inference time tests were performed both with the results displayed on screen (examples of results displayed on screen are shown in Fig.6.1, Fig.6.2 and Fig.6.3) and without the results displayed on screen but using only the results printed on the console.

The results obtained are as follows:

- **MobileNetV2**
  - ~9 minutes for libraries and model loading and first inference;
  - ~0.25s average inference time with camera visualization (FPS: ~27);
  - ~0.25s average inference time without camera visualization (FPS: ~31).
- **MobileNetV3Small**
  - ~6 minutes for libraries and model loading and first inference;
  - ~0.08s average inference time with camera visualization (FPS: ~29);
  - ~0.08s average inference time without camera visualization (FPS: ~67).
- **MobileNetV3Large**
  - ~9 minutes for libraries and model loading and first inference;
  - ~0.20s average inference time with camera visualization (FPS: ~25);
  - ~0.20s average inference time without camera visualization (FPS: ~37).

Fig.6.1 shows an example of correct classification of the activity *using_computer*, while Fig.6.2 and Fig.6.3 show two incorrect classifications of the same class of activity.

Remembering that the input of the models is in terms of size (*batch*, *timestep*, *width*, *height*, *channels*), the maximum number of *batches* that the model can analyze was also tested.
Being able to analyze a number of *batches* greater than 1 means that it is possible to analyze the stream of several cameras in parallel.
In order to do this, several *batches* were created with a single image repeated for 10 *timesteps*. The result of the tests performed on the three MobileNets is that the models were able to perform inference on more than 10 *batches*.

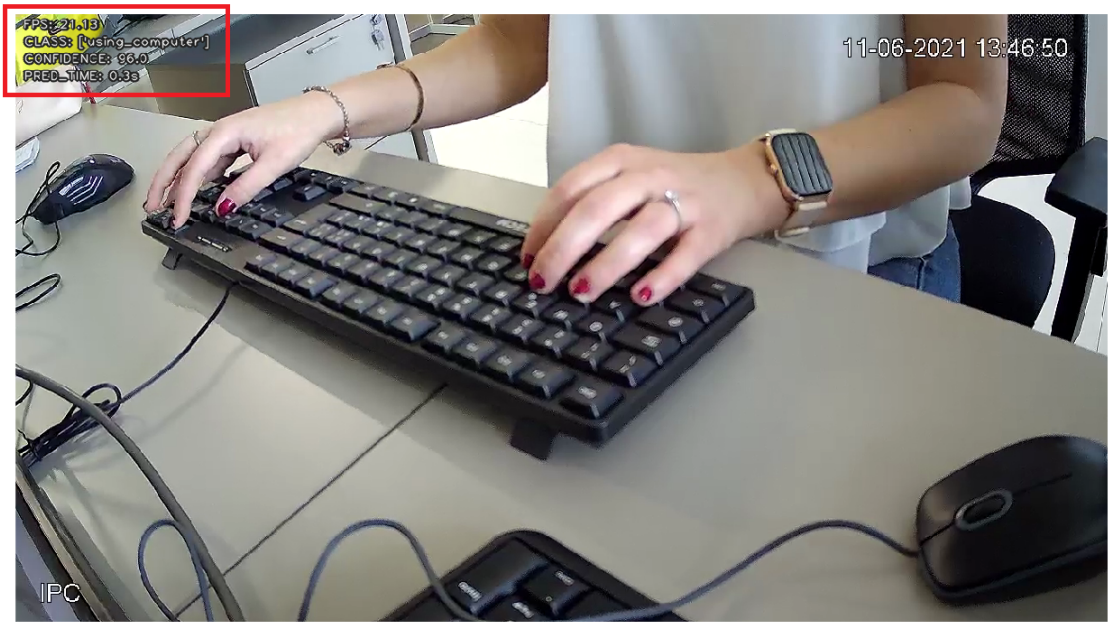


Figure 6.1: Right prediction of *using_computer* class.

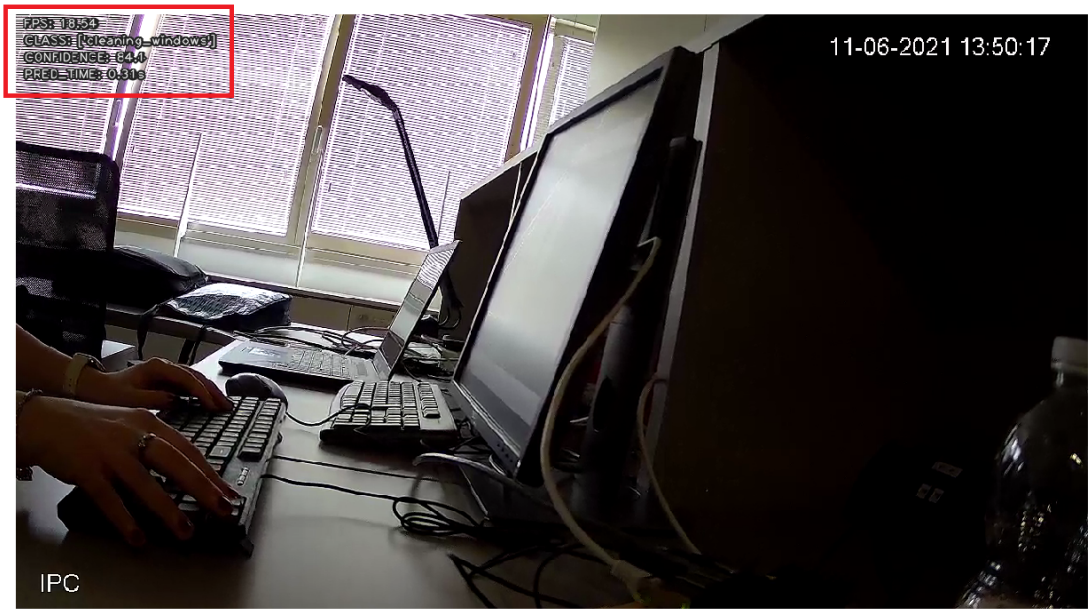


Figure 6.2: Wrong prediction of *using_computer* class.

Figure 6.3: Another wrong prediction of *using_computer* class.

Chapter 7

Discussion

The aim of this work is to create a framework able to classify in real-time indoor human activities from a video stream and to test it in a real case of Edge Computing oriented application.
To do this, different models of Neural Networks have been studied and modelled. The most performing model obtained is the single branch model with the DenseNet201 as backbone for the extraction of the spatial features (Table 6.1).
Its deep architecture, despite being quite complex, allowed to obtain the best results. In fact, performances of 82% *Acc*, 83% *Prec*, 82% *Rec* and 82% *F1* were achieved.
As already mentioned, the architecture of the DenseNet201 is quite complex, due to the *Dense* and recursive blocks, as well as to the high number of *parameters* (just over 18 million), so it was necessary to lower the *batch size* to 4 in order to train the model on Google Colab without encountering an Out Of Memory error due to the variable GPU assignment by Colab.
It can also be seen from Table 6.1 that increasing the number of model *parameters*, and so its complexity, does not lead to an improvement in performance. In fact the most complex models in terms of the number of *parameters*, the two ResNets and the InceptionResNet, did not lead to an increase in performance. On the other hand, models with a lower number of *parameters*, such as the MobileNets, were also able to achieve acceptable performance.
Looking at the results shown in Table 6.2, Table 6.3, Table 6.4, Table 6.5 and Table 6.6, referring respectively to the DenseNet201, VGG16, MobileNetV2, MobileNetV3Small and MobileNetV3Large, the performances result unbalanced among the classes: in fact, observing the *Support* we can notice that the dataset results unbalanced among the classes and therefore the classes like *hugging*, *kissing* and *washing_hair* reach performances much lower than those with high *Support* like for example *crawling_baby* and *using_computer*. This is not entirely true, in fact some classes with low *Support*, such as *ironing* and *setting_table*, achieve comparable

performance to those with higher *Support*. This may suggest that the problem is related to the quality of the dataset which means that some classes are represented by less significant images.
A good solution, to improve the performance of those classes with limited *Support*, could be Data Augmentation. With this last technique we could increase the samples of those classes that have obtained limited performance by rotating or mirroring the existing images.

An attempt to improve the performance of the SingleBranch_DenseNet201 model, by adding an additional branch to analyze masks that only highlighted portions of interest, failed. The DoubleBranch_DenseNet201 has a very complex architecture and this led to a total lack of learning. The reason due to the lack of learning is probably not only due to the complex architecture, but also to the poor quality of the RGB images and especially of the masks created (see Fig.7.1).

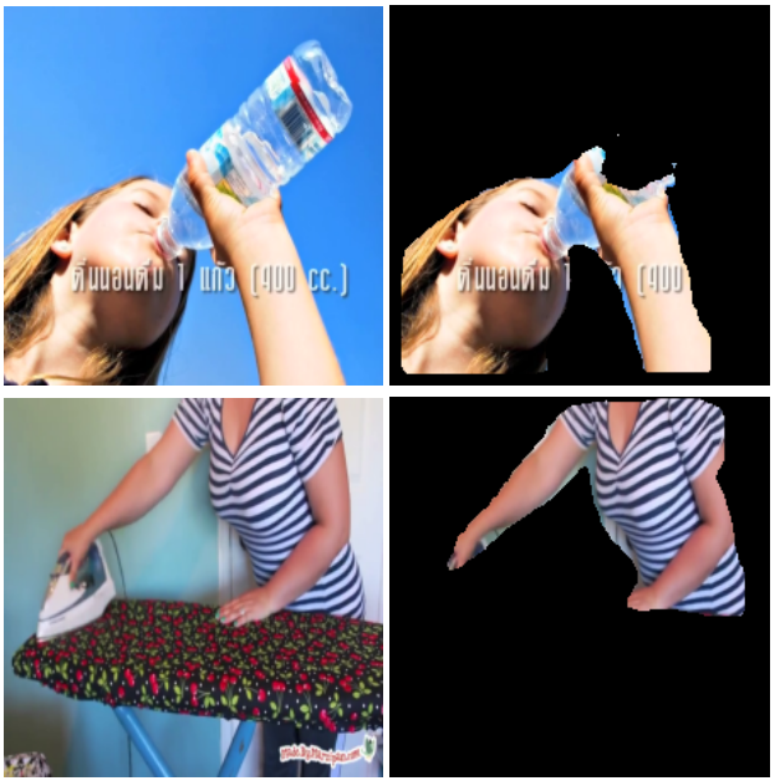


Figure 7.1: two examples of masks in which the portions of interest (bottle and iron) are eliminated.

Due to the complexity of the DenseNet201, it was therefore decided to try a further attempt using the VGG16 which is simpler and widely used in the literature for classification tasks.

Comparable results with those obtained with the SingleBranch_VGG16 were achieved by the DoubleBranch_VGG16 (see Table 6.7). The best performances achieved by DoubleBranch_VGG16 are 76% *Acc*, 76% *Prec*, 76% *Rec* and 75% *F1* (with the *SGD optimizer*) compared to 75% *Acc*, 76% *Prec*, 75% *Rec* and 75% *F1* achieved by SingleBranch_VGG16 (with the *Adagrad optimizer*).
Evaluating the performance at individual class level, on the other hand, it can be seen that DoubleBranch_VGG16 has more balanced values between the various classes.
Comparisons between the two models were also carried out with the same training *epochs* (see Table 6.8) and training settings (see Table 6.9): with the same *epochs* it can be seen that, although the performance values are comparable, with increasing *epochs* the DoubleBranch_VGG16 obtains slightly better values, while with the same *epochs* and *Adagrad optimizer* the SingleBranch_VGG16 obtains better performances. Since the SingleBranch_VGG16 did not produce any results using the *SGD optimizer*, the comparison with *Adagrad* was the only possible way to make a direct comparison between the two models with the same configuration.

Once the models were created, the model optimization procedure was performed with TF-TRT in order to perform inference within the Jetson Nano. The optimization procedure had to be performed inside the Jetson Nano as it considers the characteristics of the GPU.
As mentioned in the previous chapters, the Jetson Nano has only 4GB of RAM so only the lightest models were successfully optimized without running into an Out Of Memory error. As far as SingleBranch models are concerned, only the three MobileNets have been successfully optimized, while it has not been possible to convert the DoubleBranch models, first because the architecture is too complex for the computational power of the Jetson Nano, and also because there are incompatibility problems between TensorFlow versions: the model optimization procedure is implemented in 2.x version of TensorFlow, while the MaskRCNN, necessary for creating the masks of the second branch of the model, is implemented in version 1.x of TensorFlow.

Within the Jetson Nano, only the three MobileNets were tested for inference because they were the lightest. It was also tested that only one model at a time could be loaded into RAM.
Regarding the comparison of performances, it must be remembered that at the level of *Acc* the MobileNetV2 is the most performing model among the three MobileNets (see Table 6.1), while at the level of complexity of the model, the MobileNetV2 and MobileNetV3Large are the heaviest, with 2,257,984 and 2,667,688 number of *parameters* respectively.
The loading time of the model, libraries and the execution of the first inference depends strongly on the complexity of the model: it is in fact comparable between

the two heaviest MobileNet (MobileNetV2 and MobileNetV3Large), while for the MobileNetV3Small, which has half the number of *parameters*, there is a decrease in these times.
The complexity of the model also affects the average inference time, with MobileNetV3Small having the lowest average inference times compared to the other two more complex MobileNets.
Analyzing the results obtained from the three MobileNet models individually, it can be seen that the on-screen display of the results does not affect the average inference time compared to just printing the results in the console.
As might be expected, however, the FPS increases, sometimes significantly, if the results are printed only in the console.
Tests were also performed to evaluate the *batch* limit on which the model and the Jetson Nano are able to perform inference. It was possible to perform inference on more than 10 *batches*, which means that it is able to perform inference on a larger number of cameras. This will lead to the realization of a multi-threading mechanism for real-time and parallel inference of multiple video streams from multiple cameras.

From the tests carried out on the Jetson Nano, and observable in Fig.6.1, Fig.6.2 and Fig.6.3, we realized that for some classes of activities, in this case for example *using_computer*, the dataset is too focused only on hands and keyboard (see Fig.7.2). By focusing only on hands and keyboard, the model is able to predict well and with high confidence the class *using_computer* (Fig.6.1), while, when we focus also on elements of the surrounding environment, such as windows and panels, the model fails to predict the class (Fig.6.2 and Fig.6.3).

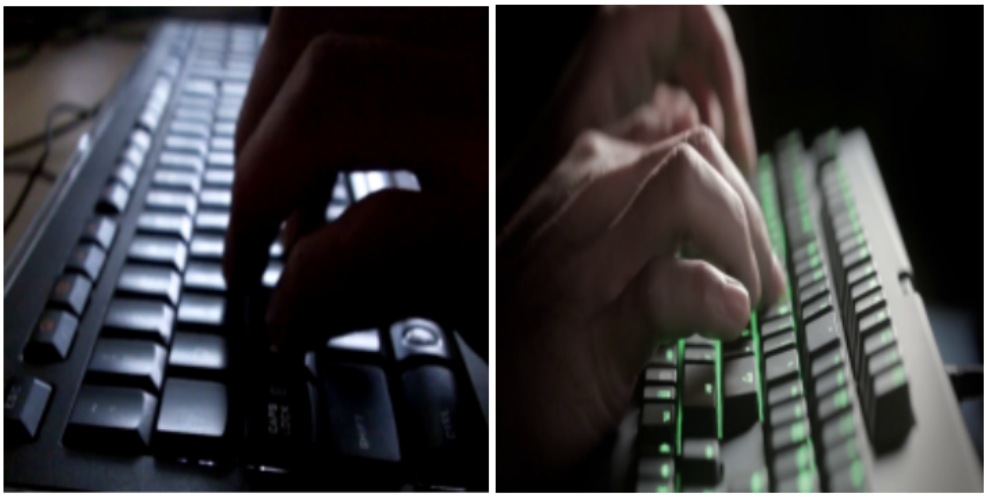


Figure 7.2: Two examples of *using_computer* class RGB images that focus on hands and keyboards.

Chapter 8

Conclusions and future work

In recent years, HAR has received considerable attention from the scientific community due to its many application domains. HAR systems can be mainly used in the care sector, typically performing activities such as the identification of potentially dangerous activities or situations for the elderly, and in the context of video surveillance systems to automate the process of detection of critical situations and alarm signalling.
Several approaches, falling into two categories, to HAR have been proposed by the scientific community and each of them differs from the others depending on the type of sensor used. The first category includes sensor-based approaches, which use sensors worn by agents and those attached to objects in the environment, and the second category includes image-based approaches, which use sensors directly fused into the environment. With the image-based method, RGB images extracted from cameras placed in the environment are analyzed. The scientific community has proposed different architectures: both single branch models that analyze the RGB frames, and double branch models that analyze the RGB frames and the optical flow.
The objective of this thesis was to elaborate and implement a Deep Learning framework able to offer a solution to the task of the Indoor HAR through the use of CNNs and RNNs able to elaborate the spatial and temporal features extracted from RGB images coming from a real-time video camera stream. The capabilities of the framework will then be evaluated in a real-case of Edge Computing oriented application.

Various network architectures have been proposed in this work. They are divided into single branch and double branch architectures. A *TimeDistributed* wrapper was added to the SingleBranch model layers to guarantee the ability to analyze

image sequences and different backbones were tested for spatial feature extraction in order to identify the best performing model. The best model obtained is the SingleBranch_DenseNet201 with an *Acc* of 82%.
The DoubleBranch models were also wrapped by a *TimeDistributed* layer for the analysis of image sequences and were implemented to analyze both RGB image sequences and RGB masks with only the portions of interest highlighted. It was not possible to model a DoubleBranch_DenseNet201 architecture due to the complexity of the DenseNet201 model, but a DoubleBranch_VGG16 model was created, which achieved comparable performances, and for some metrics even slightly better, than the SingleBranch_VGG16.
A real-case of application oriented to Edge Computing has been handled with the developed framework.
The process of optimizing and deploying the models within the NVIDIA Jetson Nano was also successfully implemented.
However, due to the limited computing power of the Jetson Nano, only the three MobileNets models were successfully optimized for inference. Of these three, the best performing model is the MobileNetV2 with an *Acc* of 77%. Finally, a mechanism for real-time inference was successfully implemented on the Jetson Nano from the video stream coming from an IP Camera.

Relative to future works, a Data Augmentation could be carried out to balance the classes that have a relatively low number of samples, it could be also carried out a focused training for the SingleBranch_DenseNet201 and for the SingleBranch_MobileNetV2 in order to increase the level of learning of the best performing models obtained in environments with "unlimited" and limited resources respectively increasing the training *epochs* and testing further training settings.
Having used the MaskRCNN with the pre-trained weights, we could also further train the MaskRCNN in order to get better masks.
When dealing with resource-constrained hardware, a good test might be to change the encoding standard to H.265 for video compression and see if there are any advantages, as well as trying to convert models to FP16 to see if there are any advantages in efficiency for inference. We can also try to use hardware with higher computing power so that we can do the optimization and deployment of heavier but more performant models such as the SingleBranch_DenseNet201.
Finally we have tested that the Jetson Nano and the model support multiple *batch* inference.

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, 6, 8, 14, 15, 19).

[2] J. K. Dhillon and A. K. S. Kushwaha et al. «A recent survey for human activity recoginition based on deep learning approach». In: *fourth international conference on image information processing* (2017). DOI: 10.1109/ICIIP.2017.8313715 (cit. on p. 3).

[3] O. D. Incel and M. Kose et al. «A review and taxonomy of activity recognition on mobile phones». In: *BioNanoScience* (2013). DOI: 10.1007/s12668-013-0088-3 (cit. on pp. 4, 5).

[4] O. D. Lara and M. A. Labrador et al. «A survey on human activity recognition using wearable sensors». In: *IEEE communications surveys & tutorials* (2012). DOI: 10.1109/SURV.2012.110112.00192 (cit. on p. 7).

[5] 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. 11, 19–22).

[6] Serkan Balli, Ensar Arif Sağbaş, and Musa Peker. «Human activity recognition from smart watch sensor data using a hybrid of principal component analysis and random forest algorithm». In: *Measurement and Control* 52.1-2 (2019), pp. 37–45. DOI: https://doi.org/10.1177/0020294018813692 (cit. on p. 13).

[7] Charissa Ann Ronao and Sung-Bae Cho. «Human activity recognition with smartphone sensors using deep learning neural networks». In: *Expert systems with applications* 59 (2016), pp. 235–244. DOI: https://doi.org/10.1016/j.eswa.2016.04.032 (cit. on p. 14).

[8] J Arunnehru and M Kalaiselvi Geetha. «Automatic activity recognition for video surveillance». In: *International Journal of Computer Applications* 75.9 (2013). DOI: https://doi.org/10.5120/13136-0537 (cit. on p. 16).

[9] K. Simonyan and A. Zisserman et al. «Two-stream convolutional networks for action recognition in videos». In: *arXiv preprint arXiv:1406.2199* (2014) (cit. on pp. 16, 17).

[10] W. Ye and J. Cheng et al. «Two-stream convolutional network for improving activity recognition using convolutional long short-term memory networks». In: *IEEE Access* (2019). DOI: 10.1109/ACCESS.2019.2918808 (cit. on pp. 16–18).

[11] M. Baccouche and F. Mamalet et al. «Sequential deep learning for human action recognition». In: *International workshop on human behavior understanding* (2011). DOI: https://doi.org/10.1007/978-3-642-25446-8_4 (cit. on p. 18).

[12] Jin Zhang, Bo Wei, and Jun Cheng. «HARaaS: HAR as a service using wifi signal in IoT-enabled edge computing». In: (2020), pp. 681–682. DOI: https://doi.org/10.1145/3384419.3430469 (cit. on p. 20).

[13] R. Yamashita and M. Nishio et al. «Convolutional neural networks: an overview and application in radiology.» In: *Insights into imaging* (2018). DOI: https://doi.org/10.1007/s13244-018-0639-9 (cit. on pp. 23, 24).

[14] K. Simonyan and A. Zisserman et al. «Very deep convolutional networks for large-scale image recognition.» In: *arXiv preprint arXiv:1409.1556* (2014). DOI: http://www.robots.ox.ac.uk/Ëœvgg/research/very_deep/ (cit. on p. 35).

[15] K. He and X. Zhang et al. «Deep residual learning for image recognition.» In: *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016). DOI: https://doi.org/10.1109/cvpr.2016.90 (cit. on p. 36).

[16] C. Szegedy and V. Vanhoucke et al. «Rethinking the inception architecture for computer vision.» In: *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016). DOI: https://doi.org/10.1109/cvpr.2016.308 (cit. on p. 36).

[17] C. Szegedy and S. Ioffe et al. «Inception-v4, inception-resnet and the impact of residual connections on learning.» In: *Proceedings of the AAAI Conference on Artificial Intelligence* (2017). DOI: https://ojs.aaai.org/index.php/AAAI/article/view/11231 (cit. on p. 36).

[18] F. Chollet. «Xception: Deep learning with depthwise separable convolutions.» In: *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017). DOI: https://doi.org/10.1109/cvpr.2017.195 (cit. on p. 36).

[19] A. Howard and M. Zhu et al. «Mobilenets: Efficient convolutional neural networks for mobile vision applications.» In: *arXiv preprint arXiv:1704.04861* (2017) (cit. on p. 37).

[20] M. Sandler and A. Howard et al. «Mobilenetv2: Inverted residuals and linear bottlenecks.» In: *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018). DOI: https://doi.org/10.1109/cvpr.2018.00474 (cit. on p. 37).

[21] A. Howard and M. Sandler et al. «Searching for mobilenetv3.» In: *Proceedings of the IEEE/CVF International Conference on Computer Vision* (2019). DOI: https://doi.org/10.1109/iccv.2019.00140 (cit. on p. 37).

[22] G. Huang and Z. Liu et al. «Densely connected convolutional networks.» In: *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017). DOI: https://doi.org/10.1109/cvpr.2017.195 (cit. on p. 37).

[23] M. Tan and Q. Le et al. «Efficientnet: Rethinking model scaling for convolutional neural networks.» In: *International Conference on Machine Learning* (2019) (cit. on p. 38).

[24] B. Zoph and V. Vasudevan et al. «Learning transferable architectures for scalable image recognition.» In: *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018). DOI: https://doi.org/10.1109/cvpr.2018.00907 (cit. on p. 38).

[25] Will Kay et al. «The kinetics human action video dataset». In: *arXiv preprint arXiv:1705.06950* (2017) (cit. on pp. 47, 48).