POLITECNICO DI TORINO

MASTER's Degree in MECHATRONIC ENGINEERING

MASTER's Degree Thesis

HAND GESTURE RECOGNITION FOR HOME ROBOTICS

Supervisors

Prof. MARCELLO CHIABERGE

Candidate

DAVIDE GUARNERI

APRIL 2023

Abstract

Robotics is a sector in deep ferment and constant change. The great interest that this area attracts is due to the ability of robots to carry out demanding and repetitive tasks with higher speed and precision than a human operator, a reason which has led to the strong growth in the development and adoption of large machinery belonging to the important sub-category of industrial robots.

However, world society has also undergone stark changes thanks to rapid technological development in all areas, with the result that, although new lifestyles, new needs, and new problems have arisen, novel solutions that can make the life of people easier have also come to light. From this point of view, a particularly active and lively segment called Service Robotics is coming to the fore, ready to bring clear improvements mainly in contexts such as medicine, precision agriculture, logistics, security, the office, the home, and smart cities, settling down as one of the most promising emerging technological trends. The fascinating side of the development of this sector is the incessant propensity to bring robots closer to humans, making them increasingly collaborative and demonstrating over time that they can perform tasks better and better.

The actual bridge between these two worlds can deservedly be represented by Artificial Intelligence, another technology that is becoming increasingly popular nowadays and allows smartly solving intricate conceptual problems characterized by complex mathematical and computer algorithms behind them.

The project presented in this thesis work is an example of the union of these two cutting-edge disciplines and consists of the development of a deep learning model capable of classifying some types of dynamic hand gestures; the interpretation of the performed gesture provided as output by this model will then be used to make a wheeled robot, designed for a domestic environment, perform some specific maneuvering procedures. To achieve this result, recognizing and classifying a frame-by-frame sequence of hand landmarks coordinates, a 2D Convolutional LSTM Deep Neural Network architecture has been chosen, using a softmax layer as the output layer.

The advantages offered by this solution mainly reside in the absence of communication interfaces, such as touch screens and joysticks, for controlling the robot and in the reduced amount of data to be processed by an algorithm that is also relatively light in terms of size and required computational capacity; these features allow to obtain a remarkable rapidity in classifying hand gestures and executing actions, that makes this solution combinable with other models for better usability and scalable for different contexts in which gesture recognition can be functional.

Acknowledgements

I dedicate this thesis to my family and friends, my perennial supports.

My sincere thanks also go to Fontana Umberto and Master Doctors Cavagnaro Niccolò and Gioachini Luca. Your help was crucial in the first part of this project, the most important.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
Artificial Intelligence

NN
Neural Network

ML
Machine Learning

ANN
Artificial Neural Network

CNN
Convolutional Neural Networks

RNN
Recurrent Neural Network

DNN
Deep Neural Network

DL
Deep Learning

TLU
Threshold Logic Unit

MLP
Multi Layer Perceptron

CV

Computer Vision

Recurrent 2D Convolutional Neural network

2D-RCNN

LSTM

Long Short Term Memory

HGR

Hand Gesture Recognition

DHGR

Dynamic Hand Gesture Recognition

DHG

Dynamic Hand Gesture

FW

Forward

BW

Backward

CW

Clockwise

CCW

Counterclockwise

Chapter 1

Introduction

1.1 Motivation and Aim of the work

Non-verbal communication plays an important role in human interactions and several studies have been conducted on this. To report some of the most significant ones, it is considered proper to mention the experiments done by the psychologists Michael Argyle and Albert Mehrabian. The former outlined what the main forms of nonverbal communication actually are, namely gestures, facial expression, eye contact (or fixed gaze), posture, touch, and spatial behavior (or proxemics)[1]; the latter showed, in one of his studies[2], that what is perceived in a voice message of neutral valence, in the context of a workshop, and emitting the message but expressing a different one with body language, can be broken down as follows:

- Body movements: 55 %
- Vocal aspect (volume, tone, rhythm): 38 %
- Verbal appearance (words): 7 %[1]

[1]The psychologist highlighted the fact that this percentage division in every context must not be

The effectiveness of a message thus depends only minimally on the literal meaning of what is said, and the way this message is perceived is heavily influenced by nonverbal communication factors.

It results that body movements are the most expressive form of non-verbal communication, in particular eye motions, facial expressions, and body gestures. If for a human individual these can, however, be considered easily recognizable and interpretable in any communicative context, this is not so immediate for a robot, in particular if in absence of sensors that would pick up such signs and communicate directly to the machine the data collected about them. This is an interesting as well as fascinating challenge that is posed within the human-machine communication context nowadays and finds a solution in the ever-evolving field of Computer Vision (CV).

In fact, although the use of joysticks, touch screens, or simply other devices wireless connected to the robot may be more accurate, giving greater certainty of the correct reception of the desired instruction and subsequent realization, CV may prove to be the bridge that can take us from a scenario where robots have to be continuously assisted by an operator to one where they can be more autonomous and collaborative towards humans, both in work contexts - with great benefit in the perspective of the development of Industry 4.0 -, like in agriculture - monitoring soils and helping with sowing and harvesting - and in industry, especially in logistics, but also in domestic contexts caring for the elderly or people with disabilities, in medicine, security, and in smart cities development.

The robotics segment that is involved in the development of all these contexts is called Service Robotics, which includes all fields of application in which a robot is not used in heavy industrial manufacturing and actually represents one of the most promising emerging technological trends (according to Statista.com, revenue in the

generalized for every context. In particular, this is valid for emotional or affective communication.

Service Robotics segment is projected to reach €25.45bn in 2023[3]). This thesis project illustrates the realization of a DL model capable of classifying different dynamic hand gestures (DHGs) and its implementation on a robot, which performs different movement procedures based on the outputs returned by the classification model. It is a multidisciplinary project encompassing topics in Service Robotics, Artificial Intelligence (AI), and some of its articulations such as DL and Data Science, used in particular for the creation of an ad-hoc dataset of sequences of frames reporting skeletal data coordinates of hands extrapolated from the videos of Nvidia's larger dataset "NVIDIA Dynamic Hand Gesture Dataset" (nvGesture)[4]. The robot in question is intended for a preferably domestic context being of small dimensions, while the technology which guides it within the environment can essentially also be applied to different robots since this is based on a video classification network. The aspects that I chose to prefer in designing the network are the quickness in recognizing the gesture, reducing as much as possible the time gap between this phase and the actual execution of the corresponding command by the robot, and therefore also computational lightness and a high level of classification reliability based on the classical metrics of accuracy, precision, recall, and F-Score. To pursue this guideline, I chose to design a truly simple architecture for the Machine Learning (ML) model, that is a 2D Recurrent Convolutional Neural Network (2D-RCNN), that can be fed with arrays containing the gestures in form of skeletal data provided by using Google's model "Mediapipe"[5][6][7] to obtain a strong reduction in the number of data collected compared to using a more classic video classification method.

1.2 Background concepts

The focus of this thesis project is the development of a DL model capable of classifying videos of the selected types of DHGss captured by a camera. It is

therefore useful to recall some theoretical notions related to DL and CV and give them an expository outline. But first is better to see where these concepts come from by analyzing a brief outline of Machine Learning, and for this purpose, I will mainly refer to the text that I have mostly used in the study of these subjects, namely "Hands-on Machine Learning with Scikit-Learn, Keras and Tensorflow" by Aurélien Géron[8].

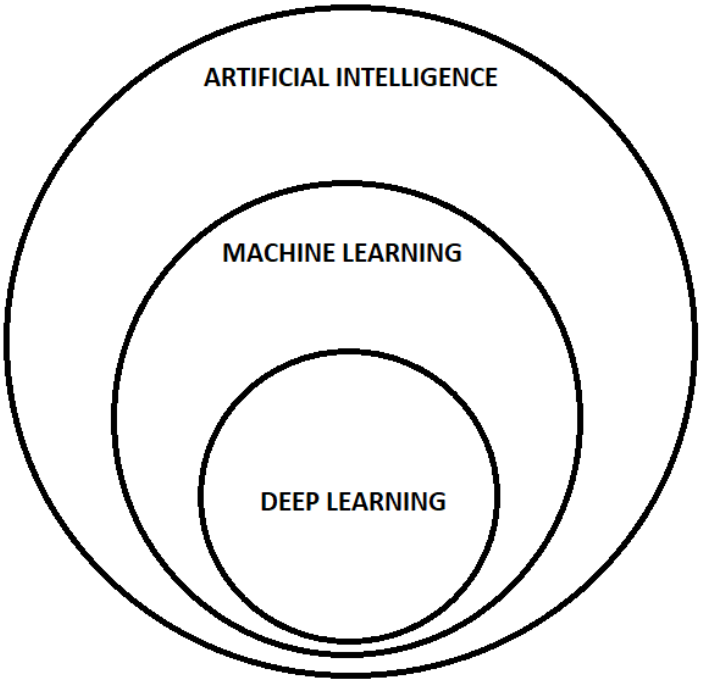

Figure 1.1: AI map

1.2.1 Machine Learning overview

Artificial Intelligence is defined as the theory and development of computer systems able to perform tasks that normally require human intelligence, such as visual perception (better known as Computer Vision), written and spoken human language recognition, decision-making, and translation between languages. A major branch of AI is Machine Learning, which is a subfield of Computer Science that enables

computers to learn without being explicitly programmed.[9] In order to do this, in ML estimators[2], which are algorithms based on mathematical models with tunable parameters called *hyperparameters*, are used. The learning phase of an estimator consists in continuously tuning these parameters to fit the model with the available data given as input.

There are so many different types of ML algorithms that it is useful to classify them into broad categories, based on the following criteria:

1. Whether or not they are trained with human supervision: Supervised, Unsupervised, and Reinforcement Learning.

2. Whether or not they can learn incrementally on the fly: Online versus Batch Learning.

3. Whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model, much like scientists do: Instance-based versus Model-based Learning.

Many problems can be solved with the help of ML, and even more are the approaches, represented by the algorithms, that can also be used taking into account the type and number of data available. Let's consider the cases highlighted in the first point of the list above.

In Supervised Learning, the training data fed to the algorithm include the desired solutions, called labels, and the two tasks covered are *classification*, where the model has to distinguish between two (binary classification) or more (multi-class classification) classes of data provided, and *regression*, where the model has

[2]For the sake of comprehension, the term "model" in general ML applications refers to the mathematical model on which an ML algorithm is based, while in DL applications the term "model" refers to the considered artificial neural network.

to predict a numerical value, called target, given a set of values called features or predictors. Some of the most important Supervised Learning algorithms are K-Nearest Neighbors, Linear Regression, Logistic Regression, Support Vector Machines, Decision Trees, Random Forests, and finally, Neural Networks (NNs), which are the object of interest of an essential sub-branch of ML.

In Unsupervised Learning, instead, the training data are unlabeled and the algorithms identify commonalities in the data and react based on the presence or absence of such commonalities in each new piece of data; hence, some of the covered tasks are *visualization and dimensionality reduction*, in which the goal is to simplify the data without losing too much information, *anomaly and novelty detection* and *clustering*, in which the model tries to detect groups, or even subgroups (hierarchical clustering), of similar objects.

Finally, in Reinforcement Learning, the learning system, called *agent*, can observe the environment, select and perform actions, and get *rewards* or *penalties* in return. It must then learn by itself what is the best strategy, called *policy*, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation.

Briefly summarizing the above concepts, Machine Learning can help solve very complex tasks without programming from scratch; more than one solution, represented by a trainable model, can be used to solve one of these tasks. The bottleneck of the problem is not the choice of the most suitable model, but the data available for training: if a model that can process with the maximum possible accuracy every input provided during the so-called "inference phase" is desirable, a lot of data should be provided during training. Therefore, data should also cover all the different output cases homogeneously and be pre-processed; the pre-processing phase consists in cleaning the data and getting them adequately prepared, managing the missing values, converting the categorical features into numbers and

transforming them into a format that the estimator can understand. Although there are many ML approaches to solving a task and every one of them is based on complex mathematical and statistical algorithms, the ML library Scikit-learn provides a simple flowchart map to follow in order to highlight the problem and the right estimator to use based on the available data, and also coding tools in order to develop models and manipulate data to feed them with[10].

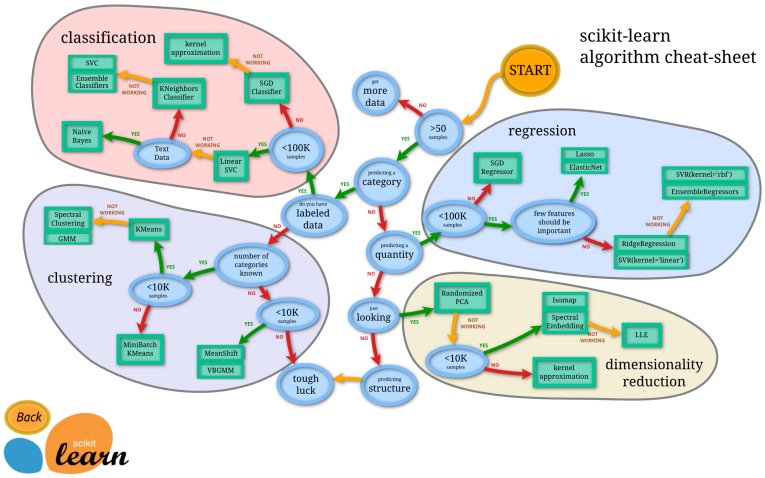

Figure 1.2: Scikit-learn flowchart map on estimator selection

In general, taking as an example a supervised learning case, the training process is carried out according to the following code framework:

Algorithm 1 Estimator training with some Scikit-learn functions

1: ▷ X and y are the features matrix and the labels vector of the dataset
2: ▷ Split X and y in X_{train}, y_{train}, X_{test}, y_{test} using the $train\_test\_split$ function
3: ▷ Set the classifier using the desired library
4: ▷ Fit and train the estimator with X_{train}, y_{train} data using the fit function

1.2.2 Deep Learning

ML also includes computational models and algorithms that have similar structures and functions to the brain's biological NNs. These computational models are

often referred to as Artificial Neural Networks (ANN). When these ANN process information from numerous input flows, they have the ability to "learn" and alter their structure in much the same way that the neurons in our brain are altered with memory. The sub-branch of AI, coming from ML, that concerns ANN is called Deep Learning. Deep Learning calls for these NNs to be organized in a multiple-layered hierarchical structure, where the output of a layer is the input of the successive one. Given these premises, the first layer of the network has therefore been conventionally called as *input layer*, the last as *output layer*, and those placed between these two *hidden layers*; when an ANN contains a deep stack of hidden layers, it is called a Deep Neural Network (DNN). The field of DL studies DNNs, and more generally models containing deep stacks of computations. Finally, the fundamental units that make up each layer have been defined as *neurons*.

In 1943, mathematicians Walter Harry Pitts and Warren McCulloch released a paper called "A Logical Calculus of Ideas Immanent in Nervous Activity"[11], which proposed the first mathematical model of a NN. The basic unit of this network model, the formal representation of a single neuron, is still the standard reference in the field of NNs and is often referred to as the McCulloch–Pitts neuron, a unit that could only perform logical AND, OR and NOT operations and comparisons, thus the only inputs and the outputs were 0 and 1.

In 1957 Frank Rosenblatt invented the *Perceptron*, which is based on a slightly different artificial neuron called a Threshold Logic Unit (TLU). The inputs and output are numbers (instead of binary ON or OFF values), and each input connection is associated with a weight. The TLU computes a weighted sum of its inputs ($z = w_1x_1 + w_2x_2 + ... + w_nx_n = x^Tw$), then applies a step function to that sum and outputs the result: $h(x) = step(z)$. The inputs x_i that a TLU receives come from TLUs arranged in the upper layer; a single layer is composed of a certain amount of TLUs and a bias neuron, which outputs a constant value b (and thus

has no input), and if all the neurons in a layer are connected to every neuron in the previous layer (i.e., its input neurons), the layer is defined as *fully connected* or *dense* layer. The output of the fully connected layer is represented by the following equation:

$$h_{W,b}(X) = \phi(XW + b) \tag{1.1}$$

In this equation:

- X represents the matrix of input features. It has one row per instance and one column per feature.
- The weight matrix W contains all the connection weights except for the ones from the bias neuron. It has one row per input neuron and one column per artificial neuron in the layer.
- The bias vector b contains all the connection weights between the bias neuron and the artificial neurons. It has one bias term per artificial neuron.
- The function ϕ is called the activation function: when the artificial neurons are TLUs, it is a step function.

Perceptron, therefore, represents the basis of modern NNs. However, they offered greater scope for innovation only by undermining some of their characteristics which, if kept fixed, would severely limit their use.

First of all, the training of the network consists of a continuous update of the weights of every TLU following a variant of the so-called *Hebbian Rule*[3]: more specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produces a wrong

[3]From the study reported by the behavior psychologist Donald Hebb in 1949 in "The Organization of Behavior" talking about the connections between biologic neurons.

prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction. The rule is shown in the following equation:

$$w_{i,j}^{(nextstep)} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i \tag{1.2}$$

In this equation:

1. $w_{i,j}$ is the connection weight between the i^{th} input neuron and the j^{th} output neuron.
2. x_i is the i^{th} input value of the current training instance.
3. $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance.
4. y_j is the target output of the j^{th} output neuron for the current training instance.
5. η is the learning rate.

If the training instances are linearly separable, this algorithm will converge to a solution (Perceptron convergence theorem).

Summarizing and highlighting the limitations of these networks, Perceptrons are artificial networks that solve complex, but linear, problems (like simple classifications) based on a layer of TLUs that process input values coming from the input layer and return predictions as the output of the network. These predictions are also based on a hard threshold represented by the step activation functions of the neurons. As things stand, Perceptrons did not represent a valid and robust solution, being also dropped by researchers in favor of higher-level problems such as logic, problem-solving, and search. For example, as depicted by Marvin Minsky and Seymour Papert in their 1969 monograph "Perceptrons: An introduction to Computation Geometry"[12], Perceptrons are not even able to solve the trivial Exclusive OR (XOR) classification problem.

It turns out that some of the limitations of Perceptrons can be eliminated by stacking multiple Perceptrons. The resulting ANN is called Multilayer Perceptron (MLP). An MLP is composed of one (passthrough) *input layer*, one or more layers of TLUs, called *hidden layers*, and one final layer of TLUs called the *output layer*.

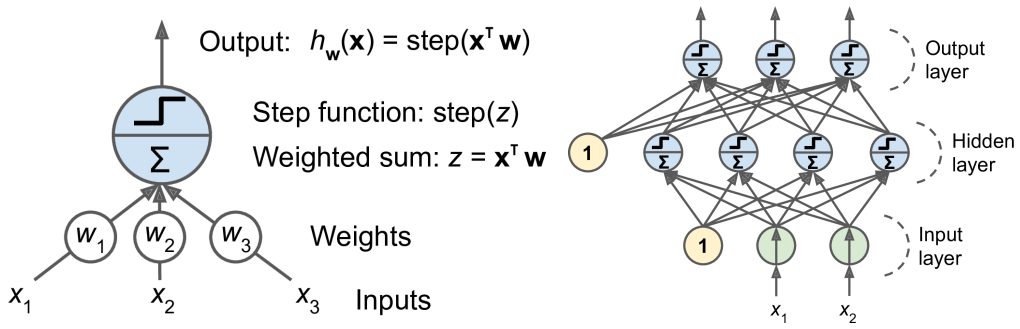

Figure 1.3: Images taken from [8] illustrating a TLU neuron with a step function as activation function(left) and a Multilayer Perceptron artificial NN

The so-obtained MLPs could also solve the XOR problem but were difficult to train due to the multi-layered configuration and the step activation function of TLUs that make the Hebbian rule and the higher complexity of the model useless. Finally, in 1986, David Rumelhart, Geoffrey Hinton, and Ronald Williams introduced the Backpropagation training algorithm, which is still used today, in their paper "Learning representations by back-propagating errors"[13]. The algorithm can be summarized in the following steps:

1. The model handles one mini-batch at a time (for example, containing 32 instances each), and it goes through the full training set multiple times. Each pass is called *epoch*.

2. Each mini-batch is passed to the network's input layer, which sends it to the first hidden layer. The algorithm then computes the output of all the neurons in this layer (for every instance in the mini-batch). The result is passed on to the next layer, its output is computed and passed to the next layer, and

so on until the output of the last layer is obtained. This is the forward pass, and it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.

3. Next, the algorithm measures the network's output error using a loss function that compares the desired output and the actual output of the network and returns some measure of the error.

4. Then it computes how much each output connection contributed to the error. This is done analytically by applying the chain rule, which makes this step fast and precise.

5. The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until the algorithm reaches the input layer. As explained earlier, this reverse passage efficiently measures the error gradient across all the connection weights in the network by propagating the error gradient backward through the network (hence the name of the algorithm).

6. Finally, the algorithm performs a Gradient Descent step to tweak all the connection weights in the network, using the error gradients it just computed.

In order for this algorithm to work properly, its authors made a key change to the MLP's architecture: they replaced the step function with the logistic (sigmoid) function. This was essential because the step function contains only flat segments, so there is no gradient to work with (Gradient Descent cannot move on a flat surface), while the logistic function has a well-defined nonzero derivative everywhere. Furthermore, these functions introduce some non-linearity between layers, validating the multi-layer configuration of the network; in fact, composing more linear functions (coming from every layer of the network) gives

as result another linear function, the same as keeping only one layer. Thus, the backpropagation algorithm works well with many other activation functions, not just the logistic function. The table below reports some examples of the most common ones.

ReLU	$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases}$
Softmax	$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} i = 1, ..., N$
tanh	$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$

Table 1.1: Examples of activation functions, operating either element-wise or vector-wise, depending on the function

From a mathematical point of view, these are all functions that are differentiable and zero-centered, so that their output is symmetrical at zero and so do the gradients.

The advantages of using ANNs to solve Regression and Classification tasks instead of more classical ML estimators reside in the following characteristics:

1. Ability to learn complex patterns:
2. Improved performance
3. Ability to handle large datasets
4. Continuous improvement

However, it's important to note that DL models can be computationally expensive and may require large amounts of data to train effectively. The training itself, also, due to the presence of many hyperparameters to set and tune and the general

complexity of the model, it's a more intricate step to realize and requires a bigger amount of time than training an estimator with Scikit-learn because, in general, a trial & error process is needed. Recalling Algorithm1, the procedure for training MLPs is quite similar, and is also possible to get already developed NNs used in similar projects, but generally, a NN can be constructed from scratch characterizing every aspect of it.

First of all, the training of a NN needs the data to be split differently, dividing the dataset in Training set, to make the network learn patterns and behavior associating features with their true labels, Validation set, in order to verify at every learning epoch if the patterns have been learned properly, and finally a Test set, to evaluate the behavior of the trained model. In general, a proportion of data of 70/20/10 is suggested for splitting. Therefore, for what concerns Training and Validation sets, these can be provided to the network in groups of samples called *batches*. The size of them represents an important hyperparameter to tune: a small batch size can lead the model to better generalization because it is forced to update its parameters more frequently; however, a larger batch size can lead the model to faster training, even with higher stress on GPU, the component able to process mathematical tensors, like data are.

The next phase is the construction of the NN. There are some end-to-end ML frameworks usable with their APIs in order to handle projects involving NNs in every phase, like PyTorch, Caffe, and Tensorflow; in this thesis project, Tensorflow is the utilized framework, also combining it with Keras, a Python built-in API that runs on top of it and that principally handles NNs. Keras allows the construction (by initialization) of the NN using two different APIs:

1. Sequential API: The model is constructed layer-by-layer, obtaining a previously analyzed MLP for example.

2. Functional API: Layer can share inputs and output and be branched, forming

more complex but powerful models. Multiple Inputs and Multiple Outputs are also contemplated by the network.

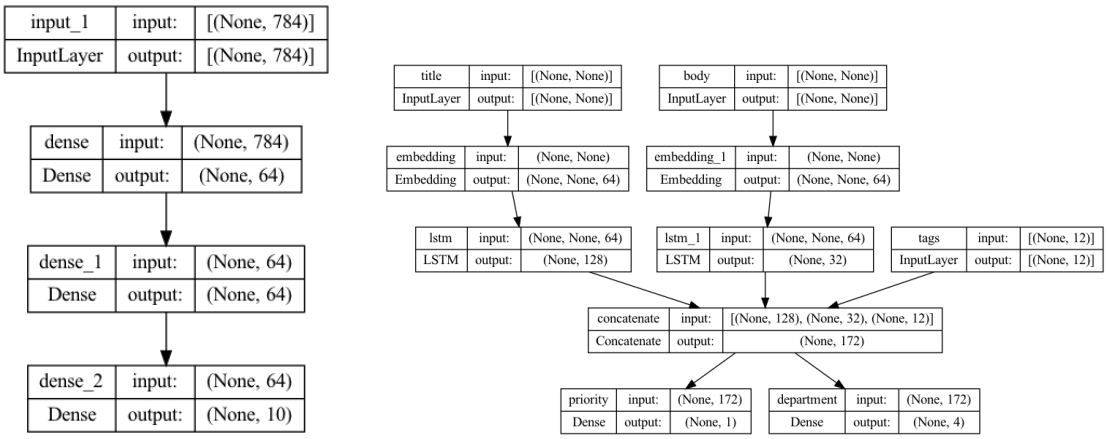

Figure 1.4: Images taken from Keras official websites illustrating a Sequential API's model configuration (left) and a Functional API's one (right)[14].

Focusing on Sequential API, layers have to be added, sized, and characterized one by one. Different types of layers need to be used to build a NN. The fundamental ones are:

1. The previously cited Dense layers.

2. One Flatten layer that transforms the inputs into a one-dimensional array and is used before feeding the dense network.

3. Batch Normalization (or BatchNorm) layers, that work by adjusting and scaling the activations from the previous layer based on the mean and variance of the current mini-batch. Their working principle is explained in the following equation

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \qquad \text{where } y_i = \gamma \hat{x}_i + \beta \tag{1.3}$$

where x_i is the input to the BatchNorm layer for the i-th sample in a mini-batch, μ is the mean of the mini-batch, σ is the variance of the mini-batch,

ϵ is a small constant value ($\approx 10^{-8}$) added for numerical stability, $\hat{x}_i$ is the normalized input for the i-th sample, γ and β are learnable scaling and shifting parameters respectively, and y_i is the output of the BatchNorm layer for the i-th sample.

4. Dropout layers that randomly inhibit a predefined percentage of neurons in the previous layer at every step of training, which becomes less sensitive to slight changes in the inputs, getting a more robust network that generalizes better.

For more particular tasks, other layers are required, like the LSTM layers and the combination of Convolution and Pooling layers; further explanations about these layers are accurately treated in the next subsection, but for sake of comprehension can be briefly reported that LSTMs are useful layers proper of recurrent NNs that let the model learn the temporal associations between parts of a sample, while Convolution and Pooling layers are typical of Classification NNs that usually work with images (or video frames), and let the model learn spatial associations between elements of the same picture. Thus, the contemporary presence of these layers lets the model notice and take also into account spatial and temporal patterns in order to make more accurate predictions in projects in which these features are fundamentals (like the one treated by this thesis project). Furthermore, for what concerns the selected layers, also the number of neurons inside them and the activation functions must be properly set.

Once the model is defined, it must be compiled: during the successive training phase, the weights of the neurons will be continuously modified to maximize a *metric*, which must be pointed out before this phase together with the *optimizer* (i.e. the optimization algorithm of the weights) and a function to be minimized the most possible that takes account of the difference between the predicted outputs of the model and the correct ones, called *loss function*.

After the model has been compiled, like for an ML estimator it must be fitted with input data and trained. The training of a Supervised DL model consists of learning patterns in the training set and testing the deducted associations on validation data; once the predictions are done, the true labels of validation data get revealed and the loss of this learning step is calculated and analyzed by the optimizer, which in the backpropagation adjusts the weights of the neurons. The loss function gets calculated both on training and validation data, and considering "accuracy" as the metric to maximize during the training phase, four curves measuring performances are obtained: *training accuracy, validation accuracy, training loss* and *validation loss*. The best-case scenario that can be outlined at the end of the training is the one in which the validation loss has settled its trend value under an acceptable value, with the consequent achievement by the desired metric of an acceptable high value. If so, the model can be further evaluated by making predictions on the test features and comparing the result with the test labels before passing to the inference phase. Otherwise, two main problems can occur:

1. Underfitting: the model is too simple or not complex enough, and the performance on both the training and validation data are poor. It turns out that validation loss keeps following the training loss and both settle down to an unsatisfying value.

2. Overfitting: the model is too complex and learns the training data too well (like it is "learning by heart"), resulting in poor generalization to new data. It turns out that validation loss stops following the training loss and inverts its descending trend maintaining a growing one.

If one of these two problems occurs, it is possible to make adjustments to the data to obtain significant performance improvements; in particular, it is possible to add new input data and normalize them between 0 and 1 since the input data within a smaller range make the optimization problem more well-behaved or setting a

learning rate schedule in the optimizer. During training, the optimization algorithm calculates the gradients of the loss function with respect to the model parameters and then updates them based on the learning rate. If the learning rate is too small, the model may take a long time to converge to the optimal solution, while if it is too large, the model may overshoot the optimal solution and fail to converge. Hence, it is possible to point out for the learning rate a step function with the desired value or to impose a schedule, i.e. a more complex function varying on training epochs.

Going instead into the specifics of the cases, if underfitting occurs it is good practice to increase the complexity of the model (number of layers and/or neurons inside them), train it for more epochs, or consider more features for input data. On the other hand, if overfitting occurs, it can be useful to reduce complexity, add BatchNorm and Dropout layers, and also apply regularization techniques to the Dense layers (that add penalty terms to the loss function, encouraging the weights to be small) and data augmentation techniques, that are transformations randomly realized on the existing data without permanently modifying them.

MSE (Mean Square Error)	$\frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}$
(Binary) Cross Entropy (average reduction on higher dimensions)	$\frac{\sum_{i=1}^{N}\sum_{j=1}^{C}\hat{y}_i \log(y_{i,j})}{N}$
Categorical Cross Entropy (sum reduction on higher dimensions)	$-\sum_{i=1}^{N}\hat{y}_i + \log\left(\sum_{i=1}^{N}\sum_{j=1}^{C} y_{i,j}\right)$

Table 1.2: Some of the most common loss functions. y is the output of the network, N is the batch size multiplied by the number of outputs (e.g. pixels), C is the number of classes, and $\hat{y}$ is the correct output.

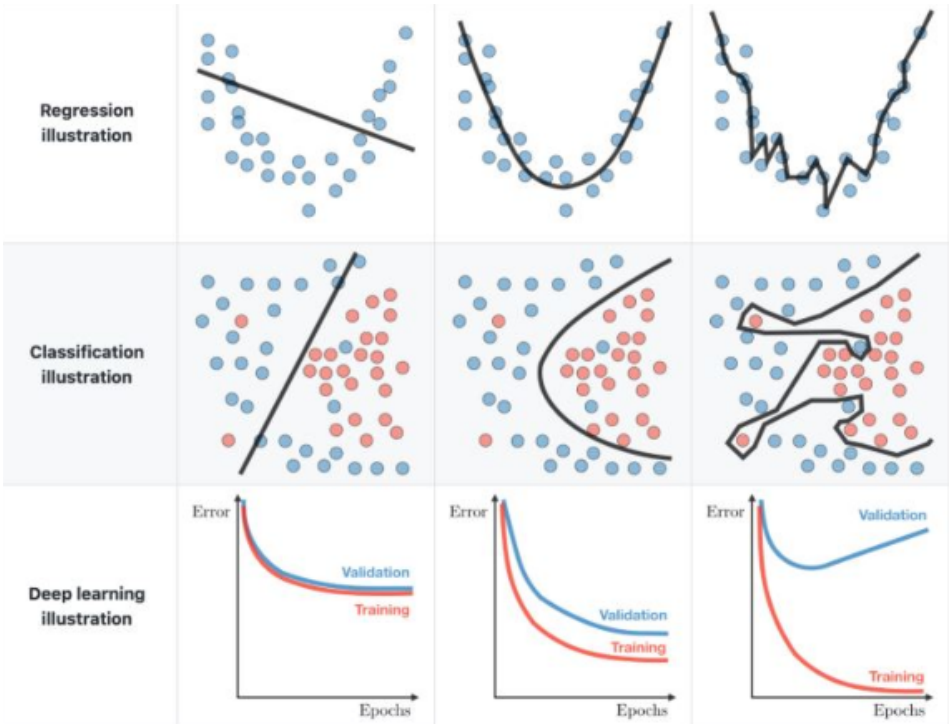

Figure 1.5: Images taken from [15] illustrating underfitting(left), optimal fitting (center), and overfitting (right) behaviors of a model on training and validation data.

1.2.3 Computer Vision

Computer vision is an interdisciplinary field of study concerning algorithms and techniques to allow computers to reproduce functions and processes of the human visual system. In other words, it is the ability to reconstruct a context around the image, giving it a real meaning. In recent years, attention to CV has significantly grown thanks to the advent of increasingly advanced ML techniques, which have made it possible to achieve performances comparable to human ones, and to the diffusion of digital images and videos.

Computer Vision algorithms can carry out more or less in-depth investigations on an image, depending on the techniques used, the type of image, and the type of task performed. Possible tasks include:

1. Image Classification: analysis of the image content and attribution of a label.
2. Object Detection: identification of one or more entities within an image.
3. Image Segmentation: subdivision of the image into sections (e.g. to highlight the pixels of a medical report in which a tumor is found).
4. Face Recognition: recognition of people's faces.
5. Action Recognition: identification of one or more entities and their relationship in time and space, to identify and describe specific actions (for example DHGss, study-case of this thesis work).
6. Visual Relationship Detection: understanding the relationship between objects in an image.
7. Emotion Recognition: detection of the sentiment of an image.
8. Image Editing: changes to an image (e.g. obscuring sensitive data).

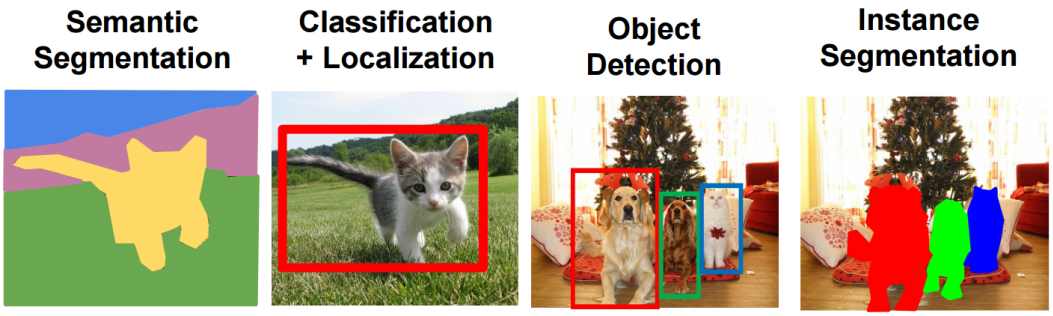

Figure 1.6: Some examples of CV tasks taken from [16]

There are several ways in which a CV architecture can extract information from images – Hand Crafted Features, Computer Vision Features, and Data-Driven Features – chosen individually or combined depending on the needs of the analysis. The former is based on the concept that algorithms can extract and define what is

relevant in the image (e.g. a specific color/shape, area, or size), while the second is based on the subdivision of the image into small regions to allow a deeper analysis. But the real frontier of CV is the set of techniques based on Data-Driven Features, which allow the recognition and classification of images (even natural ones) without having to plan the feature extraction phase, which is carried out by a particular type of DL model. This type of model plays a protagonist role in this thesis project for what concerns frames features recognition, but also in the entire CV landscape considering that it represents the ultimate evolutionary stage of the very first prototype of a model-based solution that was conceived at the dawn of this discipline.

The first studies on CV date back to 1959, with the *Pandemonium*, a model suggested by Oliver Selfridge in the homonym paper "Pandemonium"[17] which presents a framework that describes how the brain processes visual images. The pandemonium model is a fixed, hierarchical model with a pre-defined structure, based on independent entities that process visual stimuli and tell if a pattern has been recognized or not, deciding together the nature of the image. These entities are called *demons*, and, as exposed in [18], are organized in:

1. Image demons, that record the image that is received in the retina.

2. Feature demons, that are many and every one of them represents a specific feature. Each feature demon's job is to "yell" if they detect a feature that they correspond to.

3. Cognitive demons, that watch the "yelling" from the feature demons. Each cognitive demon is responsible for a specific pattern and their "yelling" is based on how much of their pattern was detected by the feature demons. The more features the cognitive demons find that correspond to their pattern, the louder they "yell".

4. Decision demon, that listens to the "yelling" produced by the cognitive demons and selects the loudest one. The demon that gets selected becomes the most reliable conscious perception.

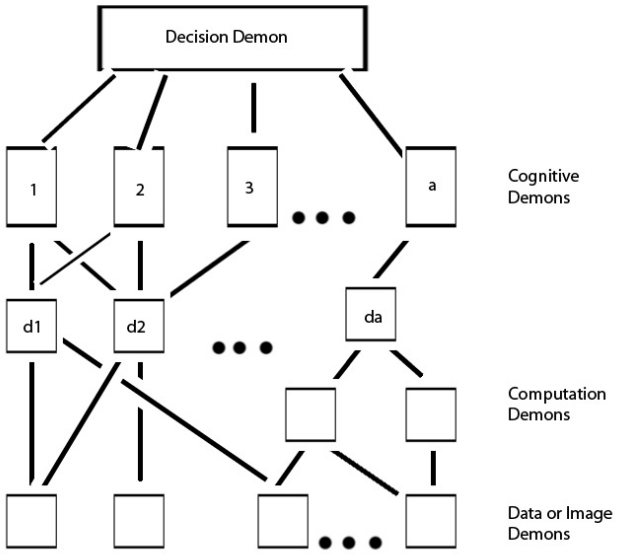

Figure 1.7: Pandemonium architecture highlighting its demons

Inevitably, looking at the figure above, the Pandemonium architecture is known to be quite similar to a modern neural network previously described in chapter 1.2.2, and indeed Selfridge can be considered a pioneer of AI and CV. However, although his model could already perform some simple tasks (such as the recognition of some geometric figures or letters of the alphabet), the major limitation of this architecture was the low computational power of the first digital computers on which it was installed. The development of computers would clearly have laid the foundations for a revival of the model, but the advent of NNs, which were much more versatile given their ability to adapt their neurons to new input data, and

without presenting the same fixed pattern for each different type of case, made this technology obsolete as a model-based solution. Also, Deep Networks are highly effective for the analysis of natural images and lend themselves very well to transfer learning.

Focusing on the structure of ML models designed for the different CV tasks that may be encountered, although basic NNs can be sufficient to solve simpler problems, these turn out to be unsuitable when the images to be analyzed are too articulated, presenting many details and colors, or with unclear shapes due to bad illumination conditions. Thus, modern DL models for Computer Vision keep presenting a dense network for feature extraction and classification, but with a series of special layers placed before it. These types of models are named Convolutional Neural Networks (CNNs), and they are fundamentals for CV applications due to their ability to recognize complex and peculiar patterns in an image. Moreover, this ability can be combined with other structures to solve tasks like Voice Recognition or Natural Language Processing, not related to CV.

The structure of these networks takes inspiration from the brain's Visual Cortex, in which, as reported in some studies conducted by David H. Hubel and Torsten Wiesel on cats[19][20] and monkeys[21], many neurons have a small local receptive field, meaning they react only to visual stimuli located in a limited region of the visual field. The receptive fields of different neurons may overlap, and together they tile the whole visual field. Moreover, the authors showed that some neurons react only to images of horizontal lines, while others react only to lines with different orientations (two neurons may have the same receptive field but react to different line orientations). They also noticed that some neurons have larger receptive fields and react to more complex patterns that are combinations of the lower-level patterns. These observations led to the idea that the higher-level neurons are based on the outputs of neighboring lower-level neurons. This robust architecture can

detect all sorts of complex patterns in any area of the visual field.

A truly similar approach in DL resides in Convolutional Layers: these layers apply some filter, or *kernels* (that represent the visual receptive fields of biological neurons), upon the input matrix to recalculate its values. The recalculation occurs by making a kernel slide along aligned series of pixels and taking the weighted sums of the overlapping elements of the two matrices. The obtained output matrices from these convolution operations are called *feature maps*.

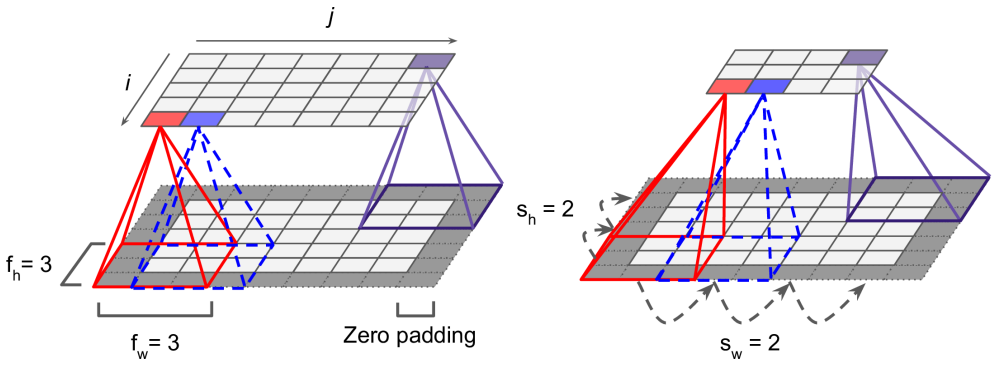

Figure 1.8: Images taken from [8] illustrating one convolution step. f_h and f_w are the dimensions of the kernel matrix, s_h and s_w are the strides.

Based on the main directions followed by the kernel matrix during convolutional operations, the Convolutional layers get distinguished in:

1. Conv1D layers, when the convolution occurs only in one spatial dimension of the input tensor.

2. Conv2D lavers, where convolution occurs in two spatial dimensions. This type of convolutional layer is particularly used when the input tensors represent grayscale images, which have only one color channel.

3. Conv3D layers, where convolution occurs in three spatial dimensions. This type of convolutional layer is particularly used when the input tensors represent colored images, which generally have three different color channels (RGB), or

videos.

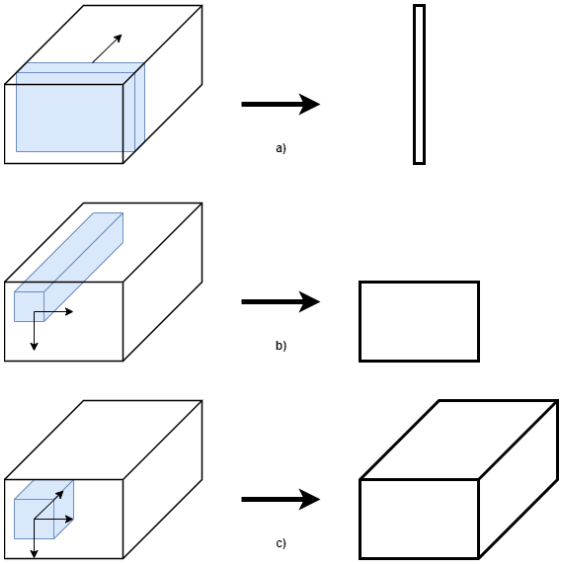

Figure 1.9: Image illustrating different convolution operations performed by Conv1D (a), Conv2D (b), and Conv3D (c) layers. The convolution operations performed by kernels following the highlighted movement directions along the sides of the input tensors on the left produce the output tensors on the right.

It can be noticed that the produced kernel dimensions are equal to $N - D$ dimensions of the input tensor at most, where N represents the number of size parameters of the input tensor and D the directions along which convolution occurs.

Formal mathematical computation of the Convolutional operation to calculate the value of a cell $S_{i,j}$ of the output matrix can be explained by the following equation:

$$S_{i,j} = (I * K)_{i,j} = \sum_{a=0}^{m-1} \sum_{b=0}^{n-1} I_{i+a,j+b} K_{a,b} \tag{1.4}$$

In this equation:

1. I and K are the Input matrix and the kernel

2. i and j are the coordinates of a general cell in the output matrix.

3. m and n are the rows and the columns of the kernel

To not obtain smaller output matrices, a frame of 0s can be applied around the input matrix; this operation is called *padding*. Two types of padding can be passed as a parameter in the function of the layer, which are "valid" padding and "same" padding. "Valid" padding means that no padding is added to the input, and the output size of the convolution operation is smaller than the input size. "Same" padding means that the amount of padding added to the input is chosen such that the output size of the convolution operation is the same as the input size. Moreover, the kernel could also slide more than one matrix cell of step at a time. This is called *stride* and can be set on every sliding direction singularly.

Having many filters can be an advantage because more patterns can be investigated, but like the biological neurons do when analyzing images to consider more complex patterns, the first ones must be combined together. This step can be realized in DL models by convolving the previously obtained feature maps to obtain new ones, but this approach inevitably ends up dramatically expanding the spatiality of the problem, increasing the number of parameters at each step and the computation effort. To overcome these problems, another layer must be inserted between two different Convolutional layers: a Pooling layer. Pooling layers "summarize" the pieces of information contained in the feature maps taking the most important ones. This downsampling procedure reduces the number of parameters and the computation effort. The working principle of these layers is quite the same as the Convolutional ones: a sliding kernel moves along the input matrix, but instead of operating a weighted sum of the overlapping values, it applies a rule on the input values in the kernel window to produce one output value. These rules define the type of Pooling Layer: so we can use a Max Pooling Layer when the output coincides with the highest value of the input matrix within the pooling region, or an Average Pooling Layer when the output is the average between the

values within the pooling region.

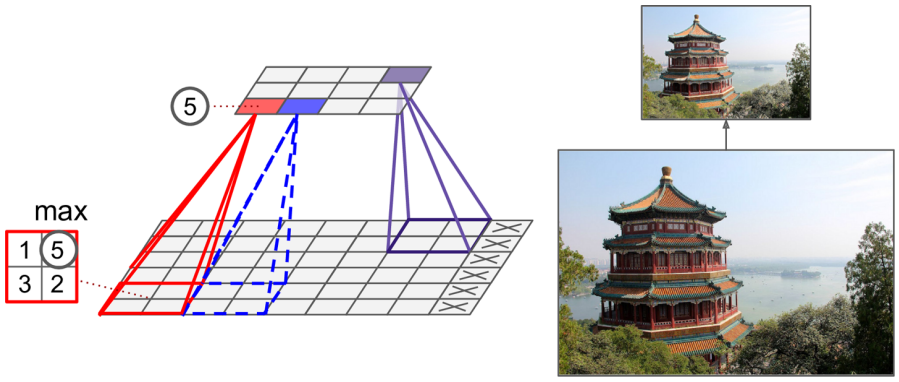

Figure 1.10: Images taken from [8] illustrating one pooling step operated by a MaxPooling layer.

Other than reducing computations, memory usage, and the number of parameters, a MaxPooling layer also introduces some level of invariance to small translations, a small amount of rotational invariance, and slight scale invariance. Such invariance (even if it is limited) can be useful in cases where the prediction should not depend on these details, such as in classification tasks.

Typical CNN architectures stack a few Convolutional layers (each one generally followed by a ReLU layer), then a Pooling layer, then another few Convolutional layers (+ReLU), then another Pooling layer, and so on. The image gets smaller and smaller as it progresses through the network, but it also typically gets deeper and deeper (i.e., with more feature maps), thanks to the Convolutional layers. At the top of the stack, a regular Feed Forward NN is added, composed of a few fully connected layers (+ReLUs), and the final layer outputs the prediction (e.g., a softmax layer that outputs estimated class probabilities). In this sense, the dense network recognizes the pattern found in the feature map obtained after the convolution of the input image.

The next step in order of difficulty lies in the analysis of video samples, which from this point of view can be considered as ordered sequences of images called

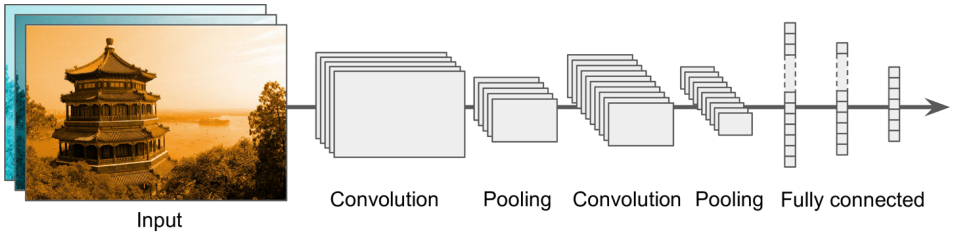

Figure 1.11: Images taken from [8] illustrating a typical CNN architecture.

frames. Both a simple DNN or a CNN can handle sequence processing, the former for very short sequences and the latter for very long ones. Recurrent Neural Networks (RNNs) are networks specialized in time series data of quite every length. An RNN is also composed of neurons like DNNs, called *recurrent neurons*. The configuration of these neurons is such that it receives at time t the input $x_{(t)}$ and its previous output $y_{(t)}$ when it is the only neuron in a network, while, when there are more neurons organized in a layer, they receive both the input vector X and the output vector y.

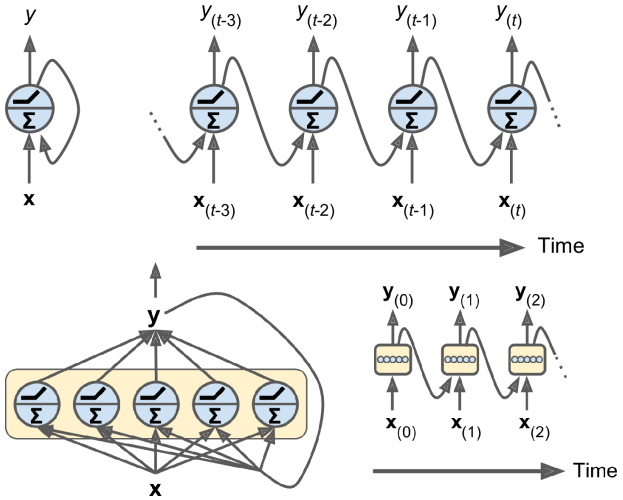

Figure 1.12: Images taken from [8] illustrating on the left the configurations of recurrent neurons (up) and a recurrent layer (down), and on the right their working principle through time.

The configurations and working principles of these types of systems recall the

ones of a memory. As a matter of fact, a part of a NN that preserves some state across time steps is called *memory cell*. In general, a cell's state at time step t, denoted h (that stands for "hidden"), is a function of some inputs at that time step and its state at the previous time step: $h_{(t)} = f(h_{(t-1)}, x_{(t)})$. Its output at time step t, denoted $y_{(t)}$, is also a function of the previous state and the current inputs.

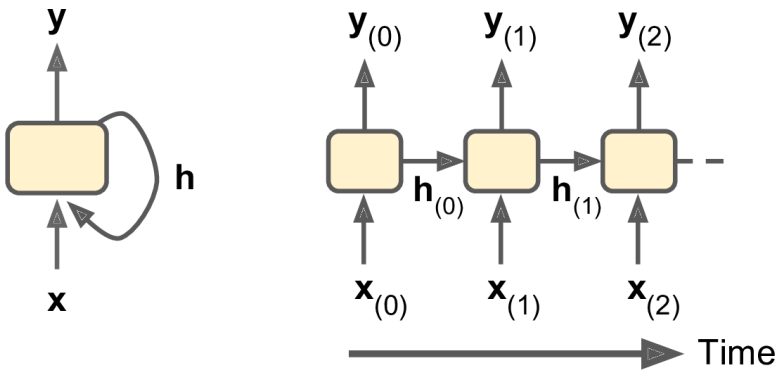

Figure 1.13: Images taken from [8] illustrating a general memory cell configuration and working principle

The single recurrent neuron or a recurrent layer are two simple examples of memory cells that perform better on a decade of time steps. However, due to the transformations that the data goes through when traversing an RNN, some piece of information is lost at each time step. After a while, the RNN's state virtually contains no trace of the first inputs. To tackle this problem, various types of cells with long-term memory have been introduced. The most popular of these long-term memory cells is the LSTM cell, proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber[22].

The LSTM layer uses an optimized implementation when running on a GPU, which is a perfect starting point because, using Keras and Tensorflow for the development of the model, data are already managed in tensor form. The LSTM cell looks exactly like a regular cell, except that its state is split into two vectors: h and c ("c" stands for "cell"), with h considered as the short-term state and c as

the long-term state.

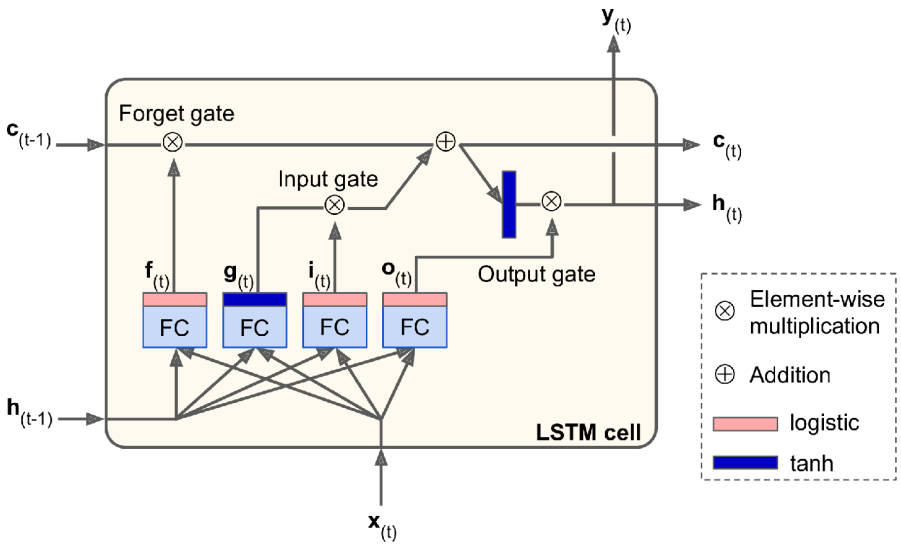

Figure 1.14: Images taken from [8] illustrating an LSTM cell.The forget gate (controlled by f) controls which parts of the long-term state should be erased, the input gate (controlled by i) controls which parts of g should be added to the long-term state, and, finally, the output gate (controlled by o) controls which parts of the long-term state should be read and output at this time step, both to h and to y.

The key idea is that the network can learn what to store in the long-term state, what to discard, and what to read from it. As the long-term state $c_{(t-1)}$ traverses the network from left to right, it first goes through a *forget gate*, dropping some memories, and then it adds some new memories via the addition operation (which adds the memories that were selected by an *input gate*). The result $c_{(t)}$ is sent straight out, without any further transformation. So, at each time step, some memories are dropped and some memories are added. Moreover, after the addition operation, the long-term state is copied and passed through the *tanh* function, and then the result is filtered by the output gate. This produces the short-term state $h_{(t)}$ (which is equal to the cell's output for this time step, $y_{(t)}$).

The current input vector $x_{(t)}$ and the previous short-term state $h_{(t-1)}$ are fed to

four different fully connected layers. They all serve a different purpose:

1. The main layer is the one that outputs g. It has the usual role of analyzing the current inputs x and the previous (short-term) state h. In a basic cell, there is nothing other than this layer, and its output goes straight out to y and h. In contrast, in an LSTM cell, this layer's output does not go straight out, but instead its most important parts are stored in the long-term state (and the rest is dropped).

2. The three other layers are gate controllers. Since they use the logistic activation function, the range of their output goes from 0 to 1. As can be seen, their outputs are fed to element-wise multiplication operations, so if they output 0s they close the gate, and if they output 1s they open it.

The following equation summarizes how to compute the cell's long-term state, its short-term state, and its output at each time step for a single instance (the equations for a whole mini-batch are very similar).

$$\begin{aligned} i_{(t)} &= \sigma(W_{xi}^T x_{(t)} + W_{hi}^T h_{(t1)} + b_i) \\ f_{(t)} &= \sigma(W_{xf}^T x_{(t)} + W_{hf}^T h_{(t1)} + b_f) \\ o_{(t)} &= \sigma(W_{xo}^T x_{(t)} + W_{ho}^T h_{(t1)} + b_o) \\ g_{(t)} &= tanh(W_{xg}^T x_{(t)} + W_{hg}^T h_{(t1)} + b_g) \\ c_{(t)} &= f_{(t)} \bigotimes c_{(t-1)} + i_{(t)} \bigotimes g_{(t)} \\ y_{(t)} &= h_{(t)} = o_{(t)} \bigotimes tanh(c_{(t)}) \end{aligned} \tag{1.5}$$

With:

1. W_{xi}, W_{xf}, W_{xo}, and W_{xg} are the weight matrices of each of the four layers for their connection to the input vector $x_{(t)}$.

2. W_{hi}, W_{hf}, W_{ho}, and W_{hg} are the weight matrices of each of the four layers for their connection to the previous short-term state $h_{(t-1)}$.

3. b_i, b_f, b_o, and b_g are the bias terms for each of the four layers. Note that TensorFlow initializes b_f to a vector full of 1s instead of 0s. This prevents forgetting everything at the beginning of training.

Artificial vision systems find numerous applications, from object recognition to biometrics, from smart surveillance (intelligent or cloud-based surveillance cameras to analyze recorded images and identify infringements) to movement tracking and diagnostic analysis in telemedicine, but also in the industrial and manufacturing fields, thanks to the possibility of being directly integrated into production lines and factory environments. Here are some examples of some fields of interest that are leveraging this technology:

1. Predictive maintenance: Computer Vision algorithms for monitoring industrial assets - mainly machinery - with a view to predictive maintenance (avoiding machine downtime by intervening in possible failures or malfunctions).

2. Product monitoring: systems for quality control and analysis of any product defects, to guarantee the highest level of customer satisfaction and limit any problems in the after-sales phase.

3. Safety in the workplace: systems for monitoring images of the plant, workers, and their actions, to identify any risk situations and/or accidents harmful to people or the environment.

Chapter 2

State of the Art

2.1 Hand Gesture Recognition problem

As reported by the scientific research on the matter and given the various case studies in which they are articulated, Machine Learning algorithms and, more specifically, DL models are demonstrated to be admirably able to classify different classes of objects given some type of input data (for example different species of dogs given images of them, different words given audio data, or categories of products received by a shop given a structured database). Considering a model trained on an acceptable amount of data, the simpler the input, the faster and easier it is for the algorithm the recognition. However, in the case of Dynamic Hand Gesture Recognition (DHGR), this assumption can be misleading: a hand gesture is an articulated motion, so for the recognition of a single class, many detailed data can be required by the model.

In this sense, the study conducted by the American psychologist Glenn David McNeil is illuminating to give a structure to the above problem. As reported in his paper "The Ontogenesis and Phylogenesis of Gestures: An Integrative Framework" [23], a dynamic hand gesture can be broken down into three phases: *preparation,*

stroke, and retraction. The preparation phase refers to the initial positioning of the hand before the gesture is made, the stroke phase refers to the movement of the hand during the gesture and the retraction phase refers to the final positioning of the hand after the gesture is completed. Generally, when executing a gesture, all these phases will be detected, and it would be futile and costly to try to focus only on the stroke phase, mistakenly considering it to be the most important phase for characterizing the entire dynamic gesture; in fact, the order in which these phases occur makes the gesture such, and this can only be classified in this sense if seen as a whole.

This can be noticed, for example, in the execution of two antagonistic gestures such as a "swipe left" and "swipe left", in which the preparation phase of one corresponds to the retraction phase of the other and vice versa, as shown in the figure below.

Figure 2.1: Image taken from [4] illustrating the comparison of 'swipe left' and 'swipe right' gesture sequences

If only the stroke phase were to be considered, however, another type of problem would be addressed, namely the classification of a static gesture, which can be used, for example, in sign language recognition or in other cases where only simpler and more direct concepts can be expressed. From the point of view of a robust

Human Machine Interaction, using static hand gestures is generally much more limiting than using dynamic ones; the latters are more natural and rich in pieces of information, so more concepts can be easily expressed.

Returning to the heart of the problem, this variety of expression has a cost in terms of data complexity: if for a static gesture much simpler data could be counted on, such as an image or coordinates of some points in space, for a dynamic gesture instead is required a series of data which always show the following characteristics:

1. Temporal correlation, because these are temporal sequences of values.

2. Spatial correlation, which can be absolute (coordinates of points in space) or relative (for example positions of fingertips concerning the wrist articulation).

In recent years, given the advantages reported in Chapter 1 of this thesis, it can be deduced that the best technology to use for the recognition and classification of DHGs is a DNN, whatever the nature of the input data they receive, even though the complexity of these requires the models to be more complex (more layers and more neurons) and, consequently, computationally costing.

In the next section of this Chapter, some different solutions coming from other scientific papers to tackle the problem are shown.

2.2 Related work

2.2.1 Recognition by Sensors Data

The first method presented involves the use of devices, hold by the hand or worn on it, mounting motion sensors on them. The main advantages of these devices are that sensors (principally accelerometers and electromagnetic sensors) are very sensitive to linear as well as angular movements, so they can produce precise data even in low-light or noisy environments. Furthermore, they can help reduce variability

in hand gestures, as they can provide consistent data even if the hand gesture is performed slightly differently each time. On the other hand, these devices should have small dimensions and weight to not be cumbersome and limit movement, have more limited computing capacity and memory than computers, and also have to perform parallel data acquisition, pre-processing of acquired data, and output prediction by the DNN, and these are actions that unfortunately can negatively affect inference.

To control the movement of a UAV with DHGs, *Changli Yu et al.* developed a wireless data glove integrating multiple sensors[24]. Gesture data acquisition, gesture recognition, and UAV control command transmission are all realized at the data glove end. The wireless data glove is composed of a control module based on an STM32 microprocessor, a sensing data acquisition module based on flexible sensors and inertial sensors, a rechargeable power supply module based on a lithium battery, and a wireless transmission module. In this article, a dual network gesture recognition method for UAV control that can be deployed in STM32 is proposed. The network includes the backpropagation network and bidirectional gated recurrent unit (Bi-GRU) network. The backpropagation network has a weak ability but low computational complexity to extract data features. This method is used for static gesture recognition with low difficulty. The GRU network is a variant of the LSTM network, which is also suitable for extracting the time dimension characteristics of the sensed sequence. However, compared with the LSTM network, the GRU network has lower computational complexity and smaller model weight. This method is utilized for the recognition of dynamic gestures that are difficult to recognize and are used as time series. The recognized gesture is converted into the corresponding UAV control command and sent to the UAV flight control terminal through wireless data transmission. Using two networks makes more efficient use of processor resources and improves the battery life of the system. Furthermore, the

recognition accuracy of each of the 15 gesture categories is higher than 95%, and the recognition time of a static gesture is 0.24 ms and that of a dynamic gesture is 155.15 ms.

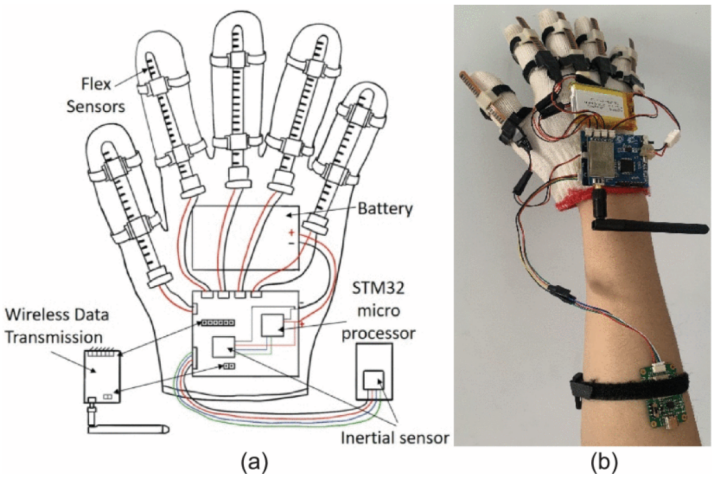

Figure 2.2: Image taken from paper[24] illustrating diagram of data glove (a) and real prototype (b).

Villani et al. proposed in [25] a more natural infrastructure-less solution to communicate with wheeled robots using a smartwatch or a sensorized wristband to obtain measurements of accelerations and angular velocities to recognize user's gestures and define velocity commands for the robots. The experimental setup consisted of a Samsung Gear S smartwatch, a Pioneer P3-AT mobile robot, and a computer with a ROS-implemented architecture on it to process the signals coming from the smartwatch. After processing signals, the computer sends an acknowledgment to the user imposing a short vibration to the smartwatch, providing haptic feedback after gesture recognition. The usability (considered as the time required to follow three paths, having different goals and setups, by the two piloting modalities) of the proposed approach was experimentally evaluated and compared to the use of a remote control device for the teleoperation of the robot, and the use of the smartwatch proved to be more intuitive and easy, allowing, in the 97%

of the performed trials, to complete the tasks in much less than the time taken by the latter approach.

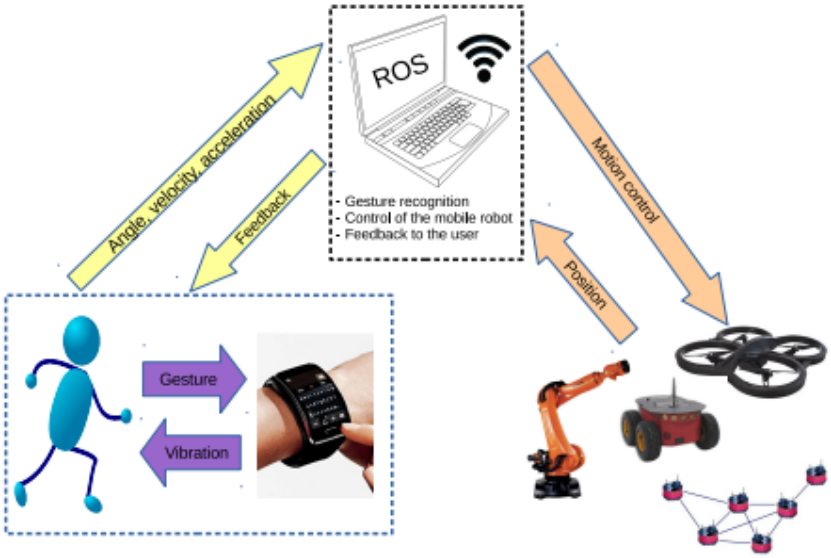

Figure 2.3: Image taken from paper[25] illustrating the procedure experimental framework of the project.

Finally, *Fungini et al.*, within the industrial research project named "Seamless", aimed to build a virtual environment where data collected by IoT sensors can be navigated through a gesture interface and virtual reality tools, presented in [26] a solution using a device that, differently from a smartwatch like used in the previously reported paper, performs the specific function of gesture controller, the Tactigon Skin[27] (or T-Skin). This device, developed by Next Industries, offers increased performance in gesture capture and motion recognition thanks to its AI algorithm, a Bluetooth Low Energy Interface, magnetic and pressure sensor to measure linear and angular motions, and four programmable keys for different command combinations. Collected data of gestures, after filtering, segmentation, and normalization pre-processing phases, have been passed to a classification net; during the projects, different experiments have been done on two different ML

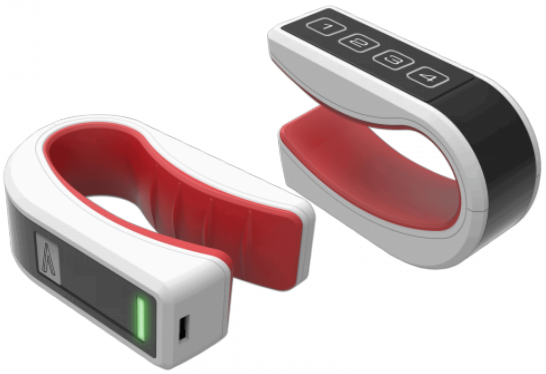

Figure 2.4: Tactigon Skin device.

models: a dynamic time warping (DTW) network and a simple Feed Forward network. The recognition software reached a performance ranging from 86% to 97% of correctly recognized gestures, depending on the single user and the gesture-performing conditions.

2.2.2 Recognition by Video Data

Solutions that implement classification NNs that receive video data as input are among the most popular in CV projects. The numerous variants of video CNNs generally present Convolutional layers for spatial classification of images and residual blocks, such as LSTMs, for temporal classification. For what concerns Convolutional layers, backbones (complex convolutional architectures made up of many layers that are placed before the Feed Forward network) are often used, and several papers present innovative architectures to improve the performances of the models more and more[28][29][30][31][32].

Purely video-type data are presented as dimension tensors (*frames, pixel_height,*

pixel_length, channels) for each sample, where:

1. *frames* are the images composing the video in every time instance of it;
2. *pixel_height* and *pixel_lenght* are the characteristic dimensions of every frame (generally, 256×256);
3. *channels* are the color channels that the camera adopts to represent the video; for example, RGB cameras have three channels, while greyscale cameras only have one.

One of the first uses of CNNs for video classification was presented by *Du Tran et al.* in [33], where the authors proposed the use of a 3D-CNN architecture, called C3D, to learn both spatial and temporal patterns of the input video tensors. They successfully demonstrated that the features of the architecture, combined with a linear classifier, can outperform or approach the best methods of that time on different video analysis benchmarks (Sport1M[34], UCF101[35], ASLAN[36], YUPPENN[37], and UMD[38].), for the tasks of Action Recognition, Action Similarity Labelling, and Scene and Object Recognition, in addition to being efficient, compact, and extremely simple to use.

The reference paper of this thesis project is the one by *Molchanov et al.* on behalf of NVIDIA called "Online Detection and Classification of Dynamic Hand Gestures with Recurrent 3D Convolutional Neural Networks"[4], where the video benchmark "Nvidia nvGesture", which will be detailed in the next chapter, is presented. The project is about the DHGR of command motions while driving a car, so in addition to creating the dataset, the team also developed a DL model to classify the different actions too, which is a Recurrent 3D Convolutional Neural Network, that consists of a deep 3D-CNN for spatiotemporal feature extraction, a recurrent layer for global temporal modeling, i.e. an LSTM layer, and a softmax layer for predicting class-conditional gesture probabilities. It performs excellent

results on the presented dataset and other benchmarks, both using singular sensors or cameras, combining them in different ways.

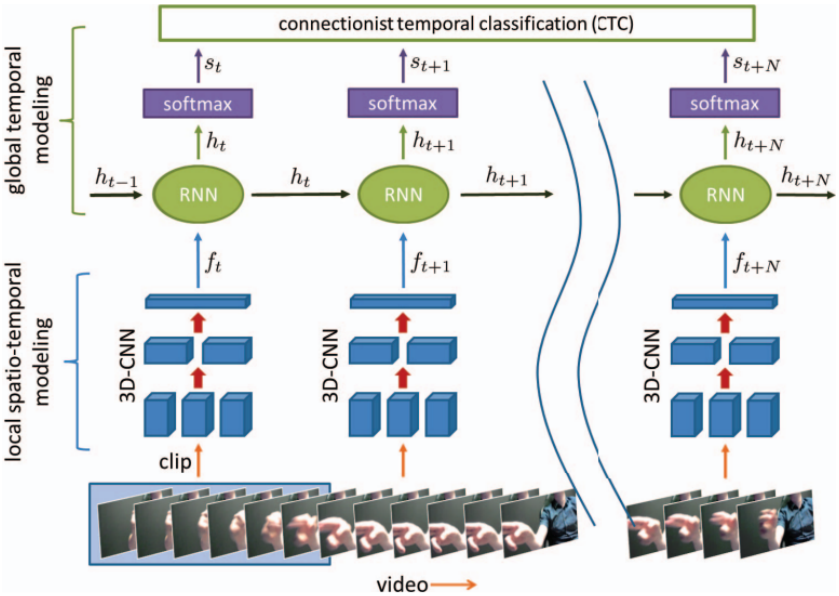

Figure 2.5: Image taken from [4] illustrating the model framework.

2.2.3 Recognition by Skeletal Data

Another interesting solution for solving the DHGR problem is represented by Skeletal Input Data, which are tensors presenting the *x,y, and z* coordinates of some landmarks of a human subject, for example the entire human body points, retina points, facial points, and, obviously, hands points. Reporting every object configuration for every time instance, an entire gesture can be represented with a tensor shape of (*time_instances, landmarks, coordinates*). Comparing a gesture represented by video data tensor of shape (*50 frames,256 pixel_height, 256 pixel_length,3 color channels*) and by a skeletal data tensor shape (*50 frames, 21 landmarks, 3 coordinates*), it is noticeable that the number of values of the latter corresponds to the 0.032% of the matter ones (3150 vs. 9830400). This strong reduction of values can correspond also to a less complex model and an increased

recognition speed during the inference phase, but also to the possible need for another model that can extrapolate skeletons out of video-recorded subjects.

Devineau et al. in [39] introduced a 3D HGR approach based on a CNN where sequences of hand skeletal joints' positions are processed by parallel convolutions; in particular, the architecture presents a multi-channel CNN with two feature extraction modules and a residual branch per channel, as shown in the figure below:

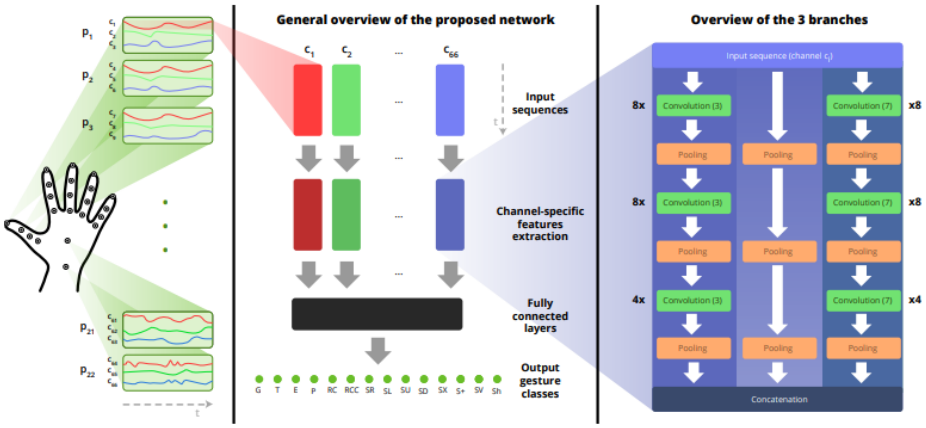

Figure 2.6: Image taken from [39] illustrating the model framework.

As explained by the team in the paper, the advantage of using two convolutional branches over a single one is that it allows the architecture to access different time resolutions of each signal; in addition, the use of residual connection for each signal allows the gradient to better backpropagate in the NN. It's interesting to notice that the so-obtained convolutional model seems to be competitive in terms of performance with other methods which use GRUs or LSTM layers. However, another interesting solution was proposed by *Xinghao et al.* in [40], presenting an alternative model based only on LSTM layers, but managing skeletal data in different ways before feeding the network with them. In particular, the model extracts and uses *Finger Motion Features* (by using a variational autoencoder), *Global Motion Features*, and the entire skeleton sequence data passing them to three parallel branches of LSTM layers placed before a fully connected network

used for classification.

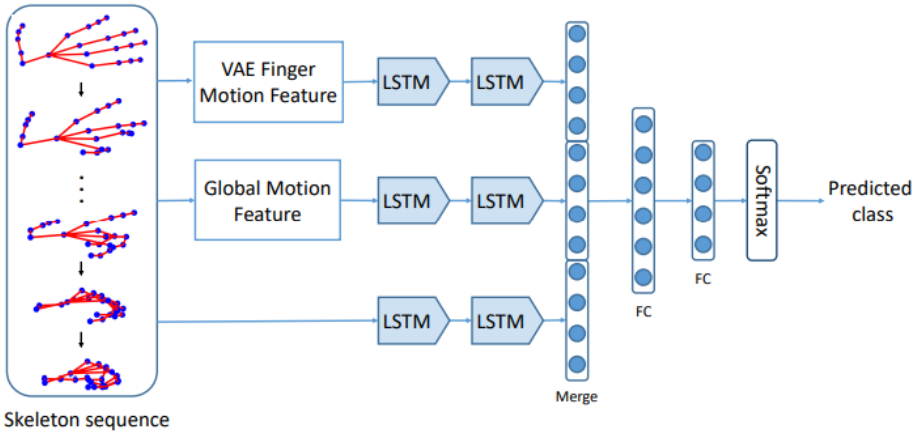

Figure 2.7: Image taken from [40] illustrating the model framework.

Both the presented methods reached an accuracy of around 91.3% after having been tested on DHG 14/28 dataset [41].

In the other two studies, solutions presenting both CNNs and LSTM layers were presented: in particular, *Nuñez et al.* in [42] proposed a model able to recognize human activities and hand gestures receiving 3D data sequences as input data, and a double-training strategy where first CNN then the entire model were trained, while *Kenneth Lai et al.* in [43] presented a CNN+RNN model that uses both skeletal and depth data in three different compared ways to make predictions, that are *feature-level fusion, score-level fusion, the decision-level fusion* of these data types. Finally, it can be also useful to report the study conducted by *Niels Schlüsener et al.* in [44] in which they implemented Google's model "Mediapipe"[5] to extract skeleton data by RGB videos to feed the network with, that is the approach used also in this thesis project.

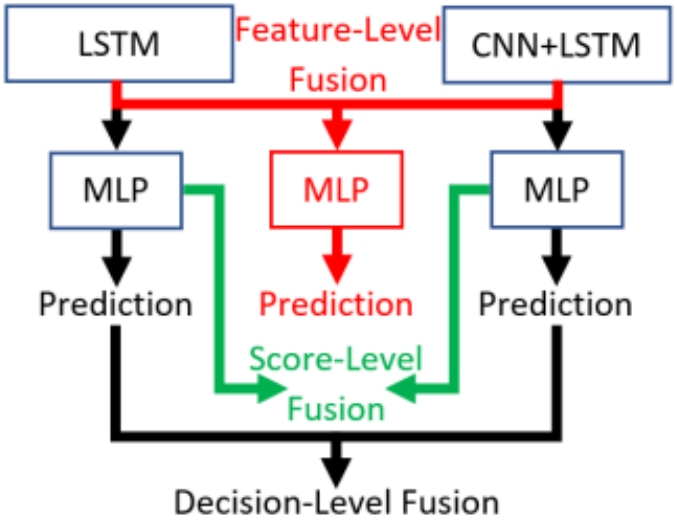

Figure 2.8: Image taken from [43] illustrating the model framework.

Chapter 3

Tools and Followed Methodology

This thesis project aims to design an AI architecture capable of recognizing DHGs acquired by a camera. This model will be installed on a home robot to raise its level of Human Machine Interaction, and will therefore be able to operate simultaneously with other models installed on it and focused on other purposes, such as the one responsible for recognizing voice commands. Therefore, I thought that the model to be designed should obtain high accuracy to best perform its task but, at the same time, it should be light and fast to be able to maintain high performances without execution lags, in particular when supported during the use by another model. For these purposes, I decided to develop a model that can receive as input a series of skeletal data, that correspond to the coordinates of landmarks of the recognized hand for every frame of the acquired video. As reported in subsection 2.2.3, this method offers performances in terms of accuracy comparable to the more classic one in which video data with frames corresponding to RGB images are used as input, but the number of values to be processed gets drastically reduced to

0.032% using the same shapes reported in the aforementioned passage of (50,21,3). In addition, by processing light data, also the model can be simpler than a video processing one, with a consequent computational effort decrease and execution speed increase.

3.1 Main Python Libraries

3.1.1 OpenCV

Figure 3.1: OpenCV logo[45]

As reported on the official website openCV.org[45], "OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in commercial products. Being an Apache 2 licensed product, OpenCV makes it easy for businesses to utilize and modify the code. The library has more than 2500 optimized algorithms, including a comprehensive set of classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects, produce 3D point clouds from stereo cameras, stitch images together to produce a high-resolution image of an entire scene, find similar images from an image database, remove red eyes from images taken using flash, follow eye movements, recognize

scenery and establish markers to overlay it with augmented reality, etc. OpenCV has more than 47 thousand people of user community and an estimated number of downloads exceeding 18 million. The library is used extensively by companies, research groups, and by governmental bodies."

As regards the use made of this library in the thesis project, through OpenCV it was possible to acquire data from the video cameras used during the tests or directly from the .mp4 files of the videos passed as input and to manipulate the frames obtained as NumPy arrays. Through the cv2 method, it is possible to initialize objects or execute functions capable of performing these actions, in particular:

1. *cv2.VideoCapture(0)*: Initializes a VideoCapture object able to read data received from "0" camera (or from a video file). Applying the method *read()* a boolean value asserting the acquisition or not of the image by the camera and the NumPy array of the frame are obtained.

2. *cv2.COLOR_RGB2BGR*: Initializes a constant value that represents a color space conversion code from RGB space to BGR one.

3. *cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)*: Function that applies the color conversion on a frame read by VideoCapture following a conversion rule like *cv2.COLOR_RGB2BGR* returning the modified frame.

4. cv2.flip(frame, 1): Function that flips an image frame horizontally (or vertically setting the second parameter to 0, or both vertical and horizontally setting a negative value).

5. *cv2.imshow('Camera', frame)*: Initializes a 'Camera' window representing the frame as an image. Noticing that this is a non-blocking function (which means that the program execution does not pause when the image is displayed), to keep the window open until the user closes it, *cv2.waitKey()* function can be used.

6. *cv2.destroyAllWindows()*: Function that closes all the windows created using cv2.imshow().

3.1.2 MediaPipe

[h]

Figure 3.2: MediaPipe logo[5][6][7]

As reported in [5], "MediaPipe is a framework for building pipelines to perform inference over arbitrary sensory data. With MediaPipe, a perception pipeline can be built as a graph of modular components, including model inference, media processing algorithms, data transformations, etc. Sensory data such as audio and video streams enter the graph, and perceived descriptions such as object localization and face landmark streams exit the graph. MediaPipe is designed for machine learning (ML) practitioners, including researchers, students, and software developers, who implement production-ready ML applications, publish code accompanying research work, and build technology prototypes. The main use case for MediaPipe is rapid prototyping of perception pipelines with inference models and other reusable components.". Some of the ML solutions proposed by MediaPipe are shown in the figure below.

The offered ML solutions are also cross-platform, working on Android and iOS mobile operative systems, and presenting libraries and functions written in Python and JavaScript programming languages.

For what concerns the purposes of this thesis project, the *hands* module (*mediapipe. solutions. mediapipe. python. solutions. hands*) was particularly used in combination with the OpenCV library for video data acquisition first

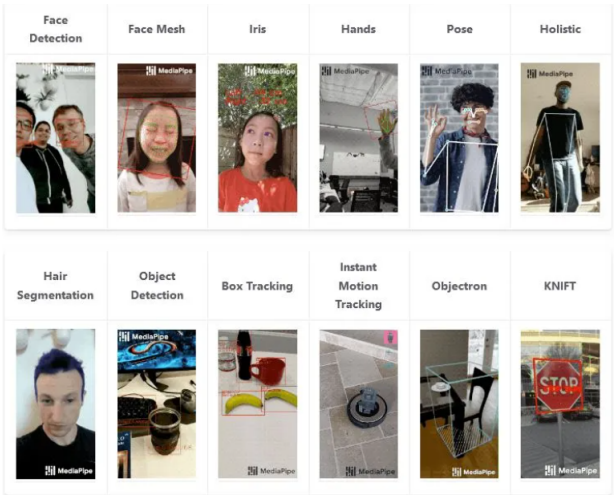

Figure 3.3: Image taken from [6] illustrating ML solutions offered by MediaPipe. "Hands", for hands recognition, is the one used in this project.

initializing the *Hands* Class, which processes an image and returns the hand landmarks and handedness (left v.s. right hand) of each detected hand, as *mp_hands.Hands(model_complexity=1, min_detection_confidence=0.5, min_tracking_confidence=0.2, max_num_hands=1)*, where:

1. *model_complexity* is a boolean value that takes account of model complexity ("0" for low or "1" for high); a high model complexity means high landmark accuracy as well as high inference latency.

2. *min_detection_confidence* that takes account of the minimum confidence value (to choose between 0.0 and 1.0) from the hand detection model for the detection to be considered successful.

3. *min_tracking_confidence* that takes account of the minimum confidence value (to choose between 0.0 and 1.0) from the landmark-tracking model for the hand landmarks to be considered tracked successfully, or otherwise hand detection will be invoked automatically on the next input image.

4. *max_num_hands* that takes account of how many hands to detect.

This Class is initialized using Python Keyword *with* inside a *while* loop in charge of taking frames by the video source and passing them to *hands.process(frame)* method that returns a *Hands* object that contains several attributes that provide information about the detected hands, such as: *multi_hand_landmarks*, that is a list of hand landmarks for each detected hand, and *multi_handedness*, that is a list of handedness information for each detected hand. It is important to highlight that the coordinates of the landmarks get directly normalized between 0 and 1 by the MediaPipe library considering the bottom left corner of the image as (0;0) and the right one (1;1); the z coordinate, in particular, does not represent the distance of a landmark from the camera but from the wrist landmark (i.e. the number 0), and is positive if a generic point is between camera and wrist, negative in the opposite case. In the figure below the landmarks of a hand are reported:

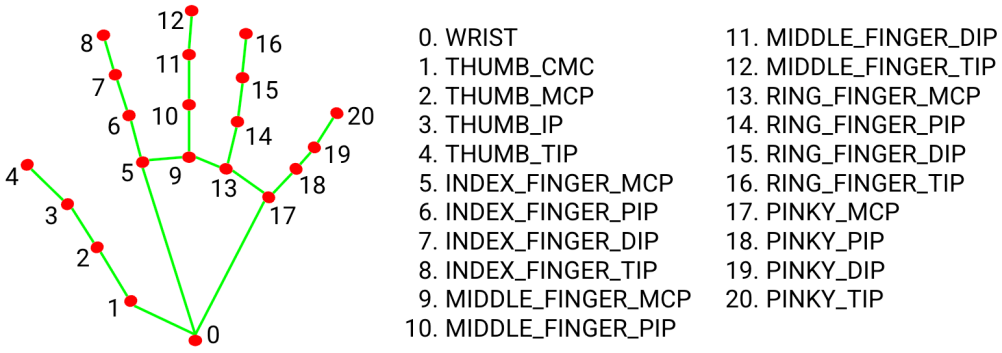

Figure 3.4: Image taken from [46] illustrating landmarks scheme and their enumeration.

3.2 Dataset Handling

The dataset chosen for the training of the developed model is the "Nvidia nvGesture" presented in [4]. This video dataset presents 25 different video classes of DHGs executed by 20 subjects from 3 to 5 times for each gesture and each recording device displaced in the room; subjects execute gestures sitting on a chair, in a

closed environment, recorded by different RGB-D cameras and Stereo-IR sensors displaced near them. The gestures in the dataset are:

1. Right swipe	11. Release	21. Clockwise Rotation
2. Left swipe	12. Trembling hand	22. Push with two fingers
3. Upward swipe	13. One/ Index up	23. "Hi"/ Open-close hand
4. Downward swipe	14. Two	24. "OK"/ Thumb up
5. Left-to-right rotation	15. Three	25. "OK"/ "O" in sign-language
6. Right-to-left rotation	16. Raising hand	
7. Up-to-down rotation	17. Lowering hand	
8. Down-to-up rotation	18. Push/ Stop	
9. Click	19. Approaching hand	
10. "Come here"	20. Counterclockwise rotation	

Table 3.1: Nvidia nvGesture Dataset HGs.

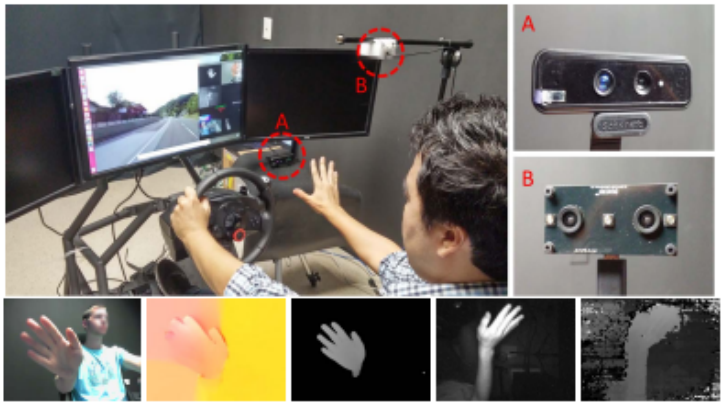

Figure 3.5: Image taken from [4] illustrating the environment for data collection. (Top) Driving simulator with a main monitor displaying simulated driving scenes and a user interface for prompting gestures, (A) a SoftKinetic depth camera (DS325) recording depth and RGB frames, and (B) a DUO 3D camera capturing stereo IR. Both sensors capture 320×240 pixels at 30 frames per second. (Bottom) Examples of each modality, from left: RGB, optical flow, depth, IR-left, and IR-disparity.

The reasons why this dataset was chosen are:

1. The wide variety of gesture classes and video available: in addition to simple movements, the robot could in fact receive commands via gestures to perform

more complex procedures, such as following the operator, going to the charging station, etc., so it could be useful to have more particular gestures than the simplest ones.

2. The presence of an RGB camera posed in front of the subjects: this can simulate the frontal camera with which the robot can observe the operator performing gestures.

3. The presence of other several sensors and cameras: these are extra data that could have been added if necessary.

4. This was a starting point for the definition of skeletal data extraction strategy through Python software: both model training and inference phases need skeletal data to be extracted from videos after hand recognition in every frame, so a similar code for these phases had been developed since the beginning of the project.

In order to create the training, validation, and test datasets, composed of skeletal data tensors, from the Nvidia nvGesture dataset, a Python script processing videos of the subjects performing the actions was written in order to perform the following actions:

1. Through the OpenCV library, videos are opened and displayed after flipping their frames horizontally to work in "selfie view" and changing the color space from RGB to BGR;

2. Upon a so-displayed video, the MediaPipe library applies a virtual skeleton upon the hand that executes the gesture. The skeleton is composed of 21 landmarks, each one characterized by three coordinates.

3. For every frame of video, in a secondary NumPy array of the video the coordinates of the landmarks are saved. When no hands were recognized by

MediaPipe, a matrix of shape (21, 3) full of 0s were saved.

4. These passages were repeated for every video of a class, forming the NumPy array of the class of shape (n_videos,max_n_frames,21 landmarks, 3 coordinates), and for every considered gesture class.

This just described is the general procedure for obtaining skeletal data from videos. However, some problems related to the creation of the dataset were polluting the data by acquiring useless gestures or movements that had nothing to do with the actual class, i.e. the presence of more than one person during the gesture acquisition, poor or exaggerated lighting in some videos, which made it difficult for MediaPipe to recognize hands, the hands visible on the steering wheel, which were recognized while remaining stationary (thus acquiring useless data), or the hands already present in the center of the video frame, which were simply retraction gestures cut from the immediately preceding videos, which was therefore followed by a dead time by staying on the steering wheel and the actual execution of the gesture of the video. To overcome these completely random drawbacks within the videos and to have the most reliable data possible, software filters have been implemented with the internal tools of the OpenCV and MediaPipe libraries, namely:

1. BGR color space instead of RGB: blue tonalities helped to get the skins more recognizable under every lighting condition and with every skin color.

2. Detection of only one hand in the frames: using the parameter *num_hands=1* in the "Hands" class, only the clearer hand recognized gets the skeleton applied on it, and so only its landmarks are recorded in the array.

3. *"Over the Steering wheel"* filter: From the starting time instance time instant of the video, if the x coordinates of the landmarks 8 (i.e. the index fingertip) do not cross the vertical line $x = 0.08$ (that is the vertical line averagely

tangent to the steering wheel) going along the right direction, the gesture detection does not take place.

4. *"On the Steering wheel"* filter: If the landmarks 8 and 20 (i.e. the index and pinky fingertips) cross the line $x = 0.08$ going left during the gesture detection and after 30 frames after the starting time instant of the video, staying between 0 and 0.08 for more than 20 consecutive frames, acquisition gets stopped.

5. *"False-start"* filter: If at the beginning of the video, the hand is detected right to the vertical line $x = 0.08$, the gesture detection does not take place, waiting for the hand to return to resting position to the left of the line (on the steering wheel); when the hand will cross this line going to the right, the gesture detection will start.

Furthermore, for some video samples the team released in a text file the beginning and the ending frame in which the gestures are executed; for these special samples, all the filters were avoided and only the frames of the highlighted interval were considered, while the other were directly transformed in blank matrices.

After some tests, and considering some basics motions for the robot and some service commands, I decided to select 7 types of gestures to be distinguished by the model, which are:

1. Class 9: Service Gesture.

2. Class 10: Motion Gesture, designed for forward movement towards the operator.

3. Class 18: Motion Gesture, designed for backward movement, moving away from the operator.

4. Class 20: Motion Gesture, designed for counterclockwise rotation.

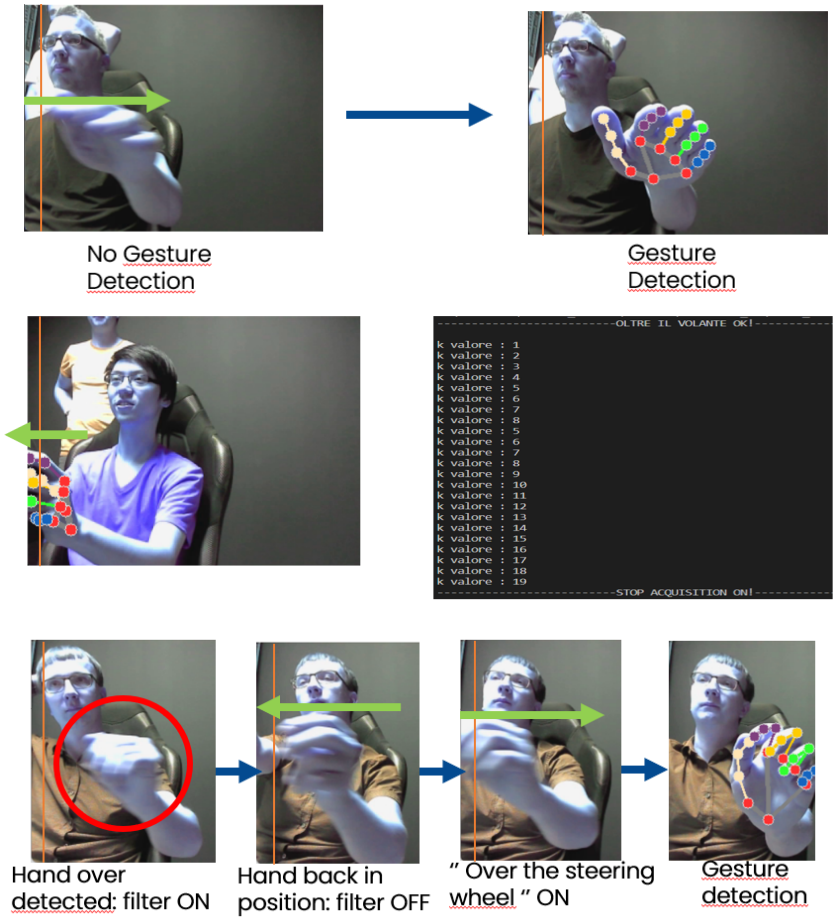

Figure 3.6: Software filters in action: "On the Steering wheel" filter (top), "Over the Steering wheel" filter (center), "False-start" filter (bottom), BGR color filter applied, one-hand recognition.

5. Class 21: Motion Gesture, designed for clockwise rotation.

6. Class 23: Service Gesture.

7. Class 24: Service Gesture.

The obtained NumPy arrays can be defined as groups of frame-by-frame reports of the coordinates of the noticed hand in every video. In order to consider only the gestures from the videos, the most prolonged sequences of not-blank frame matrices are extrapolated, with a tolerance of 10 blank frames in the sequence,

which will not be saved in the new gestures array. Then, in order to keep only the most consistent data, only the arrays presenting from 30 to 100 frames per gesture have been considered.

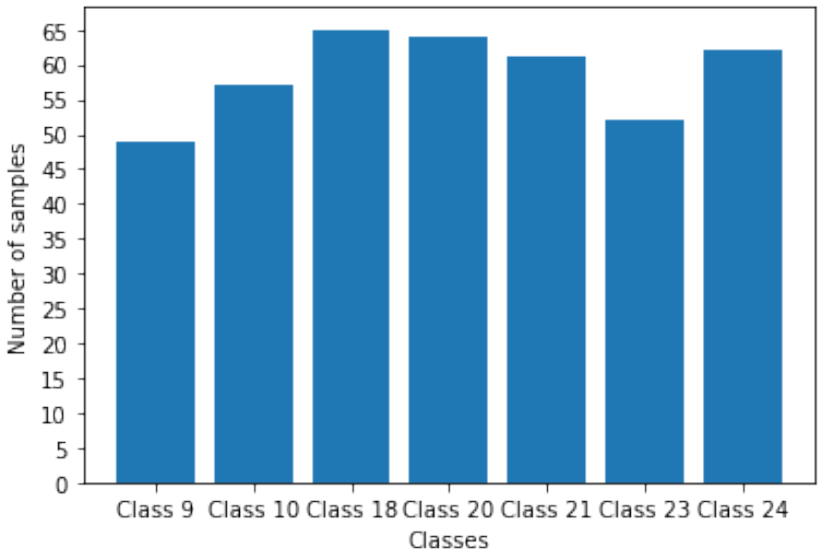

Figure 3.7: Distribution of samples per labels. The number of samples for Class 9 is 49, for Class 10 is 57, for Class 18 is 65, for Class 20 is 64, for Class 21 is 61, for Class 23 is 52, and for Class 24 is 62.

Considering the previously defined classes, the dataset is composed of a total of 410 samples. At this moment, in order to obtain NumPy arrays only containing gestures of the same shape, these had to be resampled. The resampling procedure consists in taking every sample and, applying linear interpolations to every function of landmark coordinates over time (i.e. over frames), obtaining new arrays of shape (50 resampled_frames, 21 landmarks, 3 coordinates). Finally, all the values of every array had been normalized between 0 and 1 values. This procedure might seem redundant because, as previously said as assumed by the team, MediaPipe already collects data normalized between these two values, but for an internal precision issue of the library, it is proved that data get collected between -0.2 and 1.2 values. Ending the pre-processing procedures, the 50 frames of a sample get grouped, by

using a reshaping function, all together in a bi-dimensional tensor of shape (50 frames, 63 features, 1 channel), where the features are the coordinates of every landmark in a single frame, which are now disposed all on the same row.

Finally, it is fundamental to highlight that, considering a front view of the observed scene, the recognized hand is always the one standing on the right of the video record. This will make the model able to recognize DHGR only if these are executed by the right hands. In order to overcome this situation, a simple solution can be offered by the MediaPipe model itself, which is able to recognize the handedness and return a boolean value to distinguish the two hands. Thus, if the action is performed by the left hand, a correction can be made immediately before the normalization process by flipping the coordinates of the landmarks along a vertical axis in all frames.

3.2.1 Data Augmentation

Given the limited amount of data samples, I decided to use some Data Augmentation techniques to enhance the performance of the model during training. Considering that the classical approaches only regard images, I had to develop some ad-hoc Data Augmentation functions from scratch. However, treating the frames representing the hands no longer as images but as sets of skeletal data offers the advantage of dealing with a set of 21 points in the space, so data augmentation functions can be seen as simple geometrical or vectorial transformations. These functions are:

1. Zoom-in: Scaling of the distance vectors connecting the landmarks from 1 to 20 with the wrist landmark, 0, about this point, which has the result to bring the points closer to the landmark 0 of a factor comprised between 30% and 50%.

$$\begin{cases} X' = K_v X + (1 - Kv) X_0 \\ K_v \in [0.5; 0.8] \end{cases} \tag{3.1}$$

Where X' is the scaled frame matrix of landmarks coordinates, X is the original frame matrix of landmarks coordinates, X_0 is the vector of the wrist landmark coordinates, and K_v is the scaling factor.

2. Zoom-out: Scaling of the distance vectors connecting the landmarks from 1 to 20 with the wrist landmark, 0, about this point, which has the result to move the points away from the landmark 0 of a factor comprised between 20% and 40%.

$$\begin{cases} X' = K_v X + (1 - Kv) X_0 \\ K_v \in [1.2; 1.4] \end{cases} \tag{3.2}$$

3. Gaussian noise addition: Addition of random values from a normal distribution, of mean μ equal to 0 and standard deviation σ equal to 0.001, to the data.

$$y = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.3}$$

All these functions were randomly applied (picking a value from a random generator function and comparing it to a decimal number representing a probability) through the function *map()* to some samples at every epoch during the training process.

3.3 Developed Model and training

3.3.1 Description and Scheme

Having defined the input data after merging the arrays of the desired gestures and mixing their values, the machine learning model to be able to operate the recognition of DHGs was also designed. The requirements that I wanted to establish to be satisfied were an accuracy that settled at a value higher than 90%, that it was light, and that it used less computational resources than normal video recognition

models. I would like to remind you that this model is not designed to be used alone, but instead to be combined with other models with whom can act simultaneously for the recognition of the commands to be given to a robot.

By the nature of the problem, i.e. a multi-class classification (i.e. of dataset elements equipped with exclusive labels) of DHGs, involving videos converted into arrays containing the modifications of the coordinates of a hand frame after frame, the model must take into account both the spatial configuration and the temporal succession of the frames. Therefore, given the advantages reported in the previous chapters, I opted for the use of a 2D Convolutional Recurrent Neural Network.

For what concerns the spatial analysis, given these explanations and the availability of relatively not many data, I decided to use three Convolutional layers successively accompanied by a Pooling layer each instead of using a pre-trained convolutional backbone. Thus, given the limited amount of data for every sample to be analyzed, after some tests, the best configuration I opted to use for Convolutional layers was the Conv2D layer, while, for Pooling operations, three MaxPooling2D layers were used. For what concerns the temporal analysis, instead, a simple LSTM layer placed between the convolutional block (Convolutional and Pooling layers) and the Feed Forward NN was used.

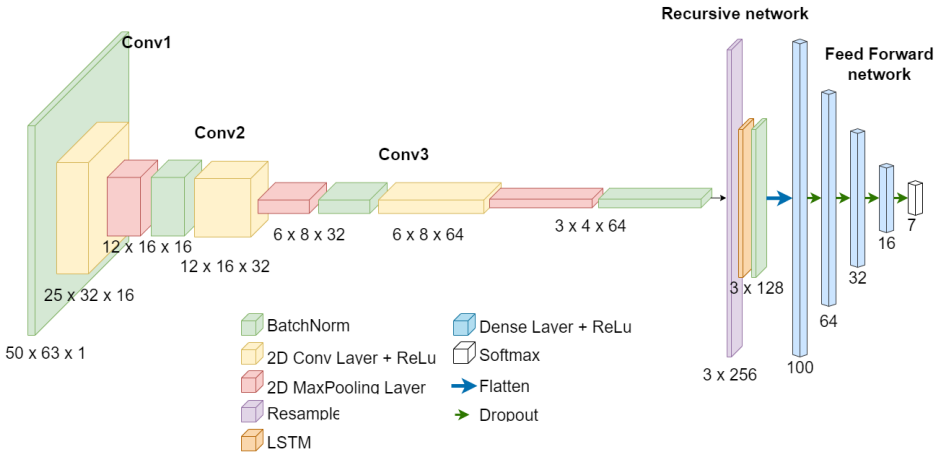

Figure 3.8: Model scheme.

3.3.2 Network operating principle and Training

Starting by formalizing the operations performed by the model, a tensor of area $C_t \in \mathbf{R}^{m\times(lk)}$ of $m = 50$ sequential resampled frames with $k = 3$ coordinates per $l = 21$ landmarks as values each at time t is defined. Each tensor is transformed into a feature representation f_t by a 2D-CNN, whose Conv2D layers have Relu as the activation function, and a recurrent LSTM network $\mathcal{F}$:

$$\mathcal{F} : \mathbf{R}^{m\times(lk)} \longrightarrow \mathbf{R}^{q}, \text{ where } f_t = \mathcal{F}(C_t) \tag{3.4}$$

by applying spatiotemporal and batch normalization filters to the clip. Then, this feature map is processed by a Flatten layer which transforms it into a 1D vector, which is finally given as input to the Feed Forward Network, whose Dense layers, except for the output one, have a Relu as activation function. Finally, the output of the entire model is given by the last layer of the latter network, i.e. a Softmax-activated Dense layer whose function is:

$$\sigma(\mathbf{X})_j = \frac{e^{X_j}}{\sum_{k=1}^{K} e^{X_k}} \tag{3.5}$$

where $\mathbf{x}$ is a vector of scores for each of the K classes, and $\sigma(\mathbf{z})_j$ is the probability assigned to class j by the softmax function and that is also used in online classification to output the gesture prediction.

The training of this DL model was carried out considering batches $\mathcal{X} = \{\mathcal{V}_0, \mathcal{V}_1, ..., \mathcal{V}_{P-1}\}$ of P samples $\mathcal{V}_\rangle$, in the form of 3D tensors containing frames with skeletal coordinates of every hand landmark, at a time. Each tensor consists of T frames, making $\mathcal{X}$ a set of $N{=}TP$ frames. Using predisposed functions, all of them had previously got the data augmentation functions applied by us *map()* functions, then had been cached in memory and shuffled 3 times in a buffer of dimension 1000, while, after the batch composition, they were also prefetched, reshaped and normalized, obtaining batches of P samples of dimension (F frames, L landmarks

coordinates). As the training cost function, it was considered the "Categorical Cross-Entropy" function

$$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{K} y_i \log\left(\sigma(\hat{y})_i\right), \tag{3.6}$$

where

1. $\hat{y} = (\hat{y}_1, \ldots, \hat{y}_K)$ is the predicted probability distribution over K classes;
2. $y = (y_1, \ldots, y_K)$ is the one-hot encoded true label;
3. $\sigma(\hat{y})_i = \frac{\exp(\hat{y}_i)}{\sum_{j=1}^{K} \exp(\hat{y}_j)}$ is the softmax function applied to the predicted probabilities.

To optimize the network parameters with respect to the loss functions, stochastic optimizer "Adam"[47] was used with default parameters and defining a Cosine Decay learning rate schedule function

$$\eta_t = \frac{1}{2} \cdot \eta_{max} \cdot \left(1 + \cos\left(\frac{\pi \cdot t}{T}\right)\right) \tag{3.7}$$

where

1. η_t is the learning rate at iteration t;
2. η_{max} is the maximum learning rate;
3. T is the total number of iterations;
4. t is the current iteration number.

This function was projected to reach the minimum value of 10^{-4}, starting from 10^{-3} at the first iteration, after 200 epochs, maintaining this value for the remaining 30 epochs.

Lastly, in order to reduce overfitting effects, regularization and dropout techniques were also used in the Feed Forward NN, in particular:

1. L2 Kernel Regularizers options in every dense layer, except for the output one, with values equal to 2% in the first three and 1% in the fourth;

2. Four Dropout layers, one between every couple of Dense layers, with values equal to 2% for the first three and 1% for the last one.

3.4 Testing phases

3.4.1 Trained model integration and Inference phase test

After achieving desired theoretical requirements from the training phase, the model was saved and downloaded in TFlite format following an ad-hoc procedure for the presence of the recurrent LSTM layer that does not permit the one suggested on Tensorflow website[48]. This procedure involves creating a tf_function as *run_model = tf.function(lambda x: model(x))* and using it with the *get_concrete_function* method as
concrete_func = run_model.get_concrete_function(tf.TensorSpec([1, 50, 63,1], model.inputs[0].dtype)) following suggestions on Tensorflow website at page[49] where are highlighted the benefit in using tf_functions. *concrete_func* is then used as *signatures* parameter when calling *model.save* function and finally opening and writing the saved model file after its conversion in TFlite format.

The downloaded model has then been integrated into a Python script to be utilized for the Inference phase using another procedure suggested on Tensorflow website page[50] in which is exposed how an *Interpreter* object gets created from the model for making predictions. Briefly, the script initiates camera recording and applies MediaPipe skeletons upon the hand of a recognized subject while performing a gesture, captures frame-by-frame the landmark coordinates of the skeleton, and creates an input tensor of that motion. The camera recording gets stopped when a minimum of 30 frames, in which the hand is recognized, are collected and,

subsequently, 20 consecutive frames without hands are detected (thus, the subject has to perform a gesture and then hide his hands from the camera angle), and when more than 100 valid frames get collected, in order to keep this maximum size, the next ones are appended while the first ones get canceled. The so-obtained tensor gets pre-processed by applying resampling, reshaping, and normalization transformations on it. Finally, the prediction is performed by passing this input tensor to the Interpreter using *set_tensor* method and invoking it with *invoke()* method, which returns as output an array containing the percentage probabilities of the predicted labels. It is important to highlight that the LSTM layer inside the model presents an active behavior during the inference phase, keeping in memory the collected states of the previously predicted gestures, which tampers all those that occur after the first; therefore, it is not possible to benefit from this layer's ability to analyze frames that follow one after another to extrapolate the temporal patterns of the tensor. In order to avoid this issue, the *reset_all_variables()* function gets called before passing the newly collected input data to the Interpreter. The Inference phase is then performed by running this script on a PC using a webcam. The obtained results are reported in the following chapter.

3.4.2 Experimental Setup and Wheeled Robot testing

The next test has been performed on a TurtleBot2[51] wheeled robot mounting an Intel RealSense d435 camera[52] and a NUC Intel i5 computer.

To make the robot move, the Python script used during the inference phase has been readjusted to obtain a ROS2 node in which is defined a Node Class and its functions, also following the procedures highlighted on ROS official website[53]. The function *spin(Node)* creates an infinite loop in which the node is initialized and performs the actions defined in a callback function cyclically, i.e. opening the

Figure 3.9: Figure illustrating the TurtleBot2 wheeled robot, the Intel RealSense D435 camera, and the NUC Intel i5 computer.

device 4 of the Intel RealSense d435, the RGB camera, collecting and pre-processes frames obtained by the camera topic which the node has subscribed to, and creating the input tensor of the model. After the gesture recognition, the highest predicted value is passed to a switch case selector; every case contains a different moving procedure realized with combinations of variables *linear* and *angular* that compose a "Twist" geometry message using the function *Twist()*, highlighting the translation or rotation axes and the linear or angular velocity values. The obtained message is published by the node on the *cmd_vel* topic, which communicates to the motors

the motion to perform as is also present a subscriber node that reads the message on the robot. The messages are published in a for loop at every defined time step for security reasons.

Since the purpose of the TurtleBot's movement in this experiment is purely demonstrative, the performed motions are rather simple and there is no control sensing of any kind to support the execution. In addition, since it was not possible to use a dedicated graphics card (or at least as high performance as that of a modern desktop or laptop PC) for the execution of the code and the computational procedures required for the model, these functions were fulfilled by the CPU of the NUC, resulting in reduced computational performance and dilation of the execution timelines; therefore, it was decided not to collect time metrics of any kind for this phase, merely recording the success of having moved the robot as the conclusion of the thesis project. The motion procedures are the following:

	Corresponding dataset class	**Movement procedure**
Gesture 1	9 (Click)	Slow BW mov. + Fast FW mov.
Gesture 2	10 ("Come here")	Slow FW movement
Gesture 3	18 (Push / Stop)	Slow BW movement
Gesture 4	20 (CW rotation)	Slow CCW rotation
Gesture 5	21 (CCW rotation)	Slow CW rotation
Gesture 6	23 ("Hello")	Slow CCW rot. + Slow CW rot.
Gesture 7	24 (Ok / Thumb up)	Prolonged CCW rotation

Table 3.2: Movement procedures for every gesture.

Every singular action is performed in 2 seconds, so procedures 1 and 6 are 4 seconds long. Furthermore, to explain what may seem a counter-intuitive choice, procedures 4 and 5 are such as to make the robot move in the opposite direction of rotation to that of the fingers, since if the subject wishes to see the front part of the TurtleBot (the one presenting the camera) go, for example, turning right, will move the index and middle fingers in that direction in clockwise rotation, starting from a vertical position with these pointing upwards. The robot, however, according

to its reference system, will have to rotate counterclockwise to see the camera go more and more to the left. The movement is therefore such to support the intuitive point of view of the operator.

Figure 3.10: Footage from two cameras of a rotational gesture. The upper figure shows the image taken by the Intel RealSense d435 camera, while the lower figure shows the shot of the action by the camera on my smartphone.

Chapter 4

Results

4.1 Model Performances and Experimental Results

Before designing the configuration of the latest model, several tests were conducted on others. As a neophyte on the subject of AI, the first important step of this project was the study of the necessary theoretical rudiments of Python and ML to independently carry the project out. In the beginning, I, therefore, used a simple MLP neural network and then I started to use 1D-CNNs, in which the first one able to give the first significant results was characterized by two sequences of Conv1D, Conv1D, and MaxPooling1D layers before a Feed Forward Network for output prediction. Considering that the performances didn't improve, I decided to use techniques to reduce overfitting, i.e. kernel regularization and the use of Dropout and Batch Normalization layers, obtaining the latest 1D-CNN configuration. The performances continued to be poor, therefore I thought that the problem could lie in the 1D Convolution type, so I updated the network with Conv2D layers to better analyze the inputs of dimension (50,63,1). The metrics utilized to measure

the final performances of every model are Accuracy, Recall, Precision, and F-Score, which are defined by the equations below:

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{True Positives} + \text{True Negatives} + \text{False Positives} + \text{False Negatives}} \tag{4.1}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{4.2}$$

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{4.3}$$

$$\text{F-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.4}$$

Where:

1. True Positives are the actual positive labels that were predicted as positive.
2. True Negatives are the actual negative labels that were predicted as negative.
3. False Positives are the actual negative labels that were predicted as positive.
4. False Negatives are the actual positive labels that were predicted as negative.

These cases can be highlighted in a graphical method using a so-called Confusion Matrix, considering the vertical axis as the axis of the actual values and the horizontal axis as the axis of the predicted values.

Returning to the model, having obtained a substantial improvement by also applying the previously described Data Augmentation techniques, I decided, in order to reach an acceptable accuracy value starting from the 2D configuration obtained at the end of this study and testing process, to act on the number of classes. The classes to be recognized were initially twelve (i.e. classes 1, 2, 5, 6, 10,

13, 14, 15, 18, 20, 21, and 24), and the obtained performances obtained by these models are reported in Table 4.1, but since the data available were not sufficient to offer many examples for each gesture and some of these were excessively redundant for the actions to be performed, I decided to reduce the number of classes to the seven considered and described in the previous chapter.

	Val. Accuracy	**Val. Loss**	**Recall**	**Precision**	**F-Score**
MLP	0.7143	1.6034	0.7143	0.7329	0.7235
First 1D-CNN	0.7532	1.0538	0.7532	0.7446	0.7489
Last 1D-CNN	0.7922	1.1454	0.7922	0.7882	0.7902
2D-CNN	0.8441	0.5030	0.8442	0.8252	0.8346
2D-CNN+DA	0.8701	0.4924	0.8701	0.8310	0.8501

Table 4.1: Performances and metrics of models using 12 classes.

As expected, first dividing the data only into training and validation datasets (corresponding to 80% and 20% of the total samples available), the performances of the models using this reduced number of gestures and the Data Augmentation techniques have significantly improved, in particular in the latest 2D configuration, but a last interesting suggestion resided in using also an LSTM layer which could take into account the succession of frames (i.e. rows of input tensors). The result was to obtain the configuration previously described in Chapter 3, the definitive one, which was able to perform at its best in terms of accuracy and loss of validation data (which are the parameters mainly taken into account to improve and tune the networks), but also in terms of Recall, Precision and F-Score metrics. Since, however, by dividing the available dataset into percentages equivalent to 70%, 20%, and 10% for the training, validation, and test data, in order to measure the metrics above, the performances were greatly worsened given the shortage of data for the model training, I decided to personally record 6 executions for each class of gestures and to process them in the same way as the videos belonging to the Nvidia dataset. I performed the recordings as if I were a hypothetical subject number 26, placing

myself at a distance from the camera comparable to the one kept by the Nvidia team and executing the gestures with the right hand starting from the left of the camera. The arrays containing the motion I recorded became the test dataset, while the others compounded the training and validation datasets of the model.

The obtained results are shown numerically in the following table, while the plots relating to the accuracy trend and the related Confusion Matrix are shown in the following images.

	Val. Accuracy	**Val. Loss**	**Recall**	**Precision**	**F-Score**
MLP	0.8568	0.7086	0.6905	0.6673	0.6787
First 1D-CNN	0.9031	0.5477	0.8333	0.8279	0.8306
Last 1D-CNN	0.8981	0.4707	0.9047	0.8950	0.8998
2D-CNN	0.9153	0.4344	0.9523	0.9517	0.9520
2D-RCNN	**0.9368**	**0.3040**	**0.9762**	**0.9758**	**0.9760**

Table 4.2: Performances and metrics of models using 7 classes.

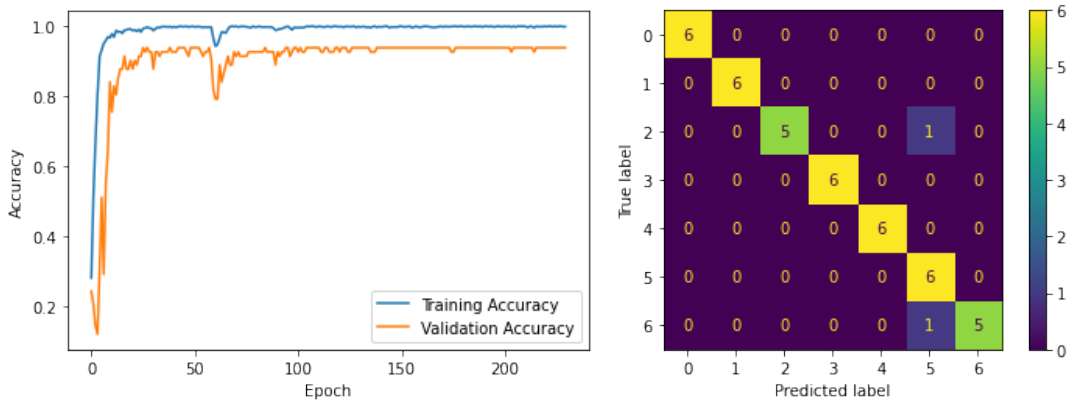

Figure 4.1: Validation and training accuracies (left) and the Confusion Matrix of the 2D-RCNN model (right).

4.2 Strenghts and Critical issues

Briefly recalling the desired characteristics that the model should have had at the end of the training, this should have been fast in recognizing gestures, light from

a computational point of view, and reliable in terms of accuracy. While the last characteristic is verified by the analysis of the previously exposed training metrics, the first and the second ones were estimated by conducting various tests in the inference phase from which the following results arose:

1. Mean Temporal Footprint: 5.8 ms. This measure has been obtained by measuring the difference between the time that an input tensor of shape (50,21,3) is obtained by camera acquisition of gesture and MediaPipe processing, and the time instant the model produces the output. This process involves pre-processing of input tensor phases of skeletal data conversion by MediaPipe, reshaping of data into (50,63,1) shape, normalization of the values, conversion of them to a float 32 format, and processing by the model.

2. Number of model parameters: 269071. This is a fantastic result when compared to those shared in [54], which shows how this characteristic is related to different video recognition models that use a classical approach of image acquisition as input. In particular, it can be seen that two MobileNet[55][56]-based models have the minimum amount of parameters among all the exposed models, with a value of 2.5 millions of parameters but in contrast to the accuracy of 65.5% at most. Furthermore, these models work under a frame rate of 5 FPS, while the model proposed in this thesis project works with data acquired at 30 FPS of camera record without any time lag during the execution of software running with it.

3. Input number of values: 3150. This value represents the 0.032% of an RGB input video of the same amount of frames and a dimension of 256 pixels $\times$ 256 pixels for every frame.

Looking at these data, I consider myself proudly satisfied with the obtained results and with the learned concepts. The targets I set for the model have been

achieved and, from this point of view, the model could represent a valid alternative to the more classical video classification models or at least could be considered an interesting starting point to study skeleton data networks in deep.

However, it is also important to highlight the critical points found in the use of this model and to comment on them:

1. The model cannot efficiently recognize gestures at a distance from the camera higher than 2 meters. This can be seen as a criticality of limited importance for what is the use to which this model is designed, i.e. the use of it on a collaborative home robot, which therefore only in some cases could be very distant from the operator of which it must recognize the gesture. In this sense, other models that benefit from other aspects than a visual input could still assume greater importance.

2. Close dependence on a second model like MediaPipe to derive skeletal data. The network, therefore, works by default already alongside another strictly necessary model. Therefore, the fact that it is light and does not require a great computational effort assumes even more importance, but it is also true that these characteristics must also be respected by the model that extrapolates the skeletal data. In the case of MediaPipe, these requirements are respected and it does not appear to be the bottleneck on the reception and processing speed of the data until the output is obtained, but it could affect the accuracy.

3. Often the output is influenced by the direction in which the gesture is performed and by the handedness. Although there were comparable quantities of samples for each gesture to be provided to the network for training and metrics testing, the model was trained with gestures performed only by right hands and presenting the preparation phase starting from the left of the video for then converging to the right for the stroke phase. The model, therefore, lacks a lot

in recognizing gestures performed with the left hand and which are carried out from right to left. However, these problems can be solved by adding further correction instructions in the pre-processing phase, like reflections with respect to a vertical axis (passing vertically through the wrist point or through the x=0.5 coordinate of the reference system) and rigid translations. These transformations were discarded during the Data Augmentation process as, unfortunately, they failed to improve the training performances, but, as the others used, they are geometrical, and can therefore act very quickly with the input tensor data. All this is aimed at restoring the registered movement to the standard conditions recognized by the model. The obligation of having to invert the labels relating to the "clockwise rotation" and "counterclockwise rotation" gestures will obviously have to be taken into account since the reflection transformation in turn reverses the directions of execution.

4. Absence of "No gesture" Class: Having not so many data didn't permit to easily integrate a too much general "No gesture" class, which would have led to the only result of lowering performances during training. Furthermore, the model naturally elaborates on every gesture when recognizing hands with the camera, and this can lead to undesired predictions and motions by the robot. In order to tackle this problem, a possible solution is to integrate into the software in which the model has been utilized a sort of attention trigger, a signal after which the model is authorized to acquire and process data. Some examples can be a specific vocal command, to be given by an external model, or a human recognition system. In this specific case, the OpenCV and MediaPipe libraries themselves offer some embedded functions to recognize the entire human body, or some parts of it, like the face. In this sense, a possible trigger could also be a smile recognition system or simply the fact that a human face is undoubtedly recognized, like the operator is explicitly

directing attention to the robot, which at this moment is called into question to interpret the incoming gesture.

4.3 Further comments and Personal impressions

Personally, I found this experimental thesis project very stimulating, interesting, and instructive. To pursue the prefixed targets immediately after giving shape to the idea of the problem to be solved, I thought that the skeleton data solution could offer satisfactory solutions, as indeed it was. However, the approach was far from simple as, unfortunately, I noticed that the attention that is placed in the literature on this type of solution, despite the benefits it can bring, is rather marginal compared to what are the more classic solutions of video classification, especially with regards to DHGs. Starting from the delivery, this is already quite pioneering, as it would have been much simpler if it had concerned static HGs, which could have required a common image classification model based on a sign language dataset, of which there are many available online. Video classification models exist and are also the subjects of many studies about them, but the available ones are pre-trained on mostly RGB video data, which does not make them a valid starting point for this type of problem, and therefore the model was also built from scratch. The real problem for me to solve in this thesis, however, was the crucial point of any machine learning project: the number of data. in fact, the datasets that include skeletal data for gesture recognition are scarce, both as regards the number of samples and as regards the availability of classes; so much so that the data of this project were obtained from the nvGesture dataset through a script in Python language developed by me that extrapolated the data of each class from every video shot by frontal RGB camera. Regarding the Nvidia dataset, I think it's useful to point out the two big problems that unfortunately affect it and which I had to handle:

1. The initial data extraction. Although the team provides two Python scripts for reading and extracting the videos, this procedure is unfortunately very smoky and tedious, since the files have to be understood and modified meticulously, apart from the fact that there are other datasets in which simple videos are provided in .zip format that can be extracted and ready to use. I do not agree with this choice, since in any case the dataset is made publicly usable.

2. The presence of disturbing elements, in particular, some of the participants of the dataset creation experience themselves, in a considerable amount of videos. The dataset remains of considerable interest as regards the great variety of hand actions filmed by different video cameras and performed by different subjects. A classic video classification model would greatly benefit in terms of data quantity and generalization by networks, and so it remains inexplicable that the team has made available repetitions of gestures in which the subjects involved in the filming were disturbed by other members of the cast while performing gestures, often invading the camera angle as well. These elements inevitably pollute the dataset, since the study situations of a person driving a vehicle who performs gestures behind the steering wheel are distorted (there can be no people standing and/or moving around a person sitting in the driver's seat), and other hands are seen and picked up as they move. This is all a pity for the merits that the dataset offers in particular for troubleshooting HGR tasks.

In retrospect, given the actual number of classes used and the fact that finally I had to register the test dataset myself, I think that, from a practical point of view, it would have been better to use a different dataset, perhaps with standing people; however, from an educational point of view, the manipulation of this dataset was enormously instructive due to the complexity of the tasks I had to solve to obtain the skeletal data I needed, not to mention that the acquired skills and the

code scripts written for these purposes were very useful for solving other problems encountered during the subsequent stages of model development.

Chapter 5

Conclusions and Future Work

This thesis project illustrates the implementation of a 2D-RCNN model capable of recognizing seven different dynamic hand gestures, captured by a video camera in a home environment, and its subsequent implementation on a robot so that it can perform as many motion procedures based on the recognized gestures. The model is fed by skeletal data of the hands obtained by the use of the MediaPipe framework by Google applied on each acquired video frame. The experience demonstrated the effective feasibility of a fast, reliable, and computationally light model, all so that it can interoperate with other models in parallel to further improve the level of Human Machine Interaction and the user experience of the robotic device using this technology. Ultimately, the setup used for the experimental tests on the robot and the movement procedures carried out for demonstration purposes only are also illustrated, highlighting the actual quality of the Human Machine Interaction level on which the future work of this project would be based. In particular, future work would certainly reside in trying to retrain the model by including many more

different data, perhaps obtaining skeletal data from other datasets or, even better, creating a new one, considering that the videos produced hardly exceeded the duration of 10 seconds. This could be useful to solve the criticalities residing in recognizing gestures from a distance higher than 2 meters and reduce the influence of the direction in which the gesture is performed and the handedness. It would also be interesting to introduce a "No gesture" type class and a trigger system, in order to refine the integration of the model in terms of interoperability between man and the machine. Lastly, some tests combining the model with other more complex ones for handling a home robot can be conducted in order to verify the performance obtained during their use in parallel.

Appendix A

Classical Computer Vision approach

It should also not be forgotten how much the technological progress of vision systems has affected the development of Computer Vision. From what has been deduced so far, man has always tried to design machinery capable of carrying out heavy work faster without tiring, however replicating his more complex cognitive and sensory structures to organize their logical structures and make the most of their potential; just think of the robotic arms, the neurons of artificial networks and vision systems, which want to replicate the functioning of human eyes. This is also a very interesting challenge since although computers have access to a calculation and image acquisition speed higher than that of humans, the latter has developed cognitive-brain characteristics that can count on the fact that data are acquired at a very high resolution and that the image association process is immediate and dictated by the experience and growth of the individual, as well as by the emotions that he feels every day in acquiring external stimuli that are much better fixed in memory. It is therefore simplistic to think of being able to recreate only the structure of the human visual system in a machine to facilitate its subsequent functions that lead to the recognition of the object. Over time, therefore, procedures and mathematical models, that could simplify the task of robots in the phases in which they are more deficient than humans and enhance their ultimate goals, have been developed.

According to these assumptions, a general hierarchical organization of actions, based on mathematical methods, can be assumed for computer vision tasks[57]:

1. Perception
2. Pre-processing
3. Segmentation

4. Description

5. Recognition

6. Interpretation

The perception phase is the process necessary to provide (digital) images to the computer. The main objects of attention in this phase are the subject to be filmed and the video camera used for acquiring the images: the lighting techniques of the subject, the camera calibration, and the vision model are therefore defined (i.e. the transformations that bring to the estimate of the coordinates of the acquired points in space) that it uses. After the image has been acquired by the camera, it must be digitized before being processed, and therefore the sampling operations are performed (which converts the continuous image signal into a discrete one, corresponding to a series of pixels, by taking measurements at regular intervals) and quantization (which discretizes the color signals of the image, reducing the number necessary to represent it). All this could also have the purpose of converting sharp and colored images into binary images, i.e. images organized in pixels that can assume only the white or black color; this practice is very commonly used in the manufactory industrial operation of computer vision.

The pre-processing phase deals with the filtering of the acquired images before the segmentation phase. The most common filtering techniques used in this phase are *neighborhood average filtering*, *median filtering*, and *average of multiple images*. In neighborhood average filtering, given an image f(x,y), a filtered image g(x,y) is generated where the intensity of each pixel is obtained through the average of the intensities of pixels of f in a predetermined neighborhood of (x,y).

$$g(x,y) = \frac{1}{P} \sum_{(m,n)\in S} f(m,n) \tag{A.1}$$

The neighborhood average tends to blur margins and sharp contours (useful for segmentation). This can mitigate this inconvenience by using the median value instead of the average one which tends to force pixels with very different intensities to be more similar to their neighboring ones, eliminating isolated transients; however, this requires a higher computational effort.

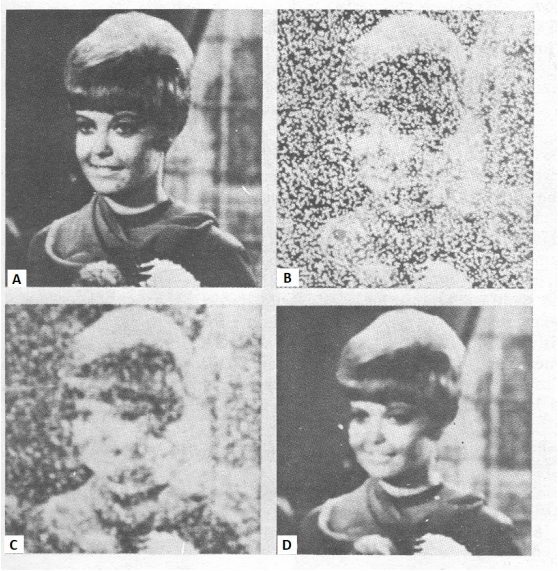

Figure A.1: A) Original image, B) Altered image, C) Neighborhood average, D) Median filtering

Furthermore, these two filters can be applied on input images using geometrical artificials called *stencils*, which are small bidimensional sets, whose coefficients are selected to reveal a given property or feature in an image (they work basically on the same principle of the kernel exposed in Appendix B). Stencils are also very useful for the next phase of preprocessing, edge detection. Considering for simplicity binary images, an edge is characterized by a sharp variation in the intensity of near pixels, that due to the sampling is generally ramp-shaped; to seek the presence of a monodimensional edge the first derivative of the intensity of the pixels is used, while to determine whether pixels lay on the bright or on the dark side of an edge the sign of the second derivative is used. Focusing on the whole bi-dimensional image, the principle still utilizes the gradient instead of the first derivative and the Laplacian operator instead of the second derivative, computed for *discrete* images.

The gradient is defined as:

$$G[f(x,y)] = \begin{pmatrix} G_x \\ G_y \end{pmatrix} = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} \tag{A.2}$$

Its magnitude is then used to detect edges:

$$G[f(x,y)] = \sqrt{G_x^2 + G_y^2} \tag{A.3}$$

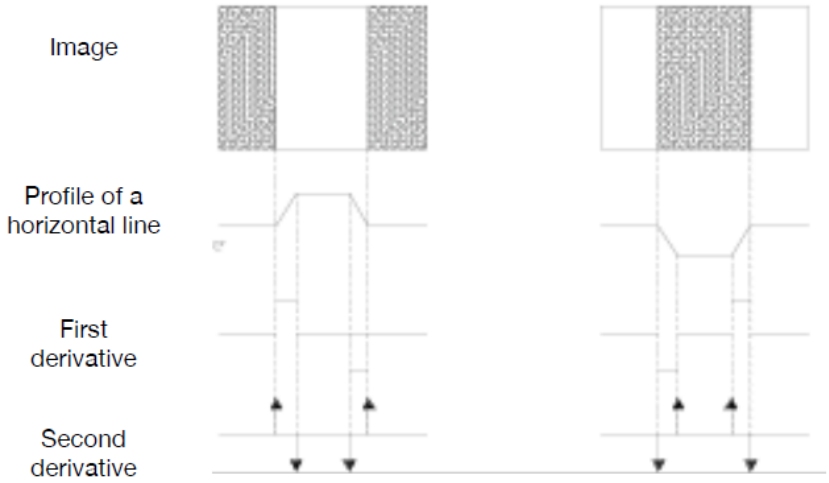

Figure A.2: Application of first and second derivatives on a bidimensional edge of an image (top) and stencils of Gradient/Sobel and Laplacian operators

For a digital image, it must be discretized as:

$$\begin{aligned} G_x &= \left(\frac{\partial f}{\partial x}\right) = f(x,y) - f(x-1,y) \\ G_y &= \left(\frac{\partial f}{\partial y}\right) = f(x,y) - f(x,y-1) \end{aligned} \tag{A.4}$$

However, this type of discretization is very sensitive to disturbances, thus an alternative solution can be found in the computation of Sobel operators, based on stencils theory[1]:

$$\begin{aligned} G_x = &[f(x+1,y-1) + 2f(x+1,y) + f(x+1,y+1)] + \\ &-[f(x-1,y-1) + 2f(x-1,y) + f(x-1,y+1)] \\ G_y = &[f(x-1,y+1) + 2f(x,y+1) + f(x+1,y+1)] + \\ &-[f(x-1,y-1) + 2f(x,y-1) + f(x+1,y-1)] \end{aligned} \tag{A.5}$$

[1]There exist many other approximations of the gradient (e.g. Prewitt, Roberts, etc.) that lead to different results depending on the starting image

-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

Figure A.3: Stencil of the Gradient or Sobel operator

Also, the Laplacian can be approximated in a similar fashion using stencils. It is mathematically defined as:

$$L[f(x,y)] = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} \tag{A.6}$$

For a discrete image it can be approximated with:

$$L[f(x,y)] = [f(x+1,y)+f(x-1,y)+f(x,y+1)+f(x,y-1)]-4f(x,y) \tag{A.7}$$

And realized with the stencil:

0	1	0
1	-4	1
0	1	0

Figure A.4: Stencil of the Laplacian of a discrete image.

Another efficient approach for edge detection is *Thresholding*. It is particularly used in industrial vision systems to detect objects, especially when a high processing speed is required. Supposing that the image comprises bright objects on a dark background, such that objects and background pixels have intensities that can be grouped into two dominant groups, a straightforward method to isolate objects from the background is to select a threshold T able to partition the two intensity; any point (x,y) such that f(x,y)>T is recognized as a point of the object, otherwise it is assumed to belong to the background. The critical point of this strategy is the choice of the threshold T. Summarizing, this approach produces a binary image

with the following rule:

$$g(x,y) = \begin{cases} 1 & if \quad f(x,y) > T \\ 0 & if \quad f(x,y) \leq T \end{cases} \tag{A.8}$$

Edge detection and Thresholding are then fundamentals for the next phase of the framework, the Segmentation phase. Segmentation is the process that subdivides the scene into its constituent parts, or objects, and it is one of the most important steps in Computer Vision because at this stage objects are extracted from a representation toward a subsequent identification and analysis. Segmentation is based on discontinuity and similarity concepts, satisfied by edge detection and thresholding procedures respectively for highlighting objects. In fact, ideally, gradient-based techniques should provide only pixels laying at the boundary between the object and the background, but practically, the selected pixel set seldom characterizes an edge because of disturbances and edge interruptions due to the non-uniformity of lightning and other disturbances. Edge detection algorithms are therefore followed by other procedures to connect and define edges, grouping pixels in significant sets representing the objects' contours. The first phase is called *local analysis*. In local analysis, the characteristics of pixels in a small neighborhood (e.g. 3X3 or 5X5) of each point (x,y) of an image that has already undergone a basic edge detection algorithm are analyzed; all the points labeled as similar are connected, forming a contour containing pixels that share common features. The two main criteria to assess similarity within neighborhoods the magnitude and the direction of the gradient vector, so two pixels (x, y) and (x', y') are considered similar if:

$$\begin{cases} |G[f(x,y)] - G[f(x',y')]| \leq T \\ |\theta - \theta'| \leq A \end{cases} \tag{A.9}$$

In this equation:

1. T is a threshold value;
2. $\theta = tan^{-1}\begin{pmatrix} G_y \\ G_x \end{pmatrix}$;
3. A is a threshold angle;

Successively, through global analysis, contour points get connected if they lay on a predefined curve. Considering n points laying on straight lines, all the possible lines that can connect two of them together are $\frac{n(n-1)}{2} \approx n^2$, with $\frac{n^2(n-1)}{2} \approx n^3$ comparisons to do. To avoid this computationally prohibitive approach, the Hough transform can be utilized. Introduced by Paul Hough in 1962 in the paper "Method

and Means for Recognizing Complex Patterns"[58], it consists in transforming the image space to a parameter space in which patterns could be represented as curves. This method was later popularized and widely used in computer vision and image processing applications.[2]. The motivation for using this transform is that, while in the Cartesian plane infinite lines pass through a point, depending on the values of parameters a and B, in the parameter space, a line corresponds to the set of (infinite) lines that pass through a point in the Cartesian space; thus, the intersection of two lines in the parameter space corresponds, in the Cartesian space, to the line that contains two given points.

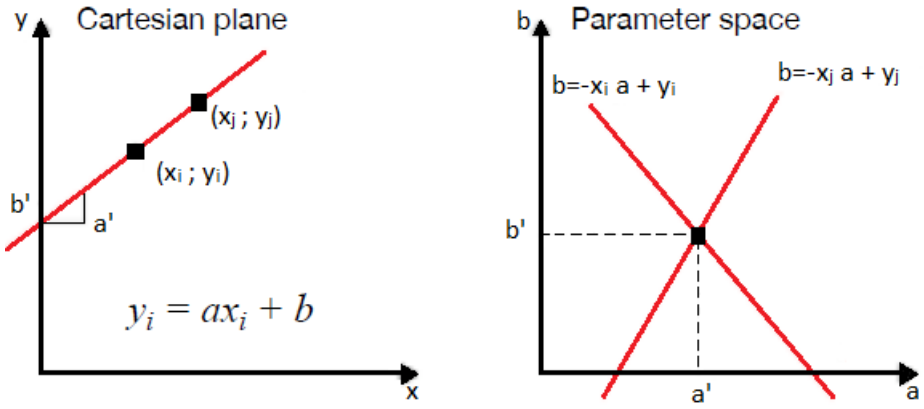

Figure A.5: Representing a line in Cartesian and parameter space.

Discretizing the parameter space in *accumulator cells* A(I,j) and defining a matrix with null initial entries, for each point (x_k, y_k) of the Cartesian plane, the parameter a is set equal to each of the values allowed by the discretization and the corresponding values $b = -x_k a + y_k$ are derived, approximating b to the closest value on the discretized b axis. If a_p gives as a solution b_p we increment by one unit the cell corresponding to the pair (a_p,b_p). A value M in the cell A(i,j) means that there are M points in the Cartesian plane that lay on the line $y = a_i x + b_j$. Dividing axis a into K elements, K corresponding values of B will be obtained, so, evaluating n points, nK operations have to be performed. A big problem resides in vertical lines detection, having a slope that tends to infinity, so the solution is to use polar coordinates to describe the lines; the difference is that the set of lines

[2]The Hough transform as it is universally used today was invented by Richard Duda and Peter Hart in 1972, who called it a "generalized Hough transform"[59] after the related 1962 patent of Paul Hough. The transform was then popularized in the computer vision community by Dana H. Ballard through a 1981 journal article titled "Generalizing the Hough transform to detect arbitrary shapes"[60].

in the Cartesian plane corresponds to a sinusoidal in the parameter space, but the approach to follow is exactly the same.

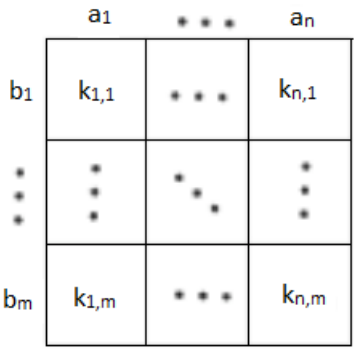

Figure A.6: Accumulator cells.

The description problem in the visual process is the extraction of features of an object. In theory, descriptors should be independent of the object dimensions, position, and orientation, and should contain sufficient information to distinguish unequivocally one object from another. The main description procedure is represented by the development of a chained code combined with shape normalization techniques. to derivation of the code consists in:

1. Selecting a grid with an adequate resolution, and if at least a given part of the cell (usually 50%) falls inside the contour, the cell is set to 1, otherwise to 0. Then, the cells set to 1 with the chained code get connected.

2. Alternatively, by dividing the contour into segments of the same length, the extremities of each segment are connected with a straight line, and the closest direction to those allowed is assigned to each line by the chained code.

For normalization, two main approaches are usually utilized:

1. With respect to the starting point, treating the code as a circular sequence and redefining the starting point such that the resulting sequence of numbers is the smallest possible integer.

2. With respect to rotations, using the prime difference of the code.

Previous normalizations are exact only if the contours do not vary with respect to rotations and scale changes. With characteristic shape functions, that are monodimensional functional descriptors of the shape of a contour. In practice, the same object digitized in two different orientations will generally present different shapes of its contour. Other hybrid descriptors are the *Characteristic shapes*

functions, which are monodimensional functional descriptors of the shape of the contour of an object. To obtain them, first a chained code is constructed with respect to the starting point, and then the characteristic shape of the code is computed; to distinguish between shapes, *function moments* can be used. The curves can be successively normalized between 0 and 1.

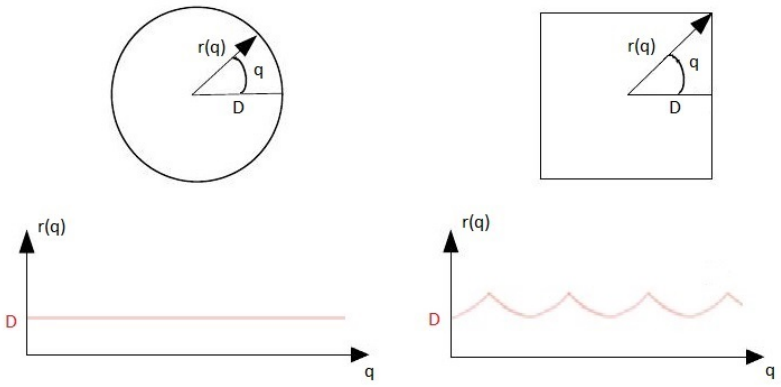

Figure A.7: Examples of characteristic shape functions.

Moreover, also descriptors based on Fourier transforms can be utilized; in fact, these are easy to normalize with respect to scale changes, rotations, starting point of the contour.

Finally, for the Recognition phase, the descriptors x_i of an object are collected in a so-called *pattern vector* X. Given M object classes, represented with $\omega_1, \omega_2, .., \omega_M$, the basic recognition problem using the decision theory is to identify M decision functions $d_1, d_2, .., d_M$ with the property that the following inequality holds for each pattern x^* belonging to class ω_i:

$$d_i(x^*) > d_j(x^*) \quad for \quad j = 1, 2, ..., M; j \neq i \tag{A.10}$$

An example is the recognition realized by computing Euclidean distance between descriptor and a *prototype vector* (or average) m_j corresponding to the j-th class of objects:

$$\begin{aligned} D_j(x^*) &= ||x^* - m_j|| \quad with \quad j = 1, 2, ..., M \\ m_j &= \frac{1}{N} \sum_{k=1}^{N} x_k \end{aligned} \tag{A.11}$$

x^* is assigned to class ω_j if the distance $D_j(x^*)$ is the shortest one. Recognition phase winks at the Machine Learning because of the tools used to concretize the effective object recognition, for example:

1. Probabilistic classifiers (e.g. Bayesian);
2. Supervised and unsupervised neural networks;
3. Clustering algorithms;
4. Fuzzy logic systems
5. Hybrid systems (neuro-fuzzy, fuzzy clustering, etc.)

Bibliography

[1] Michael Argyle. *The social psychology of everyday life*. Routledge, 2013 (cit. on p. 1).

[2] Albert Mehrabian. *Nonverbal communication*. Routledge, 2017 (cit. on p. 1).

[3] `https://www.statista.com/outlook/tmo/robotics/service-robotics/worldwide?currency=EUR` (cit. on p. 3).

[4] Pavlo Molchanov, Xiaodong Yang, Shalini Gupta, Kihwan Kim, Stephen Tyree, and Jan Kautz. «Online Detection and Classification of Dynamic Hand Gestures with Recurrent 3D Convolutional Neural Networks». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 4207–4215. DOI: `10.1109/CVPR.2016.456` (cit. on pp. 3, 35, 41, 42, 52, 53).

[5] Camillo Lugaresi et al. *MediaPipe: A Framework for Building Perception Pipelines*. 2019. DOI: `10.48550/ARXIV.1906.08172`. URL: `https://arxiv.org/abs/1906.08172` (cit. on pp. 3, 44, 50).

[6] `https://mediapipe.dev/` (cit. on pp. 3, 50, 51).

[7] `https://developers.google.com/mediapipe` (cit. on pp. 3, 50).

[8] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, 2017. ISBN: 978-1491962299 (cit. on pp. 4, 11, 24, 27–30).

[9] Bernard F. King. «Guest Editorial: Discovery and Artificial Intelligence». In: *American Journal of Roentgenology* 209.6 (2017). PMID: 29161146, pp. 1189–1190. DOI: `10.2214/AJR.17.19178`. eprint: `https://doi.org/10.2214/AJR.17.19178`. URL: `https://doi.org/10.2214/AJR.17.19178` (cit. on p. 5).

[10] F. Pedregosa et al. «Scikit-learn: Machine Learning in Python». In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 7).

[11] Frederic B. Fitch. «Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. Bulletin of mathematical biophysics, vol. 5 (1943), pp. 115–133.» In: *The Journal of Symbolic Logic* 9.2 (1944), pp. 49–50. DOI: `10.2307/2268029` (cit. on p. 8).

[12] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 2017. ISBN: 0262534770 (cit. on p. 10).

[13] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. «Learning representations by back-propagating errors». In: *Nature* 323 (1986), pp. 533–536 (cit. on p. 11).

[14] `https://keras.io/guides/functional_api` (cit. on p. 15).

[15] `https://towardsai.net/p/l/underfitting-and-overfitting-with-python-examples` (cit. on p. 19).

[16] `https://anolytics.home.blog/2019/11/12/difference-between-semantic-segmentation-and-object-detection/` (cit. on p. 20).

[17] James Hendler. «Oliver G. Selfridge (1926-2008)». In: *IEEE Intelligent Systems* 24 (Jan. 2009), pp. 12–13. DOI: `10.1109/MIS.2009.14` (cit. on p. 21).

[18] P.H. Lindsay and D.A. Norman. *Human Information Processing: An Introduction to Psychology*. Academic Press, 1977. ISBN: 9780124509603. URL: `https://books.google.it/books?id=6d9OAAAAMAAJ` (cit. on p. 21).

[19] D. H. Hubel. «Single unit activity in striate cortex of unrestrained cats». eng. In: *The Journal of physiology* 147.2 (1959), pp. 226–238. ISSN: 0022-3751 (cit. on p. 23).

[20] D. H. Hubel and T. N. Wiesel. «Receptive fields of single neurones in the cat's striate cortex». eng. In: *The Journal of physiology* 148.3 (1959), pp. 574–591. ISSN: 0022-3751 (cit. on p. 23).

[21] D. H. Hubel and T. N. Wiesel. «Receptive fields and functional architecture of monkey striate cortex». eng. In: *The Journal of physiology* 195.1 (1968), pp. 215–243. ISSN: 0022-3751 (cit. on p. 23).

[22] Sepp Hochreiter and Jürgen Schmidhuber. «Long Short-term Memory». In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: `10.1162/neco.1997.9.8.1735` (cit. on p. 29).

[23] David McNeill. «The Ontogenesis and Phylogenesis of Gestures: An Integrative Framework». In: *Semiotica* 89.1-2 (1992), pp. 187–228 (cit. on p. 34).

[24] Changli Yu, Shurui Fan, Yang Liu, and Yibo Shu. «End-Side Gesture Recognition Method for UAV Control». In: *IEEE Sensors Journal* 22.24 (2022), pp. 24526–24540. DOI: `10.1109/JSEN.2022.3218829` (cit. on pp. 37, 38).

[25] Valeria Villani, Lorenzo Sabattini, Giuseppe Riggio, Alessio Levratti, Cristian Secchi, and Cesare Fantuzzi. «Interacting With a Mobile Robot with a Natural Infrastructure-Less Interface». In: *IFAC-PapersOnLine* 50.1 (2017). 20th IFAC World Congress, pp. 12753–12758. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2017.08.1829. URL: https://www.sciencedirect.com/science/article/pii/S2405896317324527 (cit. on pp. 38, 39).

[26] Maria Fugini and Jacopo Finocchi. «Gesture Recognition in an IoT environment: a Machine Learning-based Prototype». In: May 2021 (cit. on p. 39).

[27] https://www.thetactigon.com/ (cit. on p. 39).

[28] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. «Imagenet: A large-scale hierarchical image database». In: *2009 IEEE conference on computer vision and pattern recognition* (2009), pp. 248–255 (cit. on p. 40).

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «ImageNet classification with deep convolutional neural networks». In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105 (cit. on p. 40).

[30] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on p. 40).

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep residual learning for image recognition». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cit. on p. 40).

[32] Joao Carreira and Andrew Zisserman. «Quo vadis, action recognition? a new model and the kinetics dataset». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 4724–4733 (cit. on p. 40).

[33] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. «Learning spatiotemporal features with 3d convolutional networks». In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 4489–4497 (cit. on p. 41).

[34] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. «Large-scale video classification with convolutional neural networks». In: *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition* (2014), pp. 1725–1732 (cit. on p. 41).

[35] Khurram Soomro, Amir Zamir, and Mubarak Shah. «UCF101: A dataset of 101 human actions classes from videos in the wild». In: *arXiv preprint arXiv:1212.0402* (2012) (cit. on p. 41).

[36] Gil-Jin Jang, Sung-Chan Jung, Kang-Hee Lee, and Seong-Whan Lee. «ASLAN: Attention-based spatio-temporal deep learning approach for human action recognition». In: *IEEE Access* 7 (2019), pp. 116148–116157 (cit. on p. 41).

[37] Sherin A Fathima, Swathi Jayaraman, and Arijit Ray. «YUPPENN: A new benchmark dataset for multimodal sentiment analysis in Indian languages». In: *arXiv preprint arXiv:1809.00689* (2018) (cit. on p. 41).

[38] An-An Liu, Chen Li, Guanbin Li, and Cewu Lu. «UMD: A Large-scale Video Benchmark for Staged Human-Object Interaction». In: *arXiv preprint arXiv:1807.08854* (2018) (cit. on p. 41).

[39] Guillaume Devineau, Fabien Moutarde, Wang Xi, and Jie Yang. «Deep Learning for Hand Gesture Recognition on Skeletal Data». In: *2018 13th IEEE International Conference on Automatic Face Gesture Recognition (FG 2018)*. 2018, pp. 106–113. DOI: 10.1109/FG.2018.00025 (cit. on p. 43).

[40] Xinghao Chen, Guijin Wang, Hengkai Guo, Cairong Zhang, Hang Wang, and Li Zhang. «MFA-Net: Motion Feature Augmented Network for Dynamic Hand Gesture Recognition from Skeletal Data». In: *Sensors* 19.2 (2019). ISSN: 1424-8220. DOI: 10.3390/s19020239. URL: https://www.mdpi.com/1424-8220/19/2/239 (cit. on pp. 43, 44).

[41] Quentin De Smedt, Hazem Wannous, and Jean-Philippe Vandeborre. «Skeleton-Based Dynamic Hand Gesture Recognition». In: *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2016, pp. 1206–1214. DOI: 10.1109/CVPRW.2016.153 (cit. on p. 44).

[42] Juan C. Núñez, Raúl Cabido, Juan J. Pantrigo, Antonio S. Montemayor, and José F. Vélez. «Convolutional Neural Networks and Long Short-Term Memory for skeleton-based human activity and hand gesture recognition». In: *Pattern Recognition* 76 (2018), pp. 80–94. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2017.10.033. URL: https://www.sciencedirect.com/science/article/pii/S0031320317304405 (cit. on p. 44).

[43] Kenneth Lai and Svetlana N. Yanushkevich. «CNN+RNN Depth and Skeleton based Dynamic Hand Gesture Recognition». In: *2018 24th International Conference on Pattern Recognition (ICPR)*. 2018, pp. 3451–3456. DOI: 10.1109/ICPR.2018.8545718 (cit. on pp. 44, 45).

[44] Niels Schlüsener and Michael Bücker. *Fast Learning of Dynamic Hand Gesture Recognition with Few-Shot Learning Models*. 2022. DOI: 10.48550/ARXIV.2212.08363. URL: https://arxiv.org/abs/2212.08363 (cit. on p. 44).

[45] https://opencv.org/ (cit. on p. 48).

[46] https://google.github.io/mediapipe/solutions/hands.html (cit. on p. 52).

[47] Diederik P Kingma and Jimmy Ba. «Adam: A Method for Stochastic Optimization». In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 63).

[48] https://www.tensorflow.org/tutorials/keras/save_and_load?hl=it (cit. on p. 64).

[49] https://www.tensorflow.org/guide/function (cit. on p. 64).

[50] https://www.tensorflow.org/api_docs/python/tf/lite/Interpreter (cit. on p. 64).

[51] https://www.turtlebot.com/about/ (cit. on p. 65).

[52] https://www.intelrealsense.com/depth-camera-d435/ (cit. on p. 65).

[53] https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html (cit. on p. 65).

[54] Dan Kondratyuk, Liangzhe Yuan, Yandong Li, Li Zhang, Mingxing Tan, Matthew Brown, and Boqing Gong. *MoViNets: Mobile Video Networks for Efficient Video Recognition*. 2021. arXiv: 2103.11511 [cs.CV] (cit. on p. 74).

[55] Ji Lin, Chuang Gan, and Song Han. *TSM: Temporal Shift Module for Efficient Video Understanding*. 2019. arXiv: 1811.08383 [cs.CV] (cit. on p. 74).

[56] Andrew Howard et al. *Searching for MobileNetV3*. 2019. arXiv: 1905.02244 [cs.CV] (cit. on p. 74).

[57] *Didactic material of the "Automation and planning for production systems" course by Professor Rizzo* (cit. on p. 84).

[58] Paul VC Hough. *Method and means for recognizing complex patterns*. US Patent 3,069,654. Dec. 1962 (cit. on p. 90).

[59] Richard O. Duda and Peter E. Hart. «Use of the Hough Transformation to Detect Lines and Curves in Pictures». In: *Commun. ACM* 15.1 (Jan. 1972), pp. 11–15. ISSN: 0001-0782. DOI: 10.1145/361237.361242. URL: https://doi.org/10.1145/361237.361242 (cit. on p. 90).

[60] D.H. Ballard. «Generalizing the Hough transform to detect arbitrary shapes». In: *Pattern Recognition* 13.2 (1981), pp. 111–122. ISSN: 0031-3203. DOI: https://doi.org/10.1016/0031-3203(81)90009-1. URL: https://www.sciencedirect.com/science/article/pii/0031320381900091 (cit. on p. 90).