



Escuela  
Politécnica  
Superior

# Human Segmentation with Convolutional Neural Networks



Computer Engineering Degree

## Bachelor's Thesis

Author:

Mario Martínez Requena

Advisors:

José García Rodríguez

Alberto Garcia-Garcia

Sergio Orts Escolano



Universitat d'Alacant  
Universidad de Alicante



UNIVERSITY OF ALICANTE

BACHELOR'S THESIS

# Human Segmentation with Convolutional Neural Networks

*Author*

Mario MARTÍNEZ REQUENA

*Advisors*

José GARCIA-RODRIGUEZ

Alberto GARCIA-GARCIA

Sergio ORTS ESCOLANO

*A thesis submitted in fulfilment of the requirements  
for the Bachelor's Degree in Computer Engineering*

*in the*

Department of Computer Technology

June 6, 2018



This document was proudly made with L<sup>A</sup>T<sub>E</sub>X, Plotly and TikZ.

This work is licensed under a [Creative Commons](#)  
“Attribution-ShareAlike 4.0 International” license.





*“Art is nothing if you don’t reach every segment of the people.”*

Keith Haring



# Abstract

In this project, a system capable of segmenting human shapes in images was developed. The first step towards that objective was a revision of the current *State Of The Art* (**SOTA**) about *Deep Learning* (**DL**), image segmentation, Convolutional Neural Networks (**CNNs**) and person segmentation in images. In addition, an analysis about training and testing datasets suitable for our desired task was conducted. After that, the actual implemented system will be presented. It consists of a U-Net **CNN**, an encoder-decoder type of network. Then, its layers and theoretical background were explained. Next, the results obtained segmenting the selected dataset and our own images were presented. Lastly, an overview of the project and the achievements as well as future lines of research finalize this Thesis.



# Resumen

En este trabajo fue propuesta la implementación de un sistema capaz de segmentar figuras humanas en imágenes. El primer paso hacia ese objetivo fue el de realizar una revisión de los últimos avances en *Deep Learning* ([DL](#)), segmentación de imágenes, Redes Neuronales Convolucionales ([CNN](#) por sus siglas en inglés) y segmentación de personas en imágenes. Además, se incluyó una investigación sobre conjuntos de datos para entrenamiento y prueba de nuestro sistema. Tras esto, el sistema implementado es presentado. Consiste en una red tipo *encoder-decoder* U-Net [CNN](#). De ella se explica las distintas capas que la conforman así como sus fundamentos teóricos. Después se añaden los resultados experimentos realizados sobre imágenes propias y de un conjunto de datos. Por último, esta Tesis concluye con una revisión del trabajo realizado, los logros obtenidos y futuras direcciones de investigación.



# Acknowledgements

Firstly, I would like to thank my three advisors individually:

Jose, my two most remarkable university experiences have came out of your office, the ERASMUS period and this final degree project. Thanks for your guidance in both of them.

To Sergio, thanks for your kindness, patience and assistance during this project.

And lastly, to Albert, thanks for your willingness to help with whatever you could, all the encouraging messages and unexpected jokes when talking about this project. Your personal approach have really made the Thesis easier.

Before finalizing, I would like to thank Gonzalo, Unai, Álvaro, Esteban, Manu, Quico, Paco and Juan Pablo. You definitely have been the best company during these years that I could have asked for.

Lastly, I would like to end by adding my most sincere gratitude towards my family for their unconditional love and support. This Thesis is dedicated to them.



# Contents

<b>Abstract</b>	vii
<b>Resumen</b>	ix
<b>Acknowledgements</b>	xi
<b>Contents</b>	xiii
<b>List of Figures</b>	xvii
<b>List of Tables</b>	xix
<b>List of Acronyms</b>	xxiii
<b>1 Introduction</b>	1
1.1 Outline . . . . .	1
1.2 Motivation . . . . .	1
1.3 Proposal . . . . .	2
1.4 Goals . . . . .	2
1.5 Structure . . . . .	2
<b>2 State Of The Art</b>	3
2.1 Introduction . . . . .	3
2.2 Deep Learning (DL) . . . . .	3
2.3 Convolutional Neural Networks (CNNs) . . . . .	4
2.4 Image Segmentation with Convolutional Neural Networks (CNNs)	5
2.5 Human Segmentation . . . . .	5
2.5.1 RGB Approaches . . . . .	6
2.5.2 Additional Information Layers Approaches . . . . .	7
2.6 Datasets Comparison . . . . .	10
<b>3 Methodology</b>	11
3.1 Libraries and Frameworks . . . . .	11
3.1.1 CUDA . . . . .	11
3.1.2 cuDNN . . . . .	13
3.1.3 TensorFlow . . . . .	13
3.1.4 Keras . . . . .	14
Optimizer . . . . .	15
Loss function . . . . .	16
Metrics . . . . .	16

Compiling example . . . . .	17
Fit . . . . .	17
Callbacks . . . . .	18
Example of fit and callbacks . . . . .	19
3.1.5 Other packages . . . . .	20
3.2 Software Development Environment . . . . .	21
3.2.1 First Stages of Development Environment . . . . .	21
3.2.2 Training Software Environment . . . . .	22
3.2.3 Miscellaneous . . . . .	23
3.3 Hardware . . . . .	24
<b>4 People segmentation with U-Net</b>	<b>27</b>
4.1 Introduction . . . . .	27
4.2 Dataset Preprocessing . . . . .	27
4.3 U-Net As A <i>Convolutional Neural Network (CNN)</i> . . . . .	29
4.3.1 Layers . . . . .	30
Convolutional Layers (Conv2D) . . . . .	30
Batch Normalization . . . . .	33
Activation . . . . .	34
Pooling . . . . .	35
Upsampling . . . . .	35
Concatenation . . . . .	36
4.3.2 U-Net Iteration . . . . .	36
4.4 Data augmentation . . . . .	37
4.5 Models . . . . .	38
4.6 Loss And Metrics Functions . . . . .	38
<b>5 Experiments</b>	<b>41</b>
5.1 Introduction . . . . .	41
5.2 Training and Testing Methodology . . . . .	41
5.3 Experiments . . . . .	43
5.3.1 Training . . . . .	43
5.3.2 Results . . . . .	45
5.3.3 Testing The System With Our Images . . . . .	49
<b>6 Conclusions</b>	<b>51</b>
6.1 Conclusions . . . . .	51
6.2 Results . . . . .	51
6.3 Highlights . . . . .	51
6.4 Personal . . . . .	52
6.5 Future work . . . . .	52
<b>A U - NET source code</b>	<b>53</b>
A.1 Git Repository . . . . .	54
A.2 Dependencies . . . . .	54
A.3 Executing . . . . .	55

<b>B Concepts</b>	<b>57</b>
B.1 Momentum . . . . .	57
B.2 Nesterov accelerated gradient . . . . .	57
B.3 Learning rate . . . . .	58
<b>Bibliography</b>	<b>61</b>



# List of Figures

2.1	Background substraction.	6
2.2	Youtube-assisted human segmentation architecture	7
2.3	Infrared image segmentation.	8
2.4	Infrared image segmentation methods comparison.	8
2.5	Multi-spectral image segmentation framework.	9
2.6	RGB-D segmentation system overview.	9
3.1	CUDA hierarchy of threads, blocks and grids.	12
3.2	NVIDIA SDK stack	12
3.3	Example of Tensors	13
3.4	Debugger stopped in a breakpoint displaying variables values	22
3.5	Docker arquitecture compared to traditional linux containers.	23
4.1	Masks combining done with a Python Script and Pillow	28
4.2	Architecture of the central part of the implemented U-Net.	29
4.3	Encoder-decoder networks visual explanation.	30
4.4	Architecture of fully connected artificial networks.	31
4.5	Spatial convolution	32
4.6	Batch normalization impact in accuracy	33
4.7	Activation functions.	34
4.8	Max pooling visual example	35
4.9	Upsampling visual example.	36
4.10	Different U-Net models.	38
5.1	Learning rate decay due to <i>ReduceLROnPlateau</i> .	42
5.2	TensorBoard log visualization.	42
5.3	Loss functions and dice score evolution during training and validation of the <i>unet_128</i> model.	43
5.4	Loss functions and dice score evolution during training and validation of the <i>unet_256</i> model.	44
5.5	Loss functions and dice score evolution during training and validation of the <i>unet_512</i> model.	44
5.6	Loss functions and dice score evolution during training the and validation of <i>unet_1024</i> model.	44
5.7	Accuracy obtained in during the first experiments.	45
5.8	Example of how the segmentation results are going to be displayed.	46
5.9	Good accuracy segmentation examples.	47
5.10	Bad accuracy examples.	47
5.11	Masks generated with <i>unet_512</i> model for our pictures.	49

B.1 Application of momentum to the learning process . . . . .	57
B.2 Nesterov updates compared to normal momentum . . . . .	58
B.3 Comparison between diverse learning rate schedules and adaptive learning methods. . . . .	58

# List of Tables

2.1 Datasets for human segmentation comparison. . . . .	10
3.1 Hardware specifications of Asimov. . . . .	24



# List of Listings

3.1	Simple Keras sequential model. . . . .	14
3.2	Keras model compilation example. . . . .	17
3.3	Example of callbacks and fit_generator in Keras. . . . .	19
3.4	Numpy usage in data augmentation. . . . .	20
4.1	Keras convolutional layers example code for U-Net. . . . .	32
4.2	Batch normalization in U-Net. . . . .	34
4.3	Activation layer used in the implementation of the project. . . . .	35
4.4	Pooling used on the project. . . . .	35
4.5	Upsampling used in this project. . . . .	35
4.6	Concatenation used on the project. . . . .	36
4.7	Data augmentation fragment code . . . . .	37
A.1	Code corresponding to the central part of the U-Net. This code has its visual representation on the figure 4.2. . . . .	53
A.2	Instruction for creating a new Conda environment from the tensorflow.yml file. . . . .	54
A.3	Commands for executing the project. . . . .	55



# List of Acronyms

<b>2D</b>	two-dimensional	<b>GUI</b>	Graphical User Interface
<b>3D</b>	three-dimensional	<b>HDD</b>	Hard Disk Drive
<b>Adam</b>	Adaptive Moment Estimation	<b>IDE</b>	Integrated Development Environment
<b>AI</b>	Artificial Intelligence	<b>NAG</b>	Nesterov accelerated Gradient
<b>ANN</b>	Artificial Neural Network	<b>NN</b>	Neural Network
<b>API</b>	Application Program Interface	<b>NVCC</b>	Nvidia CUDA Compiler
<b>BCE</b>	Binary Cross Entropy	<b>PTX</b>	Parallel Thread eXecution
<b>CNN</b>	Convolutional Neural Network	<b>RAID</b>	Redudant Array of Independent Disks
<b>CNTK</b>	Computational Network Toolkit	<b>ReLU</b>	Rectified Linear Unit
<b>CPU</b>	Central Processing Unit	<b>RNN</b>	Recursive Neural Network
<b>CUDA</b>	Compute Unified Device Architecture	<b>SGD</b>	Stochastic Gradient Descent
<b>CV</b>	Computer Vision	<b>SIMD</b>	Single Instruction Multiple Data
<b>GPGPU</b>	General-Purpose computing on Graphics Processing Units	<b>SOTA</b>	State Of The Art
<b>cuDNN</b>	CUDA Deep Neural Network	<b>SoC</b>	Systems on Chip
<b>DL</b>	Deep Learning	<b>SSD</b>	Solid State Drive
<b>GPU</b>	Graphics Processing Unit	<b>SSH</b>	Secure Shell



# Chapter 1

## Introduction

*This first chapter introduces the main topic of this work. It is organized in five different sections: Section 1.1 sets up the framework for the activities performed during this project, Section 1.2 introduces the motivation behind this work, Section 1.3 explains the proposal developed, Section 1.4 presents the specific and generic goals setted and Section 1.5 displays the content organization in this thesis documentation.*

### 1.1 Outline

In this Bachelor's Thesis we have researched and developed a solution oriented to segment persons in images. The project is composed by a U-NET [1] (a *Convolutional Neural Network (CNN)*) and Python scripts that help to modify the images in order to feed them properly to the network.

The main goal of the project is to obtain the best possible precision segmenting people, but also be able to perform that segmentation in near real time, since the final goal is to integrate the system on a mobile robotic platform.

This project addresses one of the tasks proposed in the COMBAHO: *come back home system for enhancing autonomy of people with acquired brain injury and dependent on their integration into society* national project, ID code (DPI2013-40534-R), funded by the *Ministerio de Economía y Competitividad (MEC)* of Spain with professors José García-Rodríguez and Miguel Ángel Cazorla-Quevedo from the University of Alicante as main researchers.

The project was developed during the period from February to June 2018 in collaboration with the *Department of Computer Technology (DTIC)* at the university of Alicante.

### 1.2 Motivation

In this document we summarized the results of the work done during the *Bachelors Degree in Computer Engineering*, taken between the years 2014-2018 at the *University of Alicante*. This work has been motivated by the collaboration with the *Department of Computer Technology* in research tasks related with High Performance Computing, Computer Vision, Deep Learning and the COMBAHO project.

## 1.3 Proposal

In this project, we propose a segmentation system that will take images as the input and will output a mask predicting which pixels are part of a human shape. The system will use a U-Net [CNN](#) as the main core of the program, and will be trained with specific datasets in order to perform the best possible prediction. This U-Net will be [GPU](#) accelerated to cut training times.

## 1.4 Goals

The main goal of this project is to develop a system capable of segmenting person in images and implement it using [CNNs](#).

The project will begin with a research on modern techniques for human and general segmentation. This research will comprise related datasets (with a comparison), an introduction of [DL](#), and more specifically [CNN](#) core concepts and also a review of existing [DL](#) frameworks.

After an introduction to the theoretical concepts of neural networks, we will develop and explain the U-Net, as well as the training method. Once the U-Net is implemented, we will perform several tests adjusting the size, hyperparameters and conditions in order to obtain a better understanding of how it works and the best performance possible. Finally, we will discuss the obtained results and propose future research directions.

## 1.5 Structure

The structure of this document is the following: The outline, motivations and goals are explained on this first Chapter 1, and then, Chapter 2 will contain the related works, history and current *State Of The Art* ([SOTA](#)). In Chapter 3, the software and hardware environment that supported the development of the project is presented. Chapter 4 introduces the implemented U-Net, its architecture, layers and related concepts, and in the following Chapter 5, the execution and results of the experiments are showcased. The last Chapter 6 summarizes what can be extracted out of this work and establish the directions for future investigation. Appendix A displays the source code of the implemented U-Net and lastly, Appendix B explains some [DL](#) concepts.

# Chapter 2

## State Of The Art

*This chapter presents the State Of The Art (**SOTA**) of Human Segmentation in colour images. It contains six main sections: Section 2.1 gives a brief introduction to this Chapter. Section 2.2 describes Deep Learning (**DL**) and its current state on image segmentation. Section 2.3 presents Convolutional Neural Networks (**CNNs**) and the reasons why they are on the verge of Computer Vision nowadays. Section 2.4 explains why and how **CNNs** are used for image segmentation. Section 2.5 finally introduces the main core of this project, Human segmentation in colour images and its **SOTA**. Finally, Section 2.6 displays a brief human-segmentation overview and comparison of existing datasets for human segmentation.*

### 2.1 Introduction

Segmenting people from images is a subject still being in focus thanks to the development of the digital photography field. Recently, this field has had a vertiginous expansion thanks to the smartphones, visual surveillance systems, etc. This kind of segmentation problem is under the umbrella of semantic segmentation, and in that way, it has benefited of the improvements that *Deep Learning* (**DL**) and *Artificial Neural Network* (**ANN**)-based methods have brought to this sector. However it still has some challenges to surpass. In this Chapter, the path from **DL** to human segmentation is described, giving a brief introduction to all the steps needed to achieve a solution for the proposed problem.

### 2.2 *Deep Learning* (**DL**)

Machine learning technologies have been powering many aspects of our daily life for years. Things as content filtering, web search, add recommendations and object identification in images, among others. Traditional machine learning was limited in the way it processes data, as many functionalities needed specific programming to perform certain tasks, not being able to receive raw data and transform it into a suitable representation without human intervention.

This disadvantage is where *Deep Learning* (**DL**) shines. **DL** is a machine learning subset characterized by being able to process raw data and automatically learn the features needed to perform determined tasks. This ability is based

on stacking several non-lineal transformation modules that convert the raw input data into a higher level, more abstract representation.

These layers vary depending on the function wanted to be performed. For example, in classification tasks, the high level layers will amplify relevant aspects dismissing the less important variations. In images, the first layers start by detecting edges in particular locations, the second ones detect edges independently of their location, the third ones assemble these edges into bigger combinations, and so on.

The important point of the previous paragraph is that the weights of these layers of feature are not designed by humans. Instead, they are learned from the data by a general purpose learning technique.

Thanks to this general learning process, **DL** has been pushing the advancement in the computing field, solving complex problems that have resisted solving attempts by traditional artificial intelligence techniques. Thanks to its ability for extracting patterns and structures out of complex raw data, and together with the increase of computational power, its application fields have been expanding to many domains of science and business.

One particular terrain that has been particularly benefited from the development of Deep Learning is Computer Vision. **DL** has been setting new standards in fields such us medical and biological image processing ([2], [3], [4]), thanks to the advancements in one of its specific parts, Convolutional Neural Networks (**CNNs**).

## 2.3 Convolutional Neural Networks (**CNNs**)

**CNNs** are a type of Neural Network [5] designed to process data that come in the form of multiple arrays. This is useful for processing things as natural language, audio, video and images. Inspired by the animal visual cortex organization [6], they are developed around four concepts: local connections, shared weights, pooling and the use of many layers (see Chapter 4 and Subsection 4.3.1 for a detailed explanation of these concepts.)

Thanks to the reduction of computing and memory usage obtained out of these four pilars, **CNNs** have been widely used on images. This reduction is mainly achieved thanks to the replacement of the whole matrix multiplication in standard Neural Networks (**NNs**). Fully connected **NNs** connect every single element of the input with each hidden element, generating huge quantities of parameters without any kind of spatial awareness. Instead, in **CNNs**, each element of the hidden layer is mapped to an specific area of the input, generating a smaller but sufficient number of parameters that also have spatial awareness. Additionally, pooling layers are applied to reduce the spatial dependency of the detected feature, or, in other words, making the network learn new features without them needing to be in an specific part of an image.

Thanks to the development of new computational techniques, *Graphics Processing Unit* (**GPU**) acceleration and the expansion of **DL** frameworks and libraries that make prototyping easier and faster than ever (see Chapter 3 for more information about some of them), **CNNs** usage have been rising, making

them useful for diverse fields such as handwriting recognition, face detection, behaviour recognition, or, the one issued in this work, image segmentation.

## 2.4 Image Segmentation with Convolutional Neural Networks (**CNNs**)

Thanks to the characteristics mentioned on Section 2.3, **CNNs** have had remarkable results solving challenging computing tasks such as image classification, segmentation and object detection, achieving state-of-the-art results in these tasks. This is thanks to the **CNN** ability to learn hierarchical representation of the raw input data.

In the last years, one of the main fields of application of **CNNs** have been in medical image segmentation ([7], [1]). Apart from the **2D** capabilities used to delineate organs, malformations [8], etc., **CNNs** have been impressive with their **3D** abilities, helping also to process MIR scans [9].

Another interesting field where **CNNs** have been applied is in semantic segmentation. This is a particular segmentation variation that aims to split the image in several classes (car, people, animal, etc.). At the beginning, Convolutional Networks were forsaken by traditional computer vision and machine learning methods but, in the ImageNet competition in 2012, **CNNs** surprised the *Computer Vision (CV)* community. The competition consisted on the image classification of one million images with 1000 different classes, and the results obtained [10] where ground-breaking, almost halving the error rates of the best competing approaches. This result was achieved thanks to the use of Graphics Processing Units (**GPUs**), Rectified Linear Units (**ReLUs**) activation functions (see Subsection 4.3.1 for more information) and a new regularization and data augmentation techniques.

Thanks to this revolutionary results, **CNNs** have been the dominant approach to computer vision in recent times, obtaining near human performance in some tasks. Some companies such as Google, Facebook and Microsoft, among others, have been quickly adopting this technology, due to its reasonable computing performance and the hardware advancement done by companies like NVIDIA, Qualcomm and Samsung, that are even developing *Systems on Chip (SoC)* that dramatically accelerate the common operations used for DL and CNN networks.

## 2.5 Human Segmentation

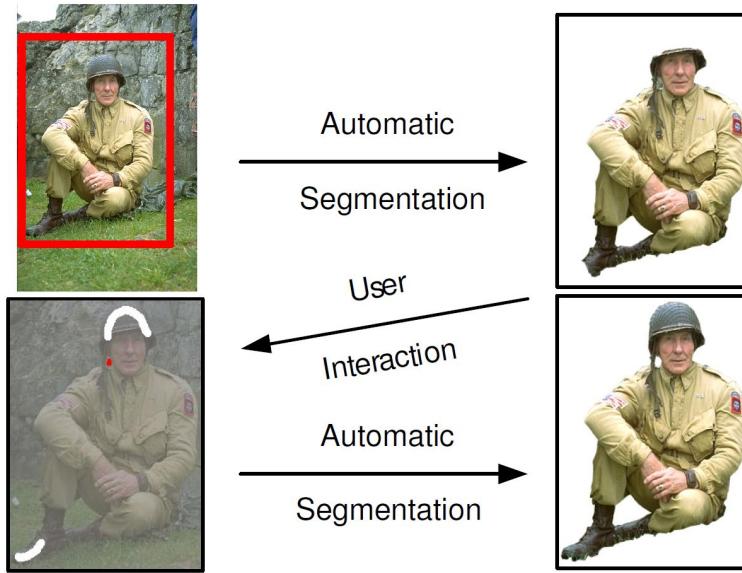
Human segmentation in images is still an important problem in computer vision that has become more popular in recent years thanks to the expansion of the digital photography in the form of smartphones, as well as visual surveillance, robotics, autonomous cars, etc. The human segmentation is under the umbrella of semantic segmentation as it aims to classify pixels on an image

into people and not-people. It could be also classified as a binary classification problem at pixel level.

Many different approaches have been taken lately to perform this segmentation task and in the following subsections some of them will be discussed.

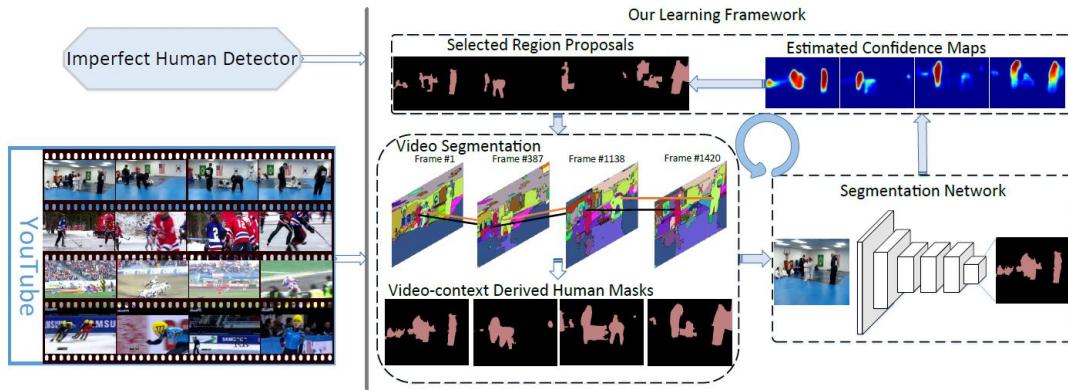
### 2.5.1 RGB Approaches

Before *Deep Learning* ([DL](#)) and *Convolutional Neural Network* ([CNN](#)) became the prominent research subject for segmentation, different ideas were tried. The 2004 article [\[11\]](#) mentions that classical image segmentation was approached using tools like edge and/or contrast detection, colour, graph cuts, etc. Paper proposes a method to subtract the foreground from the background, an approach used in the early days for human segmentation in RGB images. This method starts with the user selecting a loosely area around the subject to segment, performing an initial segmentation and letting the user correct again the segmentation with some traces before performing the last segmentation. This supervised method was combined with border matting to smooth the segmentation.



**Figure 2.1:** Background subtraction as a traditional approach of segmentation. Figure reproduced from [\[11\]](#).

Nowadays, with the neural networks pushing the State-Of-The-Art, the only limitation that they found is the size of the annotated datasets. The article [\[12\]](#) proposed a way to take advantage out of the largest video "database", YouTube. The proposed method divides the weakly-annotated videos in supervoxels, and then classify the superpixels within the supervoxels are then classified as human or not-human with graph optimization. These obtained masks are fed into a [CNN](#). The result is a weakly supervised system that improve previous [SOTA](#) on the PASCAL VOC 2012 dataset.



**Figure 2.2:** Youtube-assisted segmentation architecture. Feed-back flow between CNN and imperfect human detection could be observed on the top right part. Figure reproduced from [12].

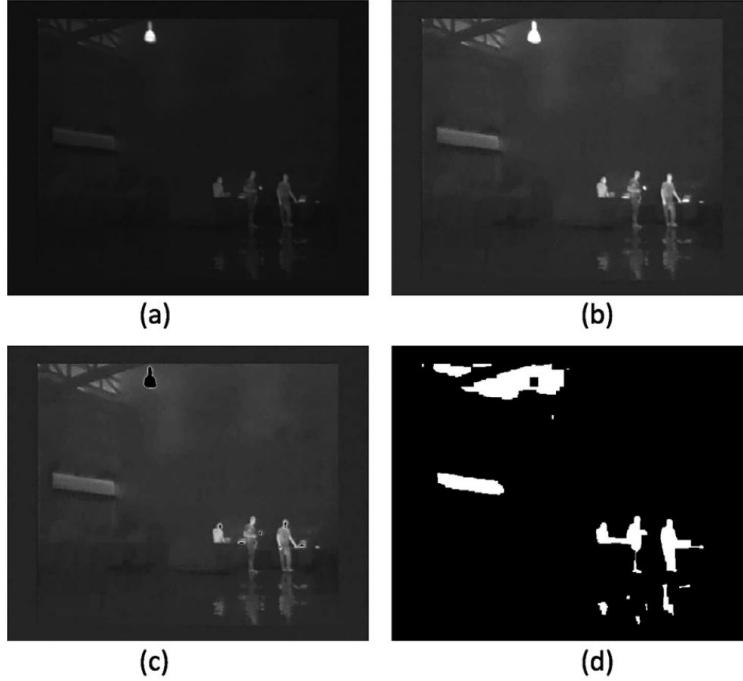
An extreme RGB image segmentation proposal is the one described in the [13] article. This paper describes an implemented system capable of 1000 fps human segmentation. In order to simplify the segmentation problem, it chooses an VGG-seg-net as the core of the approach, sacrificing some accuracy compared to the maximum achieved for the Baidu segmentation dataset [14] but obtaining a 10.000 times speed-up compared to the most accurate method for that dataset.

## 2.5.2 Additional Information Layers Approaches

One pattern repeated in various articles about human segmentation is the use of additional information layers (infrared, depth, thermal, etc.) to the RGB images to segment. These papers explain that segmentation based on just colour images could be rather imprecise as the human body could be presented in many different postures, could be occluded, or could be dressed in many different ways and colours.

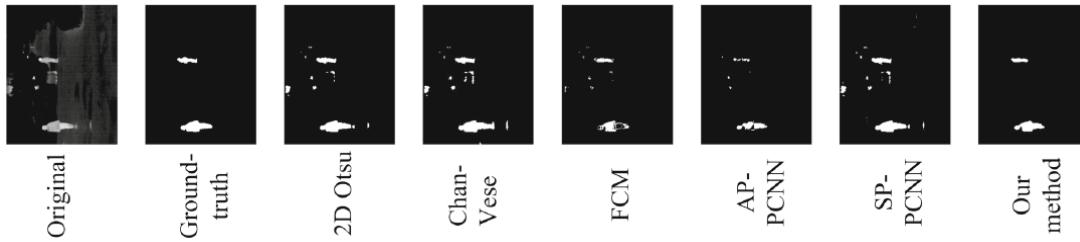
One popular additional information layer is infrared, and there are articles specifically aiming to segment people using just information from this data. This method of obtaining information has several advantages, as it does not depend on the light or colour, removing some RGB images flaws, but it still presents some challenges.

The system proposed in this article [15] uses information from just infrared surveillance cameras. In order to process these infrared images, traditional image thresholding is applied, obtaining initial blobs that are refined continuously until they fit each human figure.



**Figure 2.3:** Infrared image segmentation. a) input infrared image, b) scaled frame, c) displays the incandescent elimination and d) threshold frame, previous step before segmenting. Figure reproduced from [15].

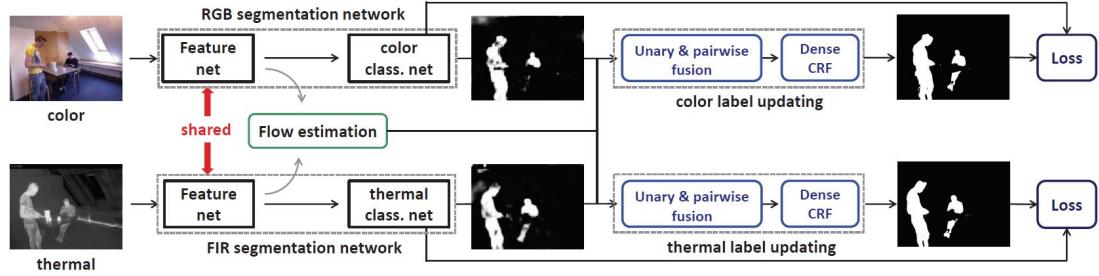
Another approximation to human segmentation using just infrared images is addressed in [16]. The proposed system is designed to be included on a mobile robot, and combined a pulse coupled neural network, the curvature gravity gradient tensor and the mathematical morphology. This paper concludes that, even though it achieves good results, some situations with infrared polluted environments are still a challenge.



**Figure 2.4:** Infrared image segmentation methods comparison. Best results obtained by the method described on the article from where the figure is reproduced [16].

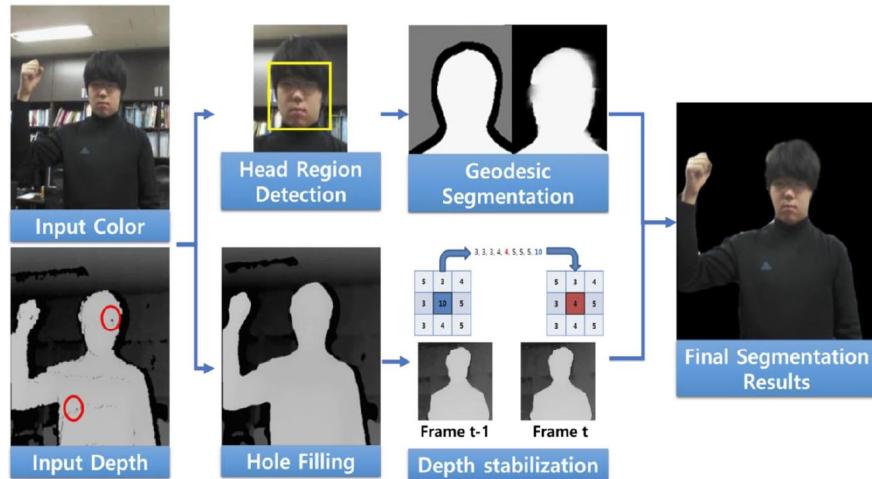
The approach presented in [17] offers an interesting combination of elements: RGB and infrared images processed with **CNNs**. This method proposes a double independent colour-thermal pipeline, where the feature maps from both sources are shared and they continually feedback each other, introducing

the training labels from one modality into another. This method has an advantage thanks to the multispectral information that manages, as it can generate highly precise segmentations without even pixel-level human annotations.



**Figure 2.5:** Multi-spectral image segmentation framework. Figure reproduced from [17].

Another stream of works leverage additional information such as depth data.[18] presents a real-time human segmentation video system that, with the help provided from the depth data, segments the human figures by dividing the frame into two regions, head region and body region. Once the image is divided, the system applies high computational geodesic matting to the head region and low computational to the body region. The traditional geodesic segmentation algorithm is modified to use depth information. Additionally, temporal and spatial smoothing is applied to enhance the coherence between different frames. Thanks to the additional depth information, this system obtains good results even when the human figure has similar colour to the background, but it failed to differentiate the human figure from other parts of the environment with similar depth.



**Figure 2.6:** RGB-D segmentation system overview. Figure reproduced from [18].

## 2.6 Datasets Comparison

In order to train our system, we needed to select a dataset that fit our desired goals. A list containing the most promising datasets and their characteristics is presented on Table 2.1.

Name	Representation	Year	Samples	Real/Synthetic	2D/3D	Overview
Unite the People [19]	Sports/Generic	2017	27.000	R	2D	Sports and generic position
MPI Dyna [20]	Human shape	2015	10	S	3D	40.000 scans of 10 subjects
Look into Person [21]	Generic	2017	50000	R	2D	Dataset mix
AAU [22]	Indoor activities	2016	5724	R	2D	RGB + D + Infrared
Pascal Voc 2012 [23]	Generic	2012	9993	R	2D	Human figures in diverse situations
Coco People [24]	Generic	2014	55000	R	2D	Semantic segmentation dataset

**Table 2.1:** Datasets for human segmentation comparison.

The second dataset listed, the *MPI Dyna* [20] contains a set of precise 3D scans of ten subjects, and it could be used to include these models on a virtual environment to generate more data, but as better and more time efficient alternatives were present, this dataset was discarded. *Look into Person* [20] is the second most interesting dataset of all the analysed. It contains a big amount of data that could have fitted our requirements, but the preprocessing needed to use it was larger than the one of the selected dataset. *AAU* [22] would have been the selected if we have opted to include additional information layers to complement the RGB approach. The last two featured datasets, *Pascal Voc 2012* [23] and *Coco People* [24] have the same common problem: they are generic semantic segmentation datasets that, even though they have the "human" class, they include a big percentage of images not useful for our human segmentation task.

The selected dataset to train our system is the *Unite the People* [19] one, as it contains a decent quantity of data, it is focused on human segmentation, it presents a good variety of situations and poses and the needed adaptation to fit our needs was the smallest of all analysed datasets (see Section 4.2 for more information).

# Chapter 3

## Methodology

This Chapter presents the computational environment used to develop this work. The first Section (3.1) lists and explains the main characteristics of the most remarkable libraries and frameworks that support the project. The second section (3.2) explains the software used to develop, train and test the neural network. The last section (3.3) presents the hardware of the server where the network has been trained.

### 3.1 Libraries and Frameworks

This section contains the theoretical background, main features and benefits behind the most important packages, libraries and frameworks used to develop this project. The subsections 3.1.1, 3.1.2 and 3.1.3 explains Keras, the TensorFlow API, and presents its basic functionality. The subsection 3.1.4 explain this powerful API called Keras, and presents its basic functionality and methodology. The last part of this section, the subsection 3.1.5, enumerate minor packages needed to develop this project.

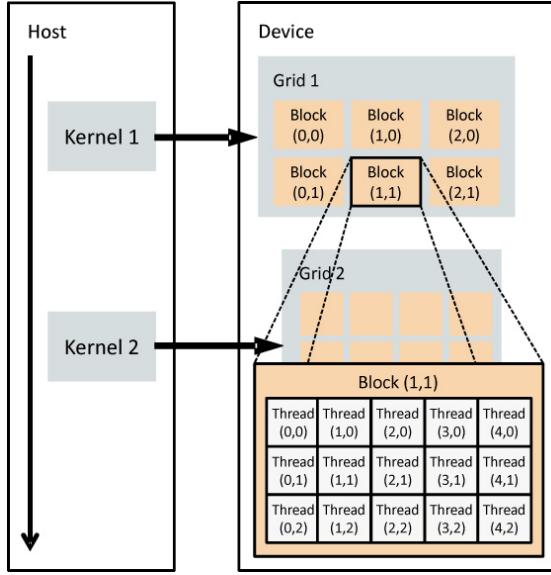
#### 3.1.1 CUDA

The NVIDIA *Compute Unified Device Architecture* (**CUDA**) [25] is a parallel computing platform and **API** which allows using an NVIDIA **GPU** for *General-Purpose computing on Graphics Processing Units* (**GPGPU**). This programming model enables programmers to use both the **CPU** (host) and **GPU** (device) (heterogeneous computing). CUDA programs are written in a variant of C/C++ and compiled to a high-level assembly language called *Parallel Thread eXecution* (**PTX**). All **PTX** instructions are *Single Instruction Multiple Data* (**SIMD**) executed by an entire warp of threads.

These extensions are composed by keywords added to those languages that enable the expression of parallelism in the application, directing the compiler to map portions of code for **GPU** execution.

The common **CUDA** program starts on the host, the **CPU**, where the elements to compute are on the computer's main memory, and they need to be transferred to the device (**GPU**) memory.

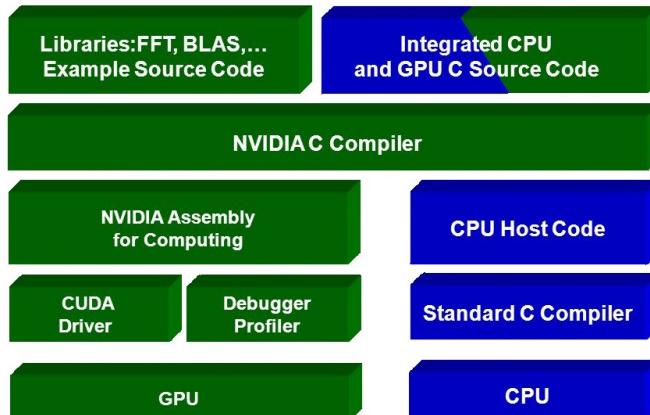
**CUDA** extensions allows the programmer to define functions, also called *kernels*, that are executed in parallel by different **CUDA** threads. These threads are organized into blocks of threads and a grid of blocks. Figure 3.1 shows this organization.



**Figure 3.1:** CUDA hierarchy of threads, blocks and grids with the typical program execution model. Image reproduced from [26].

As the above included figure outlines, each thread executes a copy of the kernel function. The threads inside the same block execute concurrently sharing memory with synchronization barriers (only when there are dependencies). Threads are grouped into blocks, and these are organized into grids along with other blocks executing the same kernel.

In order to support the programming model, **CUDA** also contains a custom driver for the **GPU** and a compiler (**NVCC** for C) capable of producing **GPU** assembly code (PTX) as well as **CPU** code for the host to be later compiled by a typical C compiler. **CUDA** also provides a profiler and a debugger for **GPU** programs. This set of tools is called **CUDA** SDK and its shown in the following Figure 3.2.



**Figure 3.2:** CUDA SDK stack (Green for GPU parts and blue for CPU ones). Image reproduced from [27].

### 3.1.2 cuDNN

As [CUDA](#) is mainly used in this project to support the [GPU](#) execution of [TensorFlow](#) 3.1.3, an special NVIDIA library needs to be added to the normal [CUDA](#) installation.

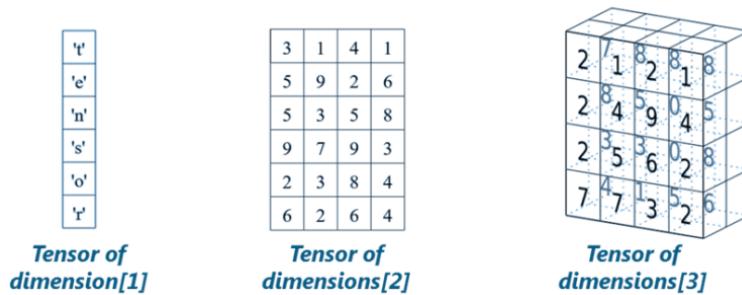
*CUDA Deep Neural Network* ([cuDNN](#)) is a library of primitives for deep neural networks that contains [GPU](#)-optimized implementations for common routines such as convolutions, pooling, normalization and activation layers (all these concepts are explained in Chapter 4). It is part of the *NVIDIA Deep Learning SDK* and supports the most common deep learning frameworks, such as *Caffe*, [CNTK](#), *Pytorch*, etc.

### 3.1.3 TensorFlow

[TensorFlow](#) [28] is an open source library for machine learning developed to substitute the old *DistBelief* system by Google Brain, a deep learning and *Artificial Intelligence* ([AI](#)) research group. It was publicly released on November, 2015 under the Apache 2.0 open source license. It is written in C++, Python and [CUDA](#).

[TensorFlow](#) uses dataflow graphs to represent computation. These dataflows consist of nodes and edges, where each node represents an instantiation of a mathematical operation, and the edges represent data, usually in the shape of tensors. These nodes are mapped to different computing units, as computers on a cluster, cores on a multicore CPU or, what will be used on this work, [GPUs](#).

The standard way to represent data on [TensorFlow](#) is with Tensors. Tensors are geometric objects represented by multidimensional data arrays where the underlying element type is defined or inferred at graph-construction time.



**Figure 3.3:** Example of Tensors. Image represented from [29].

[TensorFlow](#) offers flexibility, as every computation operation that can be expressed as a data flow graph can be implemented using it; portability, as it provides Application Program Interfaces ([APIs](#)) for most popular programming languages and could be run in a wide variety of platforms; and scalability, as the training could be run on Central Processing Units ([CPUs](#)), [GPUs](#) and it can be scaled from single cores to large scale systems.

[TensorFlow](#) was selected as the deep learning library for this project because it supports [GPU](#) acceleration, indispensable feature to shorten training times

by using the powerful server (displayed on Section 3.3) , has a widely spread community that makes it easy to learn it from scratch and because it is the recommended backend for Keras, the main API used to develop this project.

### 3.1.4 Keras

Keras [30] is a high-level open source neural network library capable of running on top of popular deep learning platforms as TensorFlow, Microsoft's CNTK and Theano. Its creator explained that it was created not to be a standalone machine-learning framework, but instead to be a high-level interface on top of the most popular deep learning backends.

Keras was created and maintained by François Chollet, a Google engineer with the following guiding principles on mind:

1. **Modularity** The main data structure of Keras, a model, could be understood as a sequence or a graph and fully configurable modules can be joined with little to no restriction. This is a helpful feature as it makes easy to put together the different layers that compose a Neural Network, such as activation functions, convolutional layers, etc.
2. **Extensibility** Appart from the already created modules, it is easy to implement and add new ones to the system, making Keras suitable for advanced research. Keras also provides examples to help with the creation of new parts.
3. **Python native** As Keras is written in Python, there are no separate model configurations files with custom file formats, making the debugging and extension easier.

Keras uses models as its main data structure. These models are a way to organize layers, and the simplest one is the sequential model, shown in Listing 3.1, which is a linear stack of layers.

```

1 from keras.models import Sequential
2 from keras.layers import Dense, Activation
3
4 model = Sequential([
5     Dense(32, input_shape=(784,)),
6     Activation('relu'),
7     Dense(10),
8     Activation('softmax'),
9 ])

```

**Listing 3.1:** Simple Keras sequential model.

The code presented above displays the creation of a simple sequential model on Keras. In its declaration are also included several layers that will be activated in that order. As the model needs to know the input shape, it needs to be specified in the first layer, as shown in the line number 5 of code.

When a model is already defined, there is one more step that has to be done before training with it: configure its learning process. This is done with the

*compile* method, and it receives three arguments: An optimizer, a loss function and a list of metrics.

## Optimizer

Optimizers on Deep Learning are a way to help the model in its labour to minimize the wanted functions. Keras has a list of already defined optimizers that can be instantiated, but this parameter could also instantiate an user defined Optimizer. The already defined optimizers are the following:

1. **SGD** *Stochastic Gradient Descent* ([SGD](#)) optimizer, with support for momentum (see [Appendix B.1](#) for explanation), learning rate decay (see [Appendix B.3](#)) and Nesterov momentum (or Nesterov accelerated gradient) (see [Appendix B.2](#)).
2. **Adagrad**. Explained on [Appendix B.3](#), is the base for many other optimizers. It is recommended to leave its parameters at their default values.
3. **Adadelta**. An extension for Adagrad, it aims to reduce its aggressiveness and monotonically-decreasing learning rate. It is recommended to leave its parameters at their default values.
4. **RMSprop**. This optimizer [31] has similarities with Adadelta as both aim to reduce the decreasing learning rates of Adagrad. It is a good choice for Recursive Neural Networks ([RNNs](#)). It is recommended to leave its parameters at their default values, except *Learning Rate*, which can be tuned.
5. **Adam**. *Adaptive Moment Estimation* ([Adam](#)) [32] is another optimizer that shares similarities with Adadelta and RMSprop as it also stores exponentially decaying average of past gradients, or, in other words, it adds the momentum concept to these optimizers. The default parameters are equal to those provided by the original paper.
6. **Adamax**. This is a variant of Adam based on the infinity norm (or uniform norm). The default parameters are equal to those provided by the original paper.
7. **Nadam**. Nadam [33] is identical to Adam, but instead of momentum it uses *Nesterov accelerated Gradient* ([NAG](#)). The default parameters are equal to those provided by the original paper, and it is recommended to leave its parameters at their default values.
8. **TFOptimizer**. This a wrapper class for native TensorFlow optimizers.

## Loss function

This parameter will specify the loss function that the model will try to minimize. It could be an already defined one, or it can be an objective function. The following loss functions are already implemented in Keras:

- mean\_squared\_error.
- mean\_absolute\_error.
- mean\_absolute\_percentage\_error
- mean\_squared\_logarithmic\_error
- squared\_hinge
- hinge
- categorical\_hinge
- logcosh
- categorical\_crossentropy
- sparse\_categorical\_crossentropy
- binary\_crossentropy
- kullback\_leibler\_divergence
- poisson
- cosine\_proximity

## Metrics

The last and the only optional argument when compiling a model is the metrics one. A metric is a function that is used to judge the performance of the model. Unlike loss function, the results of evaluating the metric function are not used as the objective function during training, as they are supposed to measure different parameters, but they could be used in specific callbacks (like EarlyStopping, explained on Section 3.1.4). When compiling a model, an existing metric could be provided as well as a Theano/TensorFlow symbolic function. It has two arguments:

- **y\_true**. Ground truth labels. Tensors from Theano/TensorFlow.
- **y\_pred**. Predictions. Tensors of the same size as y\_true.

A single tensor with the mean of the output array across all datapoints is given as a result. These are the already defined metrics in Keras:

- binary\_accuracy
- categorical\_accuracy
- sparse\_categorical\_accuracy
- top\_k\_categorical\_accuracy
- sparse\_top\_k\_categorical\_accuracy

## Compiling example

The following Listing 3.2 is an example of the compile code of the U\_Net 1024 model used in this project. It features custom-defined Keras loss function 3 and metric (line4).

```

1  model.compile(
2      optimizer=RMSprop(lr=0.0001),
3      loss=weighted_bce_dice_loss,
4      metrics=[dice_loss])

```

**Listing 3.2:** Keras model compilation example.

The *compile* method has more optional arguments than the ones presented above, such as *loss\_weights*, *sample\_weight\_mode*, etc, but they are not as important nor used.

## Fit

The basic training function in Keras is *fit*. It is used to train the model for a given number of iterations on a datasets, or epochs.

Its arguments are the following:

1. **x:** Numpy array or list of arrays (depending on how many inputs the model has) of training data.
2. **y:** Numpy array or list of arrays (depending on how many inputs the model has) of target/label data.
3. **batch\_size:** Number of samples per gradient update. Its default value is 32.
4. **epochs:** Number of iterations over the entire training x and y datasets, or, in other words, **epochs**.
5. **verbose:** This value indicates how much information about the training is displayed:
  0. No information.
  1. Progress bar.
  2. One line per epoch.
6. **callbacks:** List of *keras.callbacks.Callback* instances. This is a very helpful feature explained with more detail in Subsection 3.1.4.
7. **validation\_data:** A reference to some validation data. It could be provided via generator or tuples.
8. **validation\_steps:** This step parameter is only important if a generator is the validation data, and it indicates the number of steps (or batches) to take from the generator for every epoch.

The fit function accepts more arguments, but the most used and important are the above mentioned. An example of a fit function could be found in Sub-section 25.

Eventhough the function above explained is *fit*, the one used in this project is *fit\_generator*. The main difference is that this function trains on a data generated batch-by-batch by a Python generator. This generator is run in parallel to the model, what enables to perform data augmentation on the **CPU** while training the model on **GPU**. A *generator* needs to be provided as argument.

## Callbacks

Keras has predefined functions that are applied at given stages of the training that are called **callbacks**. These functions could be used to view internal states, save statistics, stop the training, etc. There is an example of them on this Listing Line 1. A brief view of the ones used in this project is listed below:

1. **EarlyStopping**. This callback stops the training when a monitored value has stopped improving. Its arguments are the following:
  - **monitor**: The value to be monitored. (A loss function, for example, as shown in Listing Line 3).
  - **min\_delta**: Minimum change in the monitored value qualified as "improvement".
  - **patience**: Number of epochs with no improvement required to stop the training.
  - **verbose**: Verbose mode. Same as explained in the above section.
  - **mode**: One of the following: {auto, min, max}. *min* mode will stop training when the value monitored has stopped decreasing, *max* mode will stop when the value has stopped increasing and *auto* mode, the direction is automatically inferred from the name of the monitored value.
  - **baseline**: Minimum value for the monitored quantity to reach.
2. **TensorBoard**. This parameter will save the results in a TensorBoard [34] compatible log. This is a TensorFlow tool to visualize dynamic graphs, test metrics and activations histograms.
3. **ReduceLROnPlateau**. This callback reduces the learning rate as a metric has stopped improving. It shares the majority of the parameters with the *EarlyStopping* callback, but adds some new ones:
  - **factor**: Factor by which the learning rate will be reduced.
  - **cooldown**: Number of epochs to wait before resuming normal operation after the learning rate has been reduced.

4. **CSVLogger.** Callback that saves the epochs results to a .csv file. Its parameters should contain the name of the CSV file, the separator and a boolean that indicate to continue writting from existing file or substitute it.
5. **ModelCheckpoint.** This callback will save the model after every epoch. Its parameters are the following:
  - **filepath:** Where to save the model.
  - **monitor:** Value monitored.
  - **save\_best\_only:** Boolean that indicates that a model only will be saved if is best than the current saved model.
  - **mode:** Similar to the same parameter one on EarlyStopping, this value will indicate if the model is saved based on the maximization, minimization or according to the name of the monitor value.
  - **save\_weights\_only:** Boolean that indicates if just the weights or the whole model will be saved.
  - **period:** Number of epochs between checkpoints.

### Example of fit and callbacks

The below presented code fragment is extracted from the U\_net of this project. The 'val\_dice\_loss' monitor is declared, as well as train\_generator.

```

1 callbacks = [
2     EarlyStopping(
3         monitor='val_dice_loss',
4         patience=8,
5         verbose=1,
6         min_delta=1e-5,
7         mode='max'),
8     ReduceLROnPlateau(
9         monitor='val_dice_loss',
10        factor=0.1,
11        patience=4,
12        verbose=1,
13        epsilon=1e-4,
14        mode='max'),
15     ModelCheckpoint(
16         monitor='val_dice_loss',
17         filepath='weights/' + params.title + '.hdf5',
18         save_best_only=True,
19         save_weights_only=True,
20         mode='max'),
21     TensorBoard(log_dir='logs'),
22     CSVLogger(filename='epochs/' + params.title + '.csv', separator=',',
23               append=False)
24 ]
25 model.fit_generator(
26     generator=train_generator(),

```

```

27     steps_per_epoch=np.ceil(float(len(ids_train_split)) / float(
28         batch_size)),
29     epochs=epochs,
30     verbose=2,
31     callbacks=callbacks,
32     validation_data=valid_generator(),
33     validation_steps=np.ceil(float(len(ids_valid_split)) / float(
34         batch_size))
)

```

**Listing 3.3:** Example of callbacks and fit\_generator in Keras.

In conclusion, Keras was selected as the main programming tool as it allows a really fast development of the neural networks, with its predefined layers, callbacks and easy-to-use features.

### 3.1.5 Other packages

1. **NumPy.** Numpy [35] is a Python library that adds support for large multidimensional arrays and matrices, as well as a collection of high level mathematical functions to operate on these arrays. It also provides tools for integrating C/C++ and Fortran code. Another NumPy functionalities widely used on this project are the math and random number capabilities. An example of usage can be found on the data augmentation performed on the project, displayed on Listing 3.4.

```

1 if np.random.random() < u:
2     height, width, channel = image.shape
3
4     angle = np.random.uniform(rotate_limit[0],
5                               rotate_limit[1]) # degree
6     ...

```

**Listing 3.4:** Numpy usage in data augmentation.

2. **Scikit-learn.**[36] This is a machine learning library for Python. It requires NumPy and SciPy and it features various classification, regression and clustering algorithms, such as gradient boosting, k-means and random forests. It is written in Python with some parts written in Cython, a Python variant that mixes with C to gain performance.
3. **Scikit-image.** Scikit-image [37] is an image processing library for Python. Similar to Scikit-learn, it requires NumPy and SciPy, and it includes segmentation algorithms, geometric transformations, color space manipulations, analysis, filtering, morphology, feature detection and more.
4. **Pandas.** This library [38] is written in Cython and C for performance and it is used for data manipulation and analysis. It offers rich data structures and functions designed to work with structured data fast and easily.
5. **Matplotlib.** Matplotlib [39] is the most popular Python library for producing plots and other 2D data visualizations. It creates interactive plots that are suitable for publication, and it interacts well with other libraries.

6. **OpenCV.** OpenCV (Open Source Computer Vision) [40] is a powerful library of programming functions for computer vision. It was developed by Intel and it offers functions for segmentation and recognition, image processing, gesture and figures recognition, etc, plus some statistical machine learning features. It also supports hardware acceleration that enables fast processing of images based on Intel primitives and OpenCL technologies.
7. **Pillow.** Pillow <sup>1</sup> is a fork of Pil, an image library for Python. It adds support for opening, manipulating and saving various image file formats. The Pillow library forked this main library as its support is discontinued, adding Python 3.x and additional features.

## 3.2 Software Development Environment

In this section, the development software used to implement this work will be explained. As the development of the segmentation system was divided between debugging and experimentation, the following parts will explain what was used on each part.

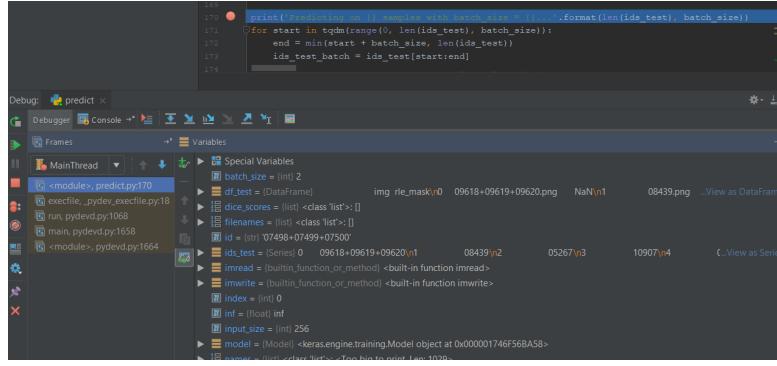
### 3.2.1 First Stages of Development Environment

These first stages consisted on implementing and debugging the neural network. During the whole project, PyCharm was used as the main IDE, and during the first steps, Conda was used as the package management.

PyCharm is a cross-platform Python IDE developed by JetBrains. This tool was used to develop the code, as its different functionalities provide a rich experience. The most useful tool is the ability to select the Python interpreter, allowing to use the environment created with Conda to execute the code natively on the IDE. Another useful tool used was the debug mode (see Figure 3.4), which can run the code step by step, freezing the execution, displaying variables and its state. This tool was essential during the first steps of development, as it was useful to detect the majority of the bugs. PyCharm was also useful in the second phase of the project, the training, as it could browse, access and modify remote code files natively on the IDE.

---

<sup>1</sup><https://pillow.readthedocs.io/>



**Figure 3.4:** Debugger stopped in a breakpoint displaying variables values

Conda is an open source package management multi-platform tool, useful for quickly creating development environments with different packages. Using a provided command prompt, an environment is created, where you can add diverse packages, and keep this from interfering with the ones in a different environment. This can be useful if the user wants to use different version of the same package on a single system or avoiding environment errors. Another important feature is the capability of saving, sharing and importing ready-to-use environments, avoiding package problems when executing your code on a different machine and allowing a fast deployment of the environment on another computer.

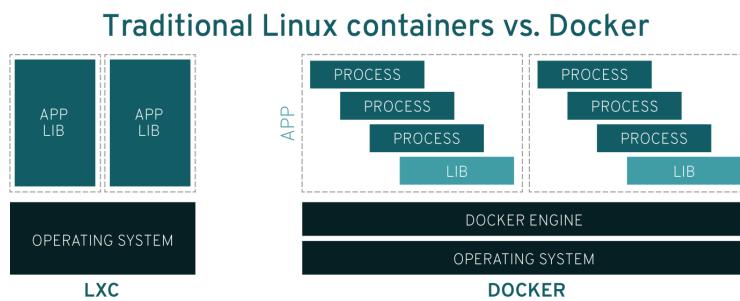
### 3.2.2 Training Software Environment

Once the program was fully debugged and running, we move on to the experimentation phase. In order to be able to train the biggest models of the neural network, powerful hardware and a flexible software was needed. The hardware part is explained on its corresponding Section 3.3, and the additional software used to train the network is explained below. The main topic in this Subsection is Docker, as it was the the principal software using during the experimentation process

Docker [41] is a computer program that performs operating-system-level virtualization also known as containerization. These containers are closed environments, with their own namespace, that allow the installation of diverse packages and programs without colliding with the already installed on the native operative system or other containers. Several instances of the same container or different containers can be executed at the same time on the same machine. This architecture, compared to in Figure 3.5. The main advantages of using docker containers are the following:

- **Modularity:** Docker allows to repair, update or even take down certain parts of an application without needing to take down the whole application.

- **Layers and image version control.** Docker images are made up of layers, and these layers are combined to make an image. Every time the image changes, a new layer is created. These layers could be used to create new containers making the build process much faster. Another feature of Docker related to layers is version control. Every change is recorded in a built-in changelog. This allows to go back to previous version if any problem happened on the new one.
- **Rapid deployment.** Thanks to the previous mentioned characteristics, a system deployment that used to take a long time could be done in with Docker in seconds.



**Figure 3.5:** Docker architecture compared to traditional linux containers. Figure reproduced from [42].

### 3.2.3 Miscellaneous

This part contains another support software used during the development of this work.

- **Git.** Git, with a private repository on BitBucket, was selected as the version control system for this work. In addition, GitKraken<sup>2</sup> was used as a *Graphical User Interface (GUI)* client, it provides an user-friendly visual feedback of the versions.
- **FileZilla.** As the dataset needs to be physically in the computer where the network is training, and the results were generated also in this machine, Filezilla's FTP capabilities were used to transfer files between training and development/debugging systems.
- **Putty.** Putty was used to access remotely to the training server to launch all the test and create and manage Docker environments.

<sup>2</sup><https://www.gitkraken.com/>

### 3.3 Hardware

*Deep Learning* (DL) and, in particular, CNN training is a computational demanding task, as it needs to process huge quantities of data. In order to being able to train in a reasonable time, it should be done in a powerful and energy efficient machine. The DTIC's *Asimov* server was assembled with that porpoise in mind, and its full hardware is displayed on Table 3.1.

The main features of the server are the three NVIDIA GPUs which were targeted at different goals. The Titan X will be devoted to deep learning, whilst a Tesla K40 was also installed for scientific computation purposes thanks to its exceptional performance with double precision floating point numbers. In addition, a less powerful GT730 was included for visualization and offloading the Titan X from that burden.

Regarding the software information, the server runs Ubuntu 16.04 LTS with Linux kernel 4.4.0 – 21–generic for x86\_64 architecture. The GPUs are running on NVIDIA driver version 361.42 and CUDA 7.5.

Asimov	
Motherboard	Asus X99-A <sup>11</sup> Intel X99 Chipset 4× PCIe 3.0/2.0 × 16 (×16, ×16/ × 16, ×16/ × 16/ × 8)
CPU	Intel(R) Core(TM) i7-5820K CPU @ 3.30GHz <sup>12</sup> 3.3 GHz (3.6 GHz Turbo Boost) 6 cores (12 threads) 140 W TDP
GPU (visualization)	NVIDIA GeForce GT730 <sup>13</sup> 96 CUDA cores   49 W TDP 1024 MiB of DDR3 Video Memory   PCIe 2.0
GPU (deep learning)	NVIDIA GeForce Titan X <sup>14</sup> 3072 CUDA cores   250 W TDP 12 GiB of GDDR5 Video Memory   PCIe 3.0
GPU (compute)	NVIDIA Tesla K40c <sup>15</sup> 2880 CUDA cores   235 W TDP 12 GiB of GDDR5 Video Memory   PCIe 3.0
RAM	4 × 8 GiB Kingston Hyper X DDR4 2666 MHz CL13
Storage (Data)	(RAID1) Seagate Barracuda 7200rpm 3TiB SATA III HDD <sup>16</sup>
Storage (OS)	Samsung 850 EVO 500GiB SATA III SSD <sup>17</sup>

**Table 3.1:** Hardware specifications of Asimov.

In addition, the server was configured for remote access using Secure Shell ([SSH](#)). The installed version is OpenSSH 7.2p2 with OpenSSL 1.0.2. Authentication based on public/private key pairs was configured so only authorized users can access the server through an [SSH](#) gateway with the possibility to forward X11 through the [SSH](#) connection for visualization purposes.

At last, a mirrored Redundant Array of Independent Disks ([RAID](#))<sup>1</sup> setup was configured with both Hard Disk Drives ([HDDs](#)) for optimized reading and redundancy to tolerate errors on a data partition to store all the needed datasets, intermediate results, and models. The Solid State Drive ([SSD](#)) was reserved for the operating system, swap, and caching.

---

<sup>1</sup><https://www.asus.com/Motherboards/X99A/specifications/>

<sup>2</sup>[http://ark.intel.com/products/82932/Intel-Core-i7-5820K-Processor-15M-Cache-up-to-3\\_60-GHz](http://ark.intel.com/products/82932/Intel-Core-i7-5820K-Processor-15M-Cache-up-to-3_60-GHz)

<sup>3</sup><http://www.geforce.com/hardware/desktop-gpus/geforce-gt-730/specifications>

<sup>4</sup><http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-titan-x>

<sup>5</sup>[http://www.nvidia.es/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001\\_v05.pdf](http://www.nvidia.es/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05.pdf)

<sup>6</sup><http://www.seagate.com/es/es/internal-hard-drives/desktop-hard-drives/desktop-hdd/?sku=ST3000DM001>

<sup>7</sup><http://www.samsung.com/us/computer/memory-storage/MZ-75E500B/AM>



# Chapter 4

## People segmentation with U-Net

*This chapter will present the different layers, parts, theoretical background and miscellaneous content related to the implemented U-Net. As the natural order dictates, the first Section (4.1) briefly introduces the U-Net chapter. Section 4.2 explains the modifications done to the used dataset. Section 4.3 presents the architecture of the U-Net, and describes the layers that compose it, as well as how they interact between each other. After this main part, Section 4.4 describes the data augmentation process embedded in the training process. The last two parts of this Chapter, Section 4.5 and Section 4.6 present the different U-Net models and the loss functions respectively.*

### 4.1 Introduction

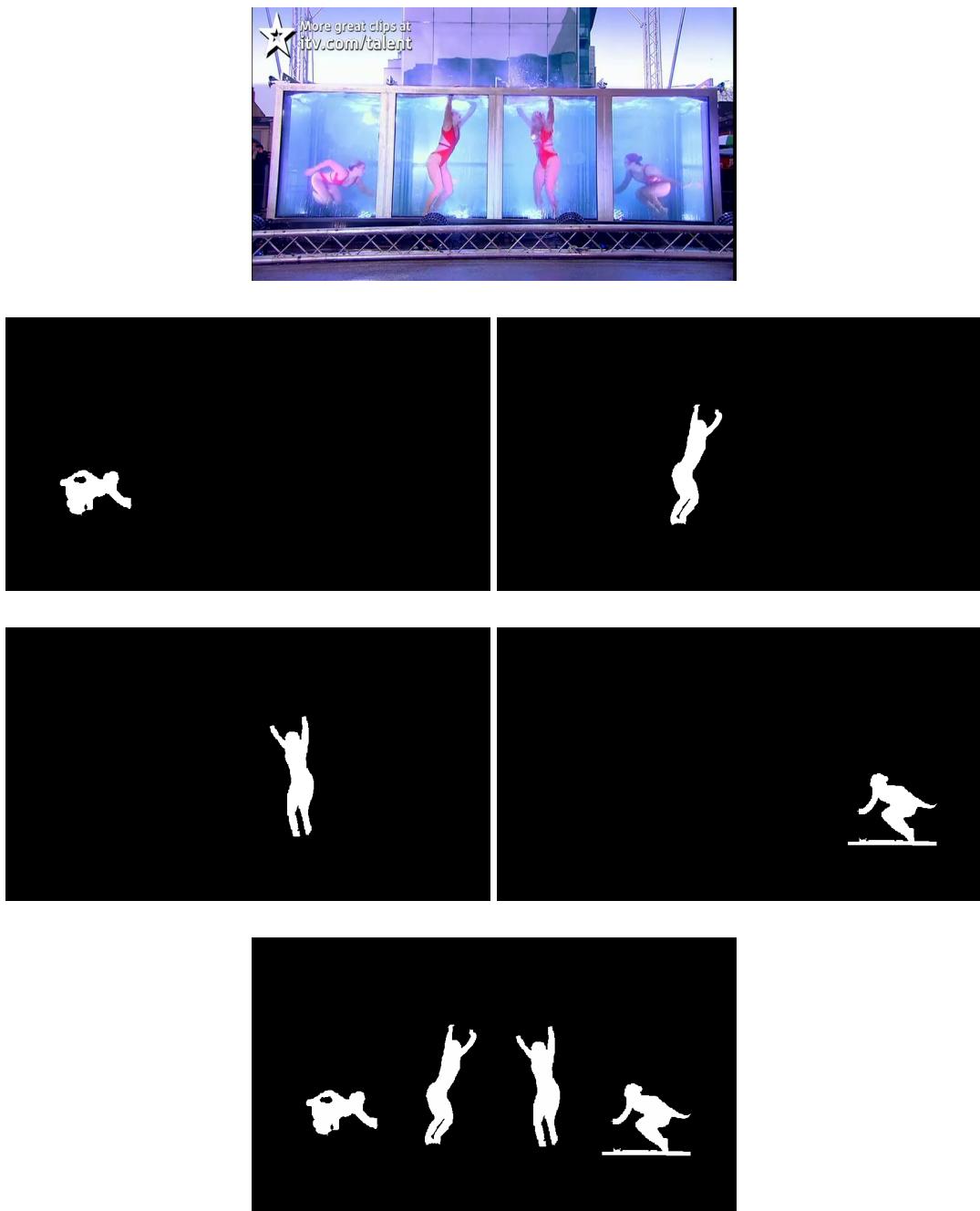
In the past years, **CNNs** have been pushing the *State Of The Art* (**SOTA**) in image segmentation [43], topping related publications(see Chapter 2). In this Chapter, the implemented U-Net and the parts that compose it will be explained. The implemented **CNN** was based on one used to compete on the Kaggle's Carvana Image Masking Challenge and provided by one of the advisors of this project, *Sergio Orts Escolano*.

### 4.2 Dataset Preprocessing

The type of segmentation that we are aiming for is class segmentation: we want to classify all the pixels of an image into two classes (People and not-people).

The main dataset used to train and test the network, Unite The People [19], has several images with more than a human figure on them (see middle part of Figure 4.1) and, instead of a single mask covering all the people, several masks, one per human.

This is problematic as we are focusing on segmenting the human class, not human instances. As the dataset originally provides mask instancing humans, we needed to unify these instances into class annotations. To that end, we needed to implement a preprocessing step to unify instance labels into a single class-level ground-truth mask (see Figure 4.1).

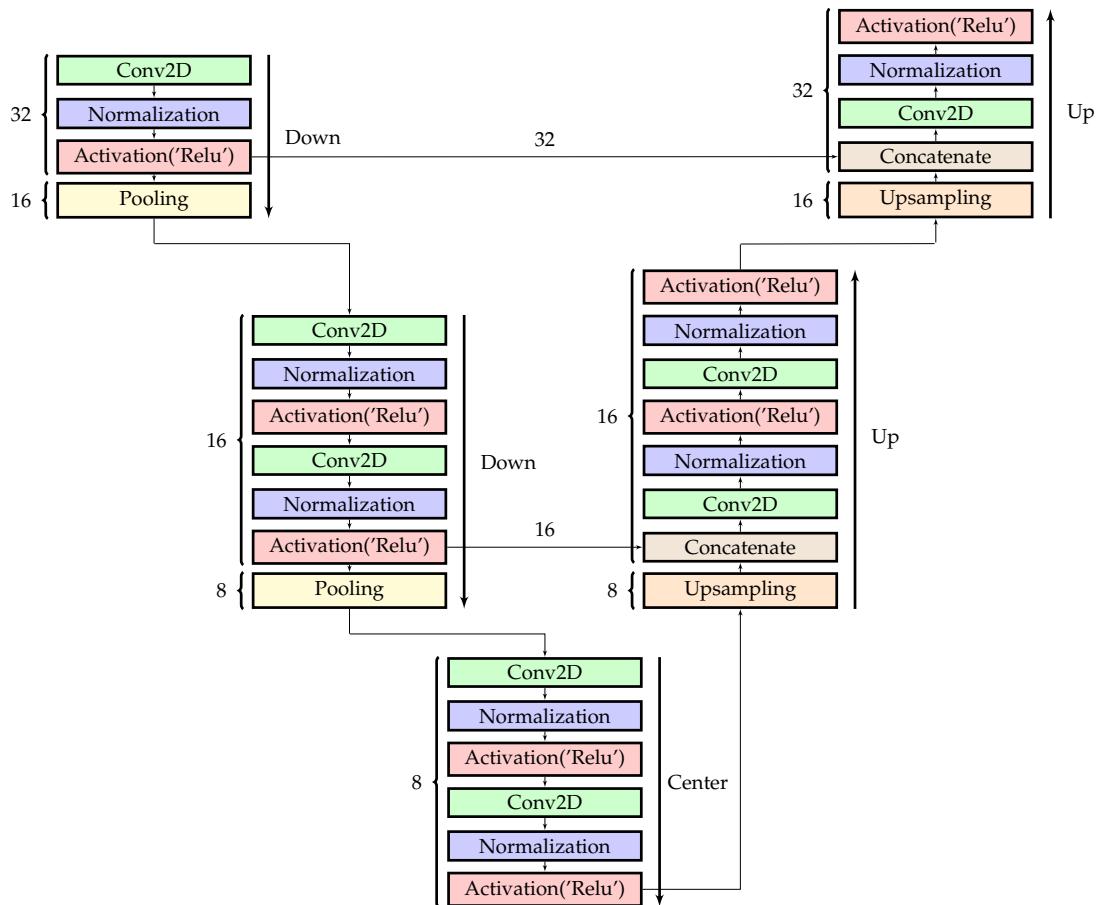


**Figure 4.1:** In the dataset, the image on top has four separate instances (middle rows) that were combined on the bottom.

As the dataset images come from other datasets (MPII Human Pose [44] and Leeds Sports Pose [45]), our dataset provided a CSV table explaining where the images came from. Taking these references into account, we were able to know which instance labels images correspond to each photography. These images got their masks combined with the Python library Pillow (see Subsection 3.1.5). The CSV table containing their names, used to split the dataset into the training and validation sets, was also updated with the new names.

### 4.3 U-Net As A Convolutional Neural Network (CNN)

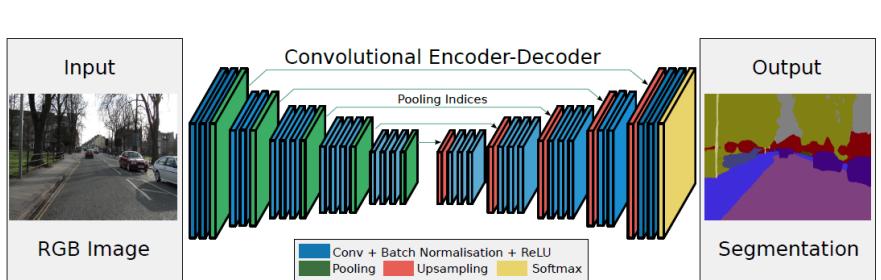
The U-Net implemented and used in this project follows a similar structure to the one of the original paper [1], as the main shape of the implemented network (see Figure 4.2) is really similar to the originally developed. In this section, the structure, the implemented layers (and their theoretical background) and their interactions will be presented.



**Figure 4.2:** Architecture of the common central part of the U-Net model Keras implementation of this project. The left part corresponds to the contracting encoder path and the right side to the expanding decoder part. This schema is based on the actual Keras code (see Listing A.1).

Before starting with the layers explanation, a general architectural description of the network is given. There is one type of CNNs that is only formed by the contracting path of our U-Net, the part that reduces the size of the "image" and continues to increase the number of filters in each convolutional layer. This type of networks aims to synthesize the information of the input into an uni-dimensional vector. An example will be a system that takes as an input an image that represents a number and the output is the represented number. This type of neural networks are called *encoder* networks.

The idea behind this project is to generate a mask that indicates which pixels correspond to human figures with nearly the same dimensions as the input image. In order to achieve this goal, encoder networks fall short. The kind of networks that we need to use are called *encoder-decoder*. The first half is the previously mentioned encoder network that summarizes the input, and the second part, the decoder, expands the size of the information until it reaches a similar dimension as the one of the input<sup>1</sup>. The structure of the encoder-decoder type of networks could be appreciated in the examples represented in Figure 4.3, Figure 4.2 and Listing A.1.



**Figure 4.3:** Encoder-decoder network architecture (Seg-Net). Image reproduced from [46].

### 4.3.1 Layers

This subsection will enumerate the different layers that compose the implemented U-Net and the theoretical background behind each one. These layers are stacked repeatedly to form different U-Net models, as explained on Section 4.5 and work together as presented on Section 4.3.2.

#### Convolutional Layers (Conv2D)

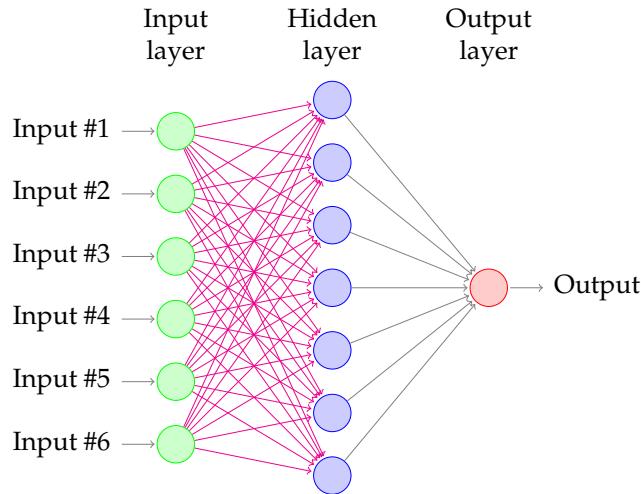
Convolution layers are probably the most important concept on CNNs. In the fully connected neural network architecture, every element of the input is connected to every hidden element in the next layer, as seen on Figure 4.4. This has two main drawbacks:

- **Weight matrix:** The weight matrix is generated with the same amount of elements as connection between two layers. In a fully connected neural network trying to process, for example, a  $32 \times 32 \times 3$  colour image with  $6 \times (5 \times 5)$  filters will produce approximately 14 million parameters. That is a very big quantity of parameters considering the reduced dimension of the image, making this method very inefficient for image processing.

---

<sup>1</sup>It is not the exact same size as every convolution layer slightly decreases the size due to padding.

- **Spatial awareness:** In the fully connected architecture, as all the inputs are treated equally, connecting them to all the neurons on the following layer, there is no possibility of obtaining spatial awareness out of this learning process.



**Figure 4.4:** Architecture of the fully connected neural networks. As every input is connected to every hidden neuron, there is not spatial awareness and the number of weights to store is huge, one per each connection between the input layer and the neurons (magenta arrows).

To solve these two problems we use convolutional layers, and they are based on two concepts: *local receptive fields* and *shared weights*.

- **Local receptive fields:** Receptive field is a biological inspired concept: it is the region that triggers a change on a neuron, and in the computer vision field, it is defined as the input region linked to a neuron.

As we have previously mentioned, when we are dealing with high dimensional inputs, such as images, it is highly unpractical to map each input parameter to every single hidden neuron as it will imply high memory and computational cost. This connection model is called *global receptive field*, and it is the opposite of how **CNNs** are organized.

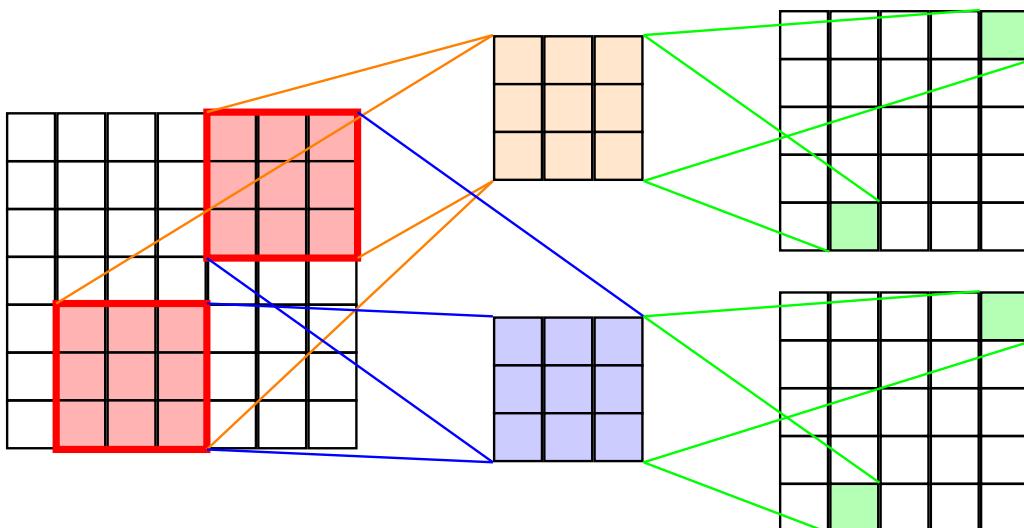
**CNNs** connect their neurons to localized areas of the input, with a method called *local receptive fields*. Convolutional layers use a local receptive field with the size of the convolution filter ( $(3 \times 3)$  on our project) that slides through the whole input, mapping a certain region of the input to each neuron. As not all the parts of the input are treated in the same way, the system acquires spatial awareness.

- **Shared weights:** Following with the comparison, in the fully connected architecture, each neuron will have a bias, and every connection with the input would make it learn a weight. However, **CNNs** use the same

weights and bias for all neurons, reducing drastically the number of parameters and improving the memory usage and computational performance. This means that all neurons of the hidden layer will detect the same feature in their correspondent area (local receptive field) of the input.

Due to this characteristic, the system acquires *translation invariance*, what means that a certain feature will be detected independently of where it is on the input image.

The local receptive field and shared weights are concepts that are used during the convolutions, as it could be seen on Figure 4.5. This figure shows how each filter generates a feature map once is applied throw the whole input. Several feature maps are created, one per each filter on the convolutional layers.



**Figure 4.5:** Spatial convolution. This diagram represents how the feature maps are created. The input, on the left side, is processed with filters, on the middle, to generate feature maps, on the right. Each filter generates one feature map. The red squares of the input generate the green features of the right part.

On our U-Net, and in the same way as the one of the original paper, ( $3 \times 3$ ) spatial convolutions are repeatedly applied to the image (visual example in Figure 4.5). This ( $3 \times 3$ ) dimension is determined as the filters need to have the same dimensions as the input, in that case, image, that have a **3D** structure (height, width, channels). In the above shown Listing (4.1), in each convolutional layer the number of filters is specified as the first parameter, and the following ones are the dimensions.

```

1     down3 = Conv2D(256, (3, 3), padding='same')(down2_pool)
2     ...
3     down4 = Conv2D(512, (3, 3), padding='same')(down3_pool)

```

```

4      ...
5      center = Conv2D(1024, (3, 3), padding='same')(down4_pool)
6      ...
7      up4 = Conv2D(512, (3, 3), padding='same')(up4)
8      ...
9      up3 = Conv2D(256, (3, 3), padding='same')(up3)
10     ...

```

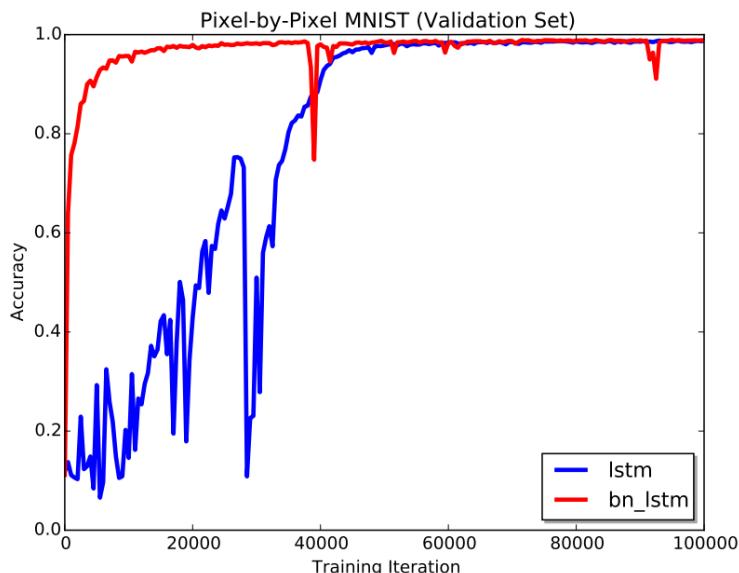
**Listing 4.1:** Keras convolutional layers example code for U-Net.

## Batch Normalization

In order to increase the stable distribution of activation values during training of our a neural network, batch normalization [47] is applied to the output of the previous Convolutional Layer.

This batch normalization normalizes the output of the previous layer by subtracting the batch mean and dividing by the batch standard deviation. This normalization helps speeding up the learning process, reducing *covariate shift*. Covariate Shift is defined as the change in distribution of network activations due to the change in network parameters during training.

The benefits of batch normalizations are that the network can get higher learning rates faster as it prevents activations from going really high or low, and reduces overfitting, with similar effects as regularization, preventing the memorization of values and their correct answers, as it adds some noise to each activation.



**Figure 4.6:** Batch normalization impact example on accuracy applied to an algorithm iterating throw MNIST dataset. Image reproduced from [48]

In the U-Net, the batch normalization is applied is the one already implemented by Keras (see Listing 4.2). This Keras implementation will keep the

mean activation close to zero and the activation standard deviation close to one. The Keras implementation also refers to the original batch normalization paper [47].

```
1 up4 = BatchNormalization()(up4)
```

**Listing 4.2:** Batch normalization in U-Net.

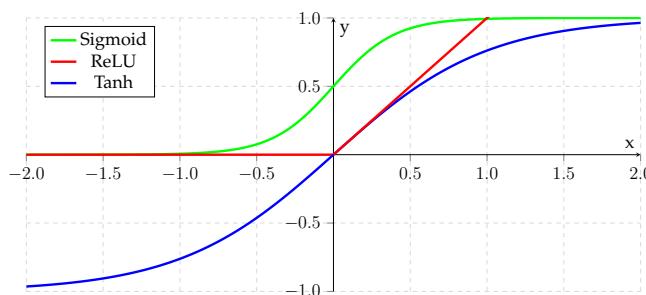
## Activation

Until this moment, the output of the previous layers is just a linear transformation of the input. This linearity implies that the network will not satisfy the *universal approximation theorem* [49]. Until we do not eliminate this linearity, the representational power of the network is limited. In order to transform the network into an universal approximator, **non-linearities** must be introduced.

These non-linearities are introduced via activation layers, with popular functions such as Sigmoid, Tanh or the one used in this neural network, *Rectified Linear Unit* (**ReLU**). These functions increase the non-linear properties without affecting the receptive fields of the previous convolution layers.

The three most popular functions Sigmoid, Tanh and ReLU (represented on Figure 4.7) are briefly explained:

- **Sigmoid:** This function squashes the results into the [0,1] range. Even though it has been widely used, it has to disadvantages that made the training process impossible in some situations: gradient saturation and not zero-centered outputs.
- **Tanh:** As the previous function, tanh squashes the results into an interval but, instead of [0, 1], tanh does it in the [-1, 1] range. As the previous function, it still saturates gradients but the output is zero-centered, making it a better alternative to the sigmoid function.
- **ReLU:** ReLU function works by thresholding the input at zero, making the gradient descent training significantly faster, as it does not saturate and has a lower computational cost than the two above mentioned functions. Thanks to these characteristics, it improved the results of the aforementioned functions in several benchmarks [50]. This makes **ReLU** the most common activation function for **CNNs**.



**Figure 4.7:** Most common activation functions. The most used for **CNNs** and the one selected for this project is the **ReLU** one.

In the project, we have used the default Keras implementation of the activation layers with the **ReLU** as the activation function (see Listing 4.3). As no additional parameters are given to the function, it will have a zero value for the negative part and no maximum value for the output.

```
1 up3 = Activation('relu')(up3)
```

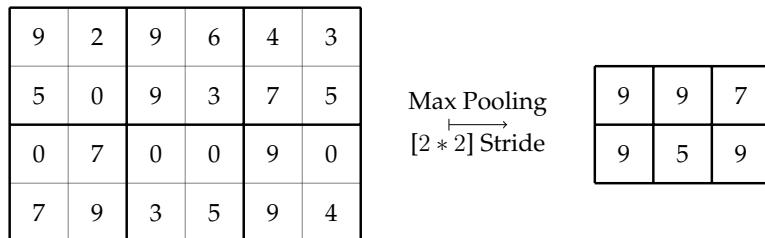
**Listing 4.3:** Activation layer used in the implementation of the project.

## Pooling

On our project, at the end of each "level" set of layers in the contracting path, the left part, a pooling layer is applied, and the reason why it is made that way is rather simple and important:

In general terms, when we are searching for a feature on an image, the absolute position is not relevant, being the relative position to other features the important characteristic. This reasoning is behind the pooling layers.

Pooling reduces the spatial size of the representation, making the number of parameters to be learnt smaller as well as the computational cost.



**Figure 4.8:** Demonstration of how max pooling works. In this example, each number of the output represents the maximum value of the corresponding  $[2 \times 2]$  pool of the input feature map.

In practice, pooling works by taking a determined set of parameters out of the feature map from the input (in both the graphical example represented in Figure 4.8 and the code example displayed in Listing 4.4, a  $(2 \times 2)$  stride), applying a function to obtain the pooled value of the new parameter and storing it on the pooled result map. There are various functions to apply when pooling, such as an average of the numbers of the pool, or in our examples, max. Max pooling works by outputting the maximum value of the input stride.

```
1 down4_pool = MaxPooling2D((2, 2), strides=(2, 2))(down4)
```

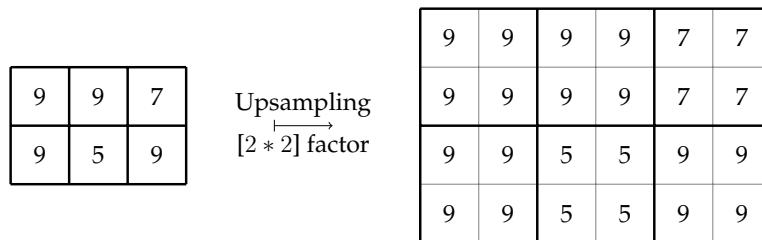
**Listing 4.4:** Pooling used on the project.

## Upsampling

```
1 up3 = UpSampling2D((2, 2))(up4)
```

**Listing 4.5:** Upsampling used in this project.

Upsampling is a process that has as the main goal to increment the size of the input data. It does it by repeating the rows and columns of the input by two specified numbers. In our case, as the pooling (or downsampling) was done by a factor of two, the upsampling is done by the same multiplier (as show in Figure 4.5). Contrary to pooling, it does not apply any specific function, just repeats the contents of the input, as can be observed in the visual example of Figure 4.9. This type of layers are applied on the expansive path.



**Figure 4.9:** Demonstration of how upsampling works. The input is the output of the pooling example of Figure 4.8. On the right part we can appreciate the information lost when doing a pooling and upsampling on the same set of features.

## Concatenation

The concatenation layer joins two feature maps by the given axis. In the Keras code implementation (see Listing 4.6), we concatenate on the expansive (or decoder) part the features coming from the upsampling below and from the last convolution at the same level on the other side, the contracting (or encoder) part (see Figure 4.2 for more information). This result in a bigger feature map, as it adds the filters obtained before the information loss that supposes contracting and more processed ones after several convolutions and activations.

```
1     up3 = concatenate([down3, up3], axis=3)
```

**Listing 4.6:** Concatenation used on the project.

### 4.3.2 U-Net Iteration

This explanation have a visual representation in Figure 4.2 and its equivalent code in Listing A.1.

On our U-Net, the convolutional layers always extract  $(3 \times 3)$  filters, and the number of them increases the closer it gets to the central part. On this level, the number of filters is 1024, decreasing by a factor of two for each step upwards.

After the convolution layer is where the main architectural difference with the original U-Net takes place [1]: a batch normalization layer. This step intended to speed up the learning process, making the learning curve much steeper.

The results of the normalization are introduced in the activation layer, which adds the **ReLU** non-linearity to the parameters.

The three aforementioned steps (convolution, batch normalization and activation layer) are repeated twice on every single level of the U-Net, with minor differences depending on the part of the network:

1. **Contracting path:** The output of applying twice the above steps is introduced on a Max Pooling layer, that, on our network, will halve the resolution before outputting to the next level. These steps are repeated until the number of filters (that increase in each level by two) reach 512, the start of the central part.
2. **Central part:** On the "valley" of the network, just the above mentioned steps are performed, duplicating for the last time the number of filters out of the convolutional layer.
3. **Expansive path:** The decoder part of our U-Net network starts with an upsampling of the output coming from the level below, as the main goal of this part is to output an image with nearly the same resolution as the one on the input. The output coming from the upsampling layer is concatenated with the output of the last convolutional layer of the same level on the contracting part (see Listing Line 34). This output have a more complex feature map, with characteristics from before the loss of information that the pooling-upsampling process generates, and a more mature set of characteristics resulting of applying several more convolution-activation layers.

## 4.4 Data augmentation

It is widely demonstrated [51] that the bigger the size of the training data, the better performance a neural network could offer. So, as general rule, maximizing the amount of input data fed to the neural network is a good practice, and this could be done in two ways: adding new datasets and/or, as we have done it, with data augmentation.

Data augmentation is a process that alters the input data in order to introduce more variations and increase artificially the data size. The code fragment contained in Listing 4.7 is a call to a method that will randomly shift, scale and rotate input images to make them slightly different.

```

1 img, mask = randomShiftScaleRotate(img, mask,
2     shift_limit=(-0.0625, 0.0625),
3     scale_limit=(-0.1, 0.1),
4     rotate_limit=(-0, 0))
5 img, mask = randomHorizontalFlip(img, mask)

```

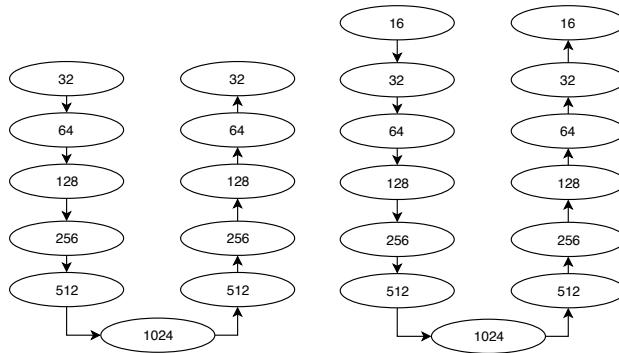
**Listing 4.7:** Data augmentation fragment code

## 4.5 Models

U-Nets (as NNs in general) need to have a fixed input dimension, and then, in order to achieve the central part and output mask dimension, a determined number of poolings and upsamplings are required. As the native dimensions of the photos vary, they are resized (normally downsampled) to meet the input size requirements of the U-Net.

In order to accommodate different input dimensions, several U-Net models are required. Architecturally there are no major differences between them, but the number of levels that they have is established with regard to its input shape. For example, a U-Net with an input size of  $(256 \times 256)$  will need five levels (five levels for the encoder part, one for the central and five levels for the decoder part) and one with  $(512 \times 512)$  will need six.

Also, in our U-Net, the number of filters on the central part of the network is constant (1024) independently of the input size. This means that every model will start and finish with a different number of filters. Referring back to the same examples as before, the initial and final number of filters for the U-Net with 256 as input size is 32, and for the 512 one is 16.



**Figure 4.10:** Example of how many levels compose different U-Net models. In that case, the left part represents the *u\_net* 256 model and the right part represents the *u\_net* 256 one. The number of filters are written inside the ellipses.

## 4.6 Loss And Metrics Functions

The main loss function used on this project is *bce\_dice\_loss*, or *Binary Cross Entropy* (**BCE**) dice loss. It is based on the dice function, that measures how well the predicted result overlaps the ground truth. This measurement is represented on the  $[0, 1]$  interval, being 1 the perfect overlapping.

The variation used that adds the **BCE** to the dice functions simply pushes the results towards the extremes of the interval. The formulas below contain

the equations for the loss function used in this project. They are reproduced from [52].

$$\begin{aligned} LOSS &= BCE - \ln(DICE) \\ BCE &= - \sum_i (y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i)) \\ DICE &= 2 \frac{\sum_i y_i p_i}{\sum_u y_i + \sum p_i} \end{aligned}$$



# Chapter 5

## Experiments

*This chapter contains the experiments conducted during this bachelor thesis. Section 5.1 briefly summarizes the environment used to train and test the neural network. Section 5.2 explains the training and testing methodology followed during this project. Finally, Section 5.3 presents the training and test results obtained during this work.*

### 5.1 Introduction

This Chapter contains all the information related to the testing done during this project. This introduction will explain briefly the implementation methodology of our [NN](#). Additionally, it also contains the reasons why we selected the used dataset and its description.

The network was developed using Keras (see Subsection 3.1.4) as the main core, supported by Tensorflow (see Subsection 3.1.3), CUDA (see Subsection 3.1.1) and cuDNN (see Subsection 3.1.2). The first steps of the training were done on the development system, but the main body of the experiments was performed on the DTIC's Asimov server, with its characteristics listed in Sub-section 3.3.

The main dataset used to perform the training and experiments is Unite the People [19] UPi-S1h variation, with more than 10.2k images after preprocessing (see Section 4.2) it to fit the desired form that will enable us to train the network. This database was selected because it is specifically oriented to human segmentation and contains different environments, occlusion and lighting situations. During all the experimentation, the dataset was randomly divided in two sets, one for training and one for validation, representing the 90% and 10% (9256 and 1029) of the original dataset, respectively. All the training process was done using the *bce\_dice\_loss* as the loss function (see Section 4.6). Additionally, we also recorded a test sequence to test the proposed approach with our own data.

### 5.2 Training and Testing Methodology

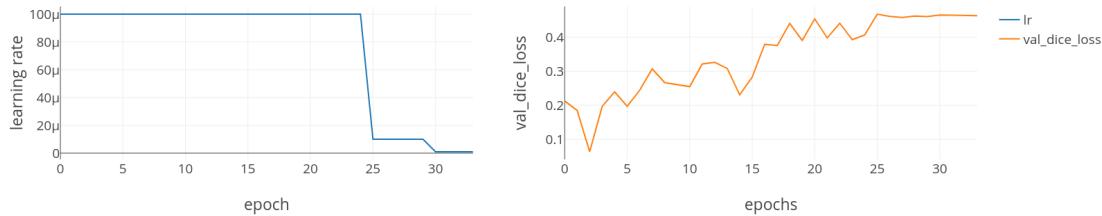
The main core of the experimentation was done in the server described in Section 3.3 using Docker (see Subsection 3.2.2). All the packages and frameworks are described on Chapter 3.

Training is a rather simple process once the environment is properly setted up. All the raw input data must be on the *train* directory, and the masks should

be on *train\_masks* folder. As all the input data is stored in one folder, the test set is specified on a CSV table in the *test* directory that contains all file names of the images in the validation set.

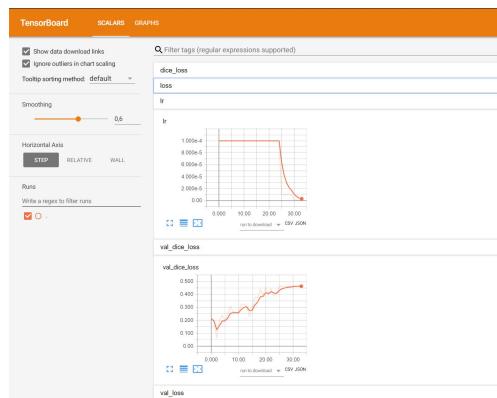
The Python file *params.py* specifies the model selected to train, the image input size (that should be equal to the model input size), the batch size, maximum number of epochs, threshold and an experiment title.

Once the desired parameters are set, in order to train, the *train.py* is executed. As it is, it will display how the time, loss, dice loss and epochs are progressing during the training of the network. The training will end when the maximum number of epochs is reached or when the *EarlyStopping* callback detects various consecutive epochs with no *val\_dice\_loss* significant improvement (see Section 3.1.4 for more information about Keras Callbacks). During training, another Keras Callback could be activated, *ReduceLROnPlateau*. This will reduce the learning rate when stops to detect improvements in *val\_dice\_loss*.



**Figure 5.1:** Learning rate decay due to the activation of *ReduceLROnPlateau*. On the top right corner, the stabilization of *val\_dice\_loss* could be appreciated, what leads to the activation of *ReduceLROnPlateau*, callback that reduces the learning rate. The stabilization of the loss also leads to the activation of *EarlyStopping*, that ends the training.

Moreover, thanks to another callback, a TensorBoard compatible log file is generated during the training progress. This logs can be visualize and provide information about the training.



**Figure 5.2:** TensorBoard log visualization. Same variables as in Figure 5.1.

Another file generated by Callbacks is a CSV table containing information about the training, epochs, loss values, etc. This file provides the same information as the TensorBoard log. These CSV tables are used to create graphs such as the one on Figure 5.1<sup>1</sup>.

However, the most important file generated during the training process is the one containing the weights of the training model. This file contains the best weights obtained during training, and it is used later for inference using validation and test images.

## 5.3 Experiments

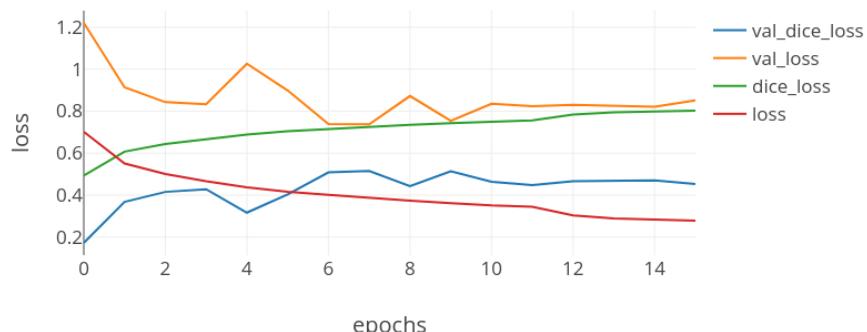
This section will discuss the training and experiments performed during the duration of this project and the obtained results.

### 5.3.1 Training

As explained in Section 4.5, the implemented NN has been modified to support different models with various input and output sizes. These models have no major architectural differences, just more repeated steps and granularity for each iteration (see Figure 4.10). Apart from the models, there are more parameters and variations that affect the experiments, such as batch size, early stopping, etc.

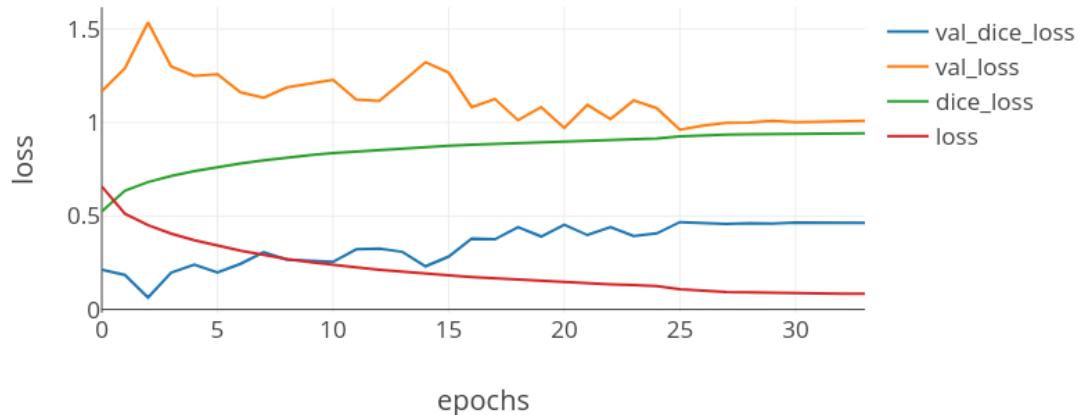
The first batch of experiments was performed under the same conditions: all with EarlyStopping and ReduceLROnPlateau (see Subsection 3.1.4 for explanation, Figure 5.1 for learning rate decay and Listing 3.3 for example code and used parameters). The default loss function for all models is *bce\_dice\_loss* (combination of *Binary Cross Entropy* (BCE) and DICE score) and the metric function is *dice\_loss*. The function used to reduce the learning rate and activate early stopping is *val\_dice\_loss*. (see Section 4.6)

This set of tests contains training and experimentation for models *unet\_128*, *unet\_256*, *unet\_512* and *unet\_1024*. The results are displayed on the following figures:

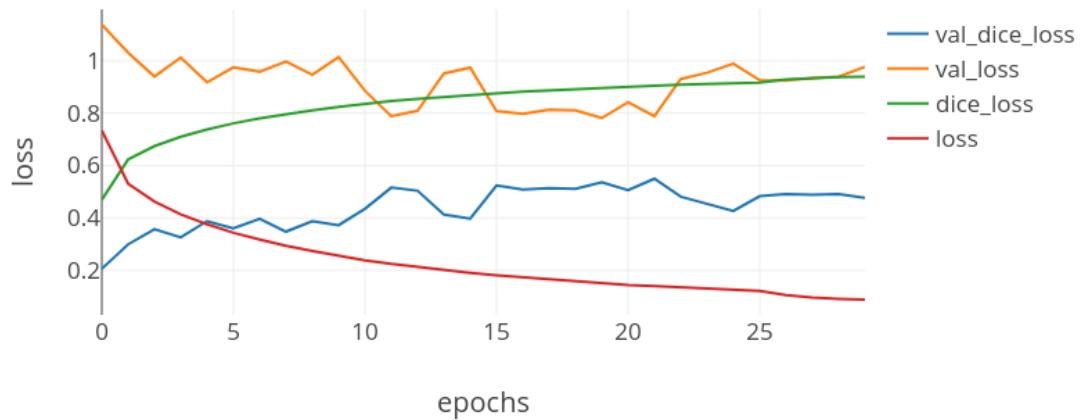


**Figure 5.3:** Loss functions and dice score evolution during training and validation of the *unet\_128* model.

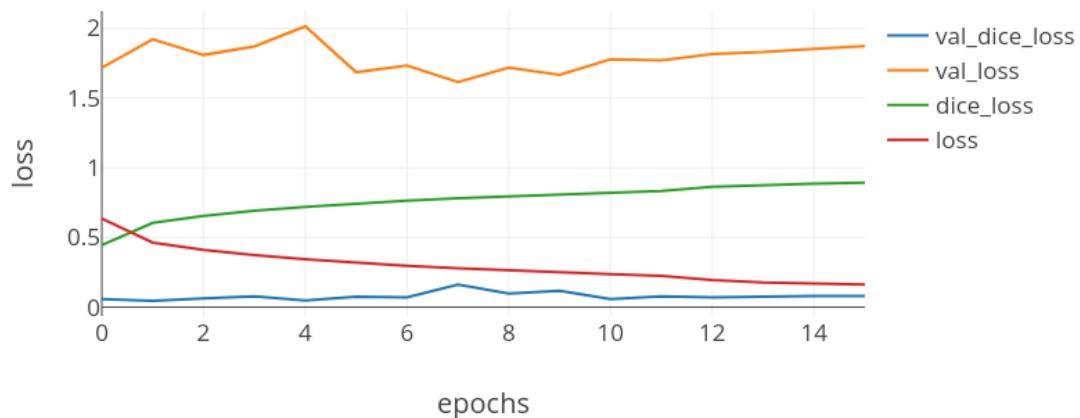
<sup>1</sup>This set of graphics were generated using Plotly online tool <https://plot.ly>.



**Figure 5.4:** Loss functions and dice score evolution during training and validation of the *unet\_256* model.



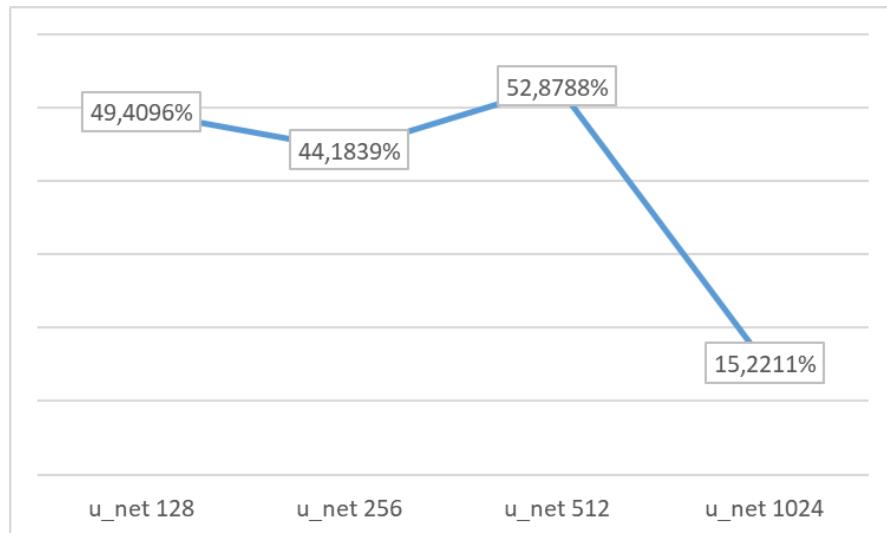
**Figure 5.5:** Loss functions and dice score evolution during training and validation of the *unet\_512* model.



**Figure 5.6:** Loss functions and dice score evolution during training and validation of the *unet\_1024* model.

The four above included Figures present the obtained results for the training of the four different models. The figures representing *unet\_128* (5.3), *unet\_256* (5.4) and *unet\_512* (5.5) have a similar training progress until they arrive to a *val\_dice\_loss* stabilization, where they stop training.

The last Figure (5.6) that represents the training process for the *unet\_1024* model is visibly different of the rest, with a much lower and constant *val\_dice\_loss*. This anomaly could be also observed on the accuracy results of these models (Figure 5.7).



**Figure 5.7:** Accuracy obtained during the first experiments.

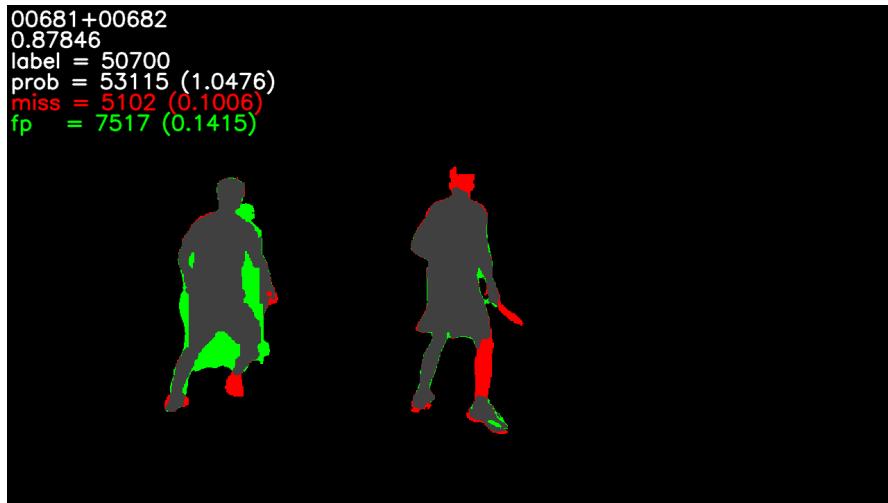
This table shows that the better dice score is the one of the *unet\_512* model. As we would see in the next section, this result is rather imprecise. Additionally, this next section will contain visual examples of how our systems work.

### 5.3.2 Results

This subsection presents some results produced by our approach. This method uses the weights file generated during training, it takes as an input the test image set and segments the images. If the segmented images have a ground truth mask, we can an image as shown in the figure (5.8) below to see how well our method performed.

This set of images have four important concepts:

- **True positive (TP):** These are the pixels correctly segmented as foreground. They are drawn in **gray**.
- **False positive (FP):** These are the pixels falsely segmented as foreground. They are drawn in **green**.
- **True negative (TN):** These are the pixels correctly segmented as background. They are drawn in **black**.
- **False negative (FN):** These are the pixels falsely segmented as background. They are drawn in **red**.

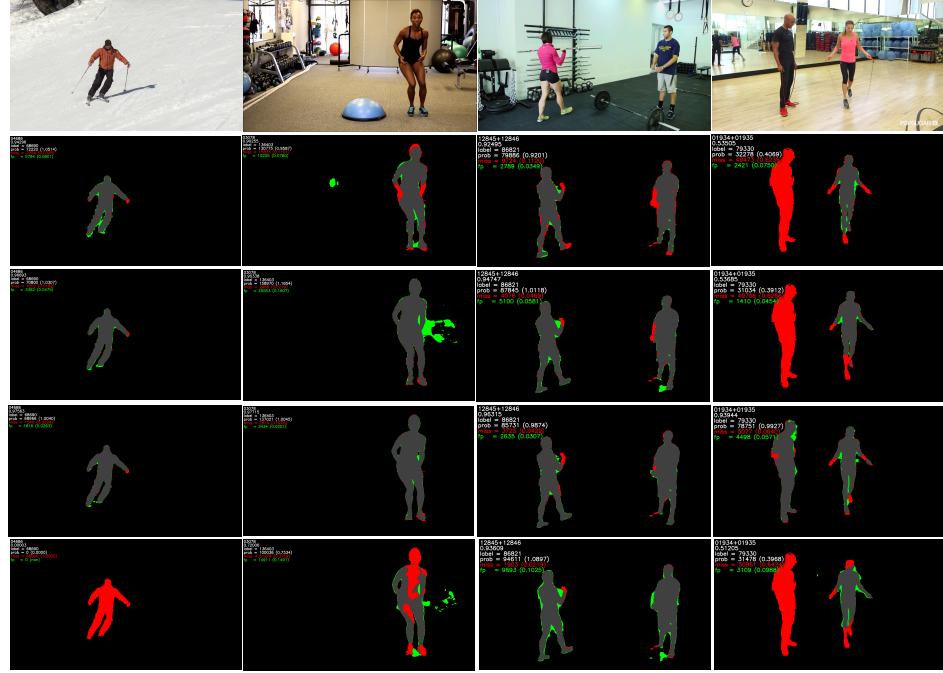


**Figure 5.8:** Example of how the segmentation results are going to be displayed.

Additionally, on the top left corner of this set of images, there is important information: the name of the segmented image, the accuracy, the total number of pixel that are part of human figure, the total number of pixels that the system thinks are part of human, the false negative pixels and the false positive pixels, respectively.

The page below contains two figures (5.9 and 5.10). They both display segmentation results of the test set. The first row in each figure shows the input data, and the results below display the segmentation performed with the weights generated by *unet\_128*, *unet\_256*, *unet\_512* and *unet\_1024* models, respectively. Figure 5.9 contains four images with very good segmentation accuracy. The first two columns contain single human images. The two right columns display two human figures. The best obtained accuracy results in all four of them are the ones out of the *unet\_512* model. In general, images containing a human figure in a standing position obtain great accuracy (+85%).

As a curious note, on the right column, the only model that detected the second human was the *unet\_512*.



**Figure 5.9:** Good accuracy examples. Two left columns of individual humans, and the two right ones of two humans. From top to bottom, raw data input and results from *unet\_128*, *unet\_256*, *unet\_512* and *unet\_1024* models.



**Figure 5.10:** Bad results. First two rows display database errors and the other ones show bad accuracy examples. From top to bottom, raw data input and results from *unet\_128*, *unet\_256*, *unet\_512* and *unet\_1024* models.

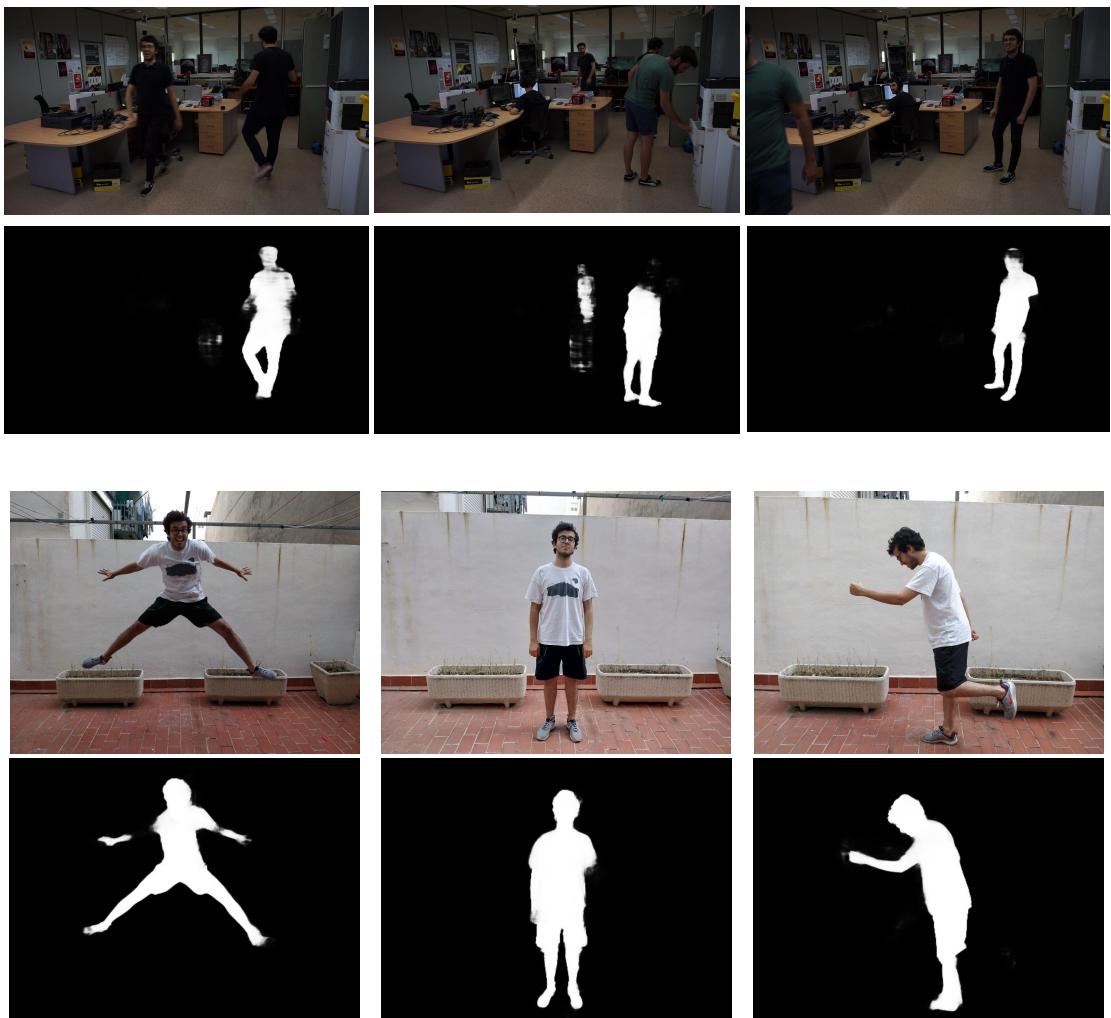
On the other side of the coin, we have examples of bad accuracy results on Figure 5.10. The left half of the figure displays errors in the dataset: in some cases, the dataset does not mask all humans in an image, producing a double negative effect. The network does not train properly as a human (or several) instance is not identified as it should be, making the network think that a human figure is not, with all the bad consequences that it could have in the learning process. The other and more direct negative impact is that, once you have your model trained and you are ready to perform the prediction over the test set, the accuracy obtained is not totally correct. For example, on the left column, the human in the back wearing orange and the most predominant human figure on the image, the central woman, are not masked as humans. The same happens on the second left column, where the second biggest human figure, is also not masked. However, our system manage to segment properly the central woman in the first column and the not-masked girl on the second column.

The other examples on the right half of Figure 5.10 are bad segmentation results. The image with a horse on it seems to work in a different way for every model that has processed it. The *unet\_128* model predicts both the woman and the horse, the *unet\_256* one only segments the woman (obtaining the best result for this image), the *unet\_512* model only segments the horse and *unet\_1024* model does not even segment anything.

Lastly, on the right column, a mix of our system's flaws are displayed: the ability to detect all the subjects on a large group and to segment little or occluded human figures. The first flaw could be also seen on the two left columns. But, even without detecting two of the human figures, the *unet\_512* model achieves a 85% of accuracy for the last image.

### 5.3.3 Testing The System With Our Images

The tests described on this subsection are done using our own taken pictures. As we have already demonstrated, the best obtained results are the ones produced by the *unet\_512* model, so we used the weights outputted by this model. As we do not have ground truth segmentation for these images, the results are just the mask predicted by our system.



**Figure 5.11:** Masks generated with *unet\_512* model for our pictures.

The results of these tests presented the same characteristics as the ones of the Subsection above (5.3.2): some images are segmented nearly perfect and, in some examples, when multiple people are present, the system does not detect all of them. In these examples it could caused due to the low exposure of the images and the mix between dark clothing and environment. Additionally, the system sometimes fails to segment certain body parts (see the bottom right corner images).



# Chapter 6

## Conclusions

*This last chapter of the thesis contains the final thoughts and conclusions. Section 6.1 gives a final summary of this work. Section 6.1 explains the obtained results of this project. Section 6.3 reflects the most remarkable objectives achieved. Section 6.4 overviews what this project has supposed personally for me and lastly, Section 6.5 sets the direction of future research and development related to the implemented system.*

### 6.1 Conclusions

In this thesis, we have implemented a system capable of segmenting human figures in many different positions. The system is composed by a **SOTA** U-Net **CNN** that enables us to obtain nearly perfect results for part of the images from the validation dataset. This system was developed after conducting a research about the current **SOTA DL** related to image segmentation that confirmed that the best approach to this field is the use of **CNNs**. We have also performed experiments to evaluate the performance of our system, testing different U-Net models.

### 6.2 Results

The obtained results show a mix between excellent and poor performance. The excellent performance was obtained on a variety of human poses, being the best segmented one the standing position. Also, as the system only works with RGB images, it is highly dependant of lightning and colours, performing significantly better in environments with regular lightning, good contrast and no occlusion.

On the other side of the coin, the bad results are achieved under certain circumstances: bad lighting or exposure, occlusion and cropped or small human figures. All these problems could be improved or solved with the addition of the proposals explained in Section 6.5.

### 6.3 Highlights

The main highlights and contributions of this project are listed below:

- Dataset adaptation to fit the class segmentation model.

- Current **SOTA** research about **DL** and image segmentation related fields.
- Familiarization with **DL** tools and frameworks.
- U-Net *Convolutional Neural Network* (**CNN**) development and refinement for human shapes in images segmentation.
- Use of Docker to run the **DL** algorithms on a high performance **GPU** server.

## 6.4 Personal

This final Bachelor's Thesis has been my first proper contact with **DL** and Computer Vision, so I have tried to absorb as much information as I could during the development of this project and from my advisors. I also completed the "Deep Learning" course from Udacity<sup>1</sup> in order to understand many related concepts and theoretical background.

## 6.5 Future work

Due to the time constraints of the project and how tedious the experiments were, there are additional tasks that are left as a future work:

- We have used only single training dataset containing over 10.000 images. Considering the wide variety of shapes the human body could be presented, additional training data could have made a difference. On a side note, some database flaws (see Subsection 5.3.2 for more information) were discovered in a late stage of the project, what prevent us for trying other datasets.
- Bigger input size model are supposed to perform better, but, surprisingly, *unet\_1024* model did not. Extensive debugging and testing was performed but the cause of its malfunction could not be diagnosed. Future works should be centred on trying to figure out how to solve this problem.
- As the research on the current **SOTA** showed, there is a trend nowadays focused on adding more input information layers, such as depth or infrared images. This approach could have been really interesting as the results shown in that field(see Subsection 2.5.2 for more information) were really promising.
- Additional experiments on hyperparameter tuning, evaluation of different loss functions, other architectures, etc.

---

<sup>1</sup><https://www.udacity.com/course/deep-learning-ud730>

# Appendix A

## U - NET source code

```
1 down3 = Conv2D(256, (3, 3), padding='same')(down3)
2 down3 = BatchNormalization()(down3)
3 down3 = Activation('relu')(down3)
4 down3_pool = MaxPooling2D((2, 2), strides=(2, 2))(down3)
5 # 20
6     down4 = Conv2D(512, (3, 3), padding='same')(down3_pool)
7     down4 = BatchNormalization()(down4)
8     down4 = Activation('relu')(down4)
9     down4 = Conv2D(512, (3, 3), padding='same')(down4)
10    down4 = BatchNormalization()(down4)
11    down4 = Activation('relu')(down4)
12    down4_pool = MaxPooling2D((2, 2), strides=(2, 2))(down4)
13 # 10
14     center = Conv2D(1024, (3, 3), padding='same')(down4_pool)
15     center = BatchNormalization()(center)
16     center = Activation('relu')(center)
17     center = Conv2D(1024, (3, 3), padding='same')(center)
18     center = BatchNormalization()(center)
19     center = Activation('relu')(center)
20 # center
21 up4 = UpSampling2D((2, 2))(center)
22 up4 = concatenate([down4, up4], axis=3)
23 up4 = Conv2D(512, (3, 3), padding='same')(up4)
24 up4 = BatchNormalization()(up4)
25 up4 = Activation('relu')(up4)
26 up4 = Conv2D(512, (3, 3), padding='same')(up4)
27 up4 = BatchNormalization()(up4)
28 up4 = Activation('relu')(up4)
29 up4 = Conv2D(512, (3, 3), padding='same')(up4)
30 up4 = BatchNormalization()(up4)
31 up4 = Activation('relu')(up4)
32 # 20
33 up3 = UpSampling2D((2, 2))(up4)
34 up3 = concatenate([down3, up3], axis=3)
35 up3 = Conv2D(256, (3, 3), padding='same')(up3)
36 up3 = BatchNormalization()(up3)
37 up3 = Activation('relu')(up3)
```

**Listing A.1:** Code corresponding to the central part of the U-Net. This code has its visual representation on the figure 4.2.

## A.1 Git Repository

The source code of this Bachelor's Thesis is available in *BitBucket*<sup>1</sup>. The repository contains a Python project with the following structure:

```
tfg_mmartinez_src
├── dice
├── epochs
├── experiments
├── logs
├── model
├── preprocess
├── test
├── train
├── train_masks
├── weights
├── params.py
├── predict.py
├── script.py
└── train.py
├── tensorflow.yml
└── README.txt
```

## A.2 Dependencies

The first step in the deployment is the installation of CUDA. Then, the project needed packages are listed on the *tensorflow.yml* file included in the BitBucket repository. This is a ready-to-use environment for Conda. It can create a functional environment for executing the project with the following Conda command:

```
1 conda env create -f tensorflow.yml
```

**Listing A.2:** Instruction for creating a new Conda environment from the *tensorflow.yml* file.

---

<sup>1</sup>BitBucket repository containing the source code for this project: [https://bitbucket.org/3dpl/tfg\\_mmartinez\\_src/src](https://bitbucket.org/3dpl/tfg_mmartinez_src/src)

### A.3 Executing

The BitBucket repository contains several weights files that can be used for inference (generating segmentation mask). To predict with those files, the parameter "title" from *params.py* should be changed to the name of the weight, as well as the U-Net model and input size. For the rest of the commands, see Listing A.3.

```
1 #train
2           python3 train.py
3 #predict
4           python3 predict.py
```

**Listing A.3:** Commands for executing the project.

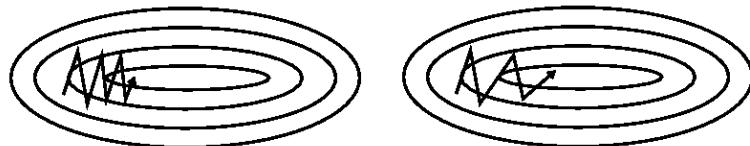


# Appendix B

## Concepts

### B.1 Momentum

One of the main problems with Deep Learning is that the learning progress could be slowed by many oscillations that, even if they are approaching the minimum, they could be doing it in an oscillating way (left part of Figure B.1). This process would definitely arrive to the global minimum, but it will take a long time.

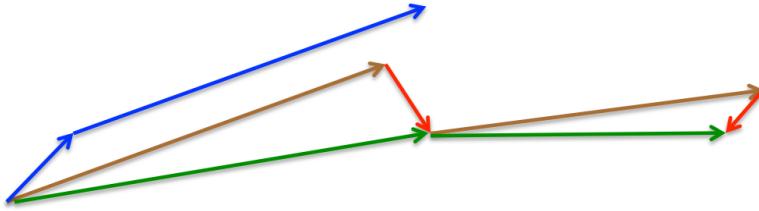


**Figure B.1:** Application of momentum to the learning process.  
Left side without momentum. Right side with momentum

The idea behind the momentum concept is to accelerate this learning process by giving the algorithm a clue of where it should be directing. This is done with a momentum term that has a value between 0 and 1. This term increase the size of the steps if they are taken towards the the minimum. This helps the learning process by smoothing out variations and oscillations and making jumps in the right direction larger.

### B.2 Nesterov accelerated gradient

This optimizer has a core idea similar to the momentum one (see Section B.1), as it aims to speed up the learning as it approaches to the optimum result, but this method extends the momentum giving it notion of where it is going. Momentum first computes the current gradient (small blue vector in Figure B.2) and then take a big jump in the direction of the updated accumulated gradient (big blue vector). Nesterov method takes a big jump in the direction of the previous accumulated gradient (brown vectors), updates the gradient and then makes a correction. This three-steps process results in the complete Nesterov update (green vector).

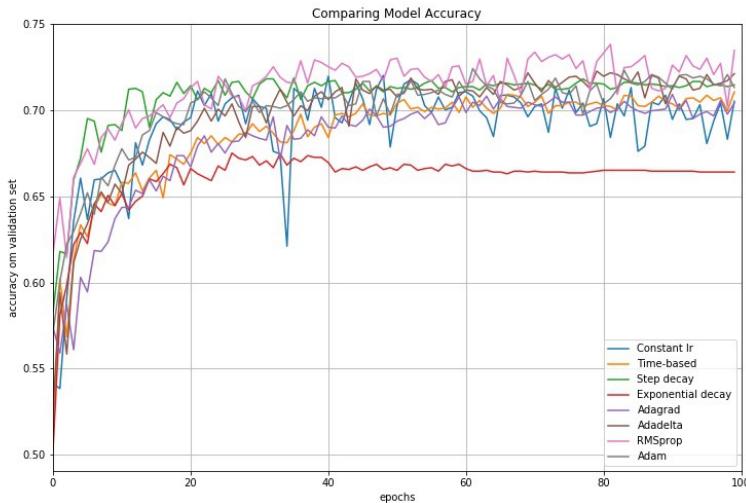


**Figure B.2:** Nesterov updates compared to normal momentum

### B.3 Learning rate

The learning rate is a hyper-parameter that controls how much the system is adjusting the weights of our network with respect to the loss, or, in other words, how fast does it learn.

When training with neural networks, it is usual to reduce the learning rate as the training goes on. There are many ways to do it, as it could be adjusted in advance with regard to a constant, time, steps or exponential functions; or it could be adaptive and adjust itself as the learning progresses. The main adaptive learning rate is the *Adagrad* one, and the majority of the rest are extensions or modifications derived from this one. It seeks to perform larger updates for more sparse parameters, and smaller updates for less sparse parameters. The following figure B.3 displays a comparison between diverse adaptive learning rate methods and learning rate schedules.



**Figure B.3:** Comparison between diverse learning rate schedules and adaptive learning methods. Figure reproduced from [53].

Keras 3.1.4 has several adaptive methods already implemented that have a default learning rate constant in its initialization that adjust how aggressive they are.



# Bibliography

- [1] O. Ronneberger, P. Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Vol. 9351. LNCS. (available on arXiv:1505.04597 [cs.CV]). Springer, 2015, pp. 234–241. URL: <http://lmb.informatik.uni-freiburg.de/Publications/2015/RFB15a>.
- [2] Korsuk Sirinukunwattana, Shan E Ahmed Raza, Yee-Wah Tsang, et al. "Locality sensitive deep learning for detection and classification of nuclei in routine colon cancer histology images". In: *IEEE transactions on medical imaging* 35.5 (2016), pp. 1196–1206.
- [3] Pim Moeskops, Max A Viergever, Adriënne M Mendrik, et al. "Automatic segmentation of MR brain images with a convolutional neural network". In: *IEEE transactions on medical imaging* 35.5 (2016), pp. 1252–1261.
- [4] Mark JJP van Grinsven, Bram van Ginneken, Carel B Hoyng, et al. "Fast convolutional neural network training using selective data sampling: Application to hemorrhage detection in color fundus images". In: *IEEE transactions on medical imaging* 35.5 (2016), pp. 1273–1284.
- [5] Itamar Arel, Derek C Rose, and Thomas P Karnowski. "Deep machine learning-a new frontier in artificial intelligence research [research frontier]". In: *IEEE computational intelligence magazine* 5.4 (2010), pp. 13–18.
- [6] David H Hubel and Torsten N Wiesel. "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex". In: *The Journal of physiology* 160.1 (1962), pp. 106–154.
- [7] Wenlu Zhang, Rongjian Li, Houtao Deng, et al. "Deep convolutional neural networks for multi-modality isointense infant brain image segmentation". In: *NeuroImage* 108 (2015), pp. 214–224.
- [8] Andre Esteva, Brett Kuprel, Roberto A Novoa, et al. "Dermatologist-level classification of skin cancer with deep neural networks". In: *Nature* 542.7639 (2017), p. 115.
- [9] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. "V-net: Fully convolutional neural networks for volumetric medical image segmentation". In: *3D Vision (3DV), 2016 Fourth International Conference on*. IEEE. 2016, pp. 565–571.
- [10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

- [11] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. "Grabcut: Interactive foreground extraction using iterated graph cuts". In: *ACM transactions on graphics (TOG)*. Vol. 23. 3. ACM. 2004, pp. 309–314.
- [12] Xiaodan Liang, Yunchao Wei, Liang Lin, et al. "Learning to segment human by watching Youtube". In: *IEEE transactions on pattern analysis and machine intelligence* 39.7 (2017), pp. 1462–1468.
- [13] Chunfeng Song, Yongzhen Huang, Zhenyu Wang, et al. "1000fps human segmentation with deep convolutional neural networks". In: *Pattern Recognition (ACPR), 2015 3rd IAPR Asian Conference on*. IEEE. 2015, pp. 474–478.
- [14] Baidu people segmentation dataset. 2013. URL: <http://www.cbsr.ia.ac.cn/users/ynyu/dataset/>.
- [15] Antonio Fernández-Caballero, José Carlos Castillo, Juan Serrano-Cuerda, et al. "Real-time human segmentation in infrared videos". In: *Expert Systems with Applications* 38.3 (2011), pp. 2577–2584.
- [16] Fuliang He, Yongcai Guo, and Chao Gao. "Human segmentation of infrared image for mobile robot search". In: *Multimedia Tools and Applications* (2017), pp. 1–14.
- [17] Sungil Choi, Seungryong Kim, Kihong Park, et al. "Multispectral human co-segmentation via joint convolutional neural networks". In: *Image Processing (ICIP), 2017 IEEE International Conference on*. IEEE. 2017, pp. 3115–3119.
- [18] Yeong-Seok Kim, Jong-Chul Yoon, and In-Kwon Lee. "Real-time human segmentation from RGB-D video sequence based on adaptive geodesic distance computation". In: *Multimedia Tools and Applications* (2017), pp. 1–13.
- [19] Christoph Lassner, Javier Romero, Martin Kiefel, et al. "Unite the People: Closing the Loop Between 3D and 2D Human Representations". In: *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. 2017. URL: <http://up.is.tuebingen.mpg.de>.
- [20] Gerard Pons-Moll, Javier Romero, Naureen Mahmood, et al. "Dyna: A Model of Dynamic Human Shape in Motion". In: *ACM Transactions on Graphics, (Proc. SIGGRAPH)* 34.4 (Aug. 2015), 120:1–120:14.
- [21] Ke Gong, Xiaodan Liang, Xiaohui Shen, et al. "Look into Person: Self-supervised Structure-sensitive Learning and A New Benchmark for Human Parsing". In: *CoRR* abs/1703.05446 (2017). arXiv: [1703.05446](https://arxiv.org/abs/1703.05446). URL: <http://arxiv.org/abs/1703.05446>.
- [22] Cristina Palmero, Albert Clapés, Chris Bahnsen, et al. "Multi-modal RGB–Depth–Thermal Human Body Segmentation". In: *International Journal of Computer Vision* 118.2 (2016), pp. 217–239. ISSN: 1573-1405. DOI: [10.1007/s11263-016-0901-x](https://doi.org/10.1007/s11263-016-0901-x). URL: <https://doi.org/10.1007/s11263-016-0901-x>.

- [23] M. Everingham, L. Van Gool, C. K. I. Williams, et al. "The Pascal Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88.2 (June 2010), pp. 303–338.
- [24] Tsung-Yi Lin, Michael Maire, Serge Belongie, et al. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.
- [25] CUDA Nvidia. "Compute unified device architecture programming guide". In: (2007).
- [26] L Lopez-Diaz, D Aurelio, L Torres, et al. "Micromagnetic simulations using Graphics Processing Units". In: *Journal of Physics D: Applied Physics* 45.32 (2012), p. 323001. URL: <http://stacks.iop.org/0022-3727/45/i=32/a=323001>.
- [27] David Luebke. "GPU architecture: Implications & trends". In: *SIGGRAPH 2008: Beyond Programmable Shading Course Materials* (2008).
- [28] Martín Abadi, Paul Barham, Jianmin Chen, et al. "TensorFlow: A System for Large-Scale Machine Learning." In: *OSDI*. Vol. 16. 2016, pp. 265–283.
- [29] Ashish Bakshi. *TensorFlow Tutorial Deep Learning Using TensorFlow*. URL: <https://www.edureka.co/blog/tensorflow-tutorial/>.
- [30] François Chollet et al. *Keras*. 2015.
- [31] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: COURSERA: *Neural networks for machine learning* 4.2 (2012), pp. 26–31.
- [32] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).
- [33] Timothy Dozat. "Incorporating nesterov momentum into adam". In: (2016).
- [34] *TensorBoard: Visualizing Learning*. 2018. URL: [https://www.tensorflow.org/programmers\\_guide/summaries\\_and\\_tensorboard](https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard).
- [35] NumPy Developers. "NumPy". In: *NumPy Numpy*. Scipy Developers (2013).
- [36] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, et al. "Scikit-learn: Machine learning in Python". In: *Journal of machine learning research* 12.Oct (2011), pp. 2825–2830.
- [37] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, et al. "scikit-image: image processing in Python". In: *PeerJ* 2 (2014), e453.
- [38] W McKinney. "Pandas, python data analysis library. 2015". In: *Reference Source* (2014).
- [39] John D Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in science & engineering* 9.3 (2007), pp. 90–95.
- [40] Gary Bradski and Adrian Kaehler. "OpenCV". In: *Dr. Dobbs journal of software tools* 3 (2000).
- [41] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (2014), p. 2.

- [42] *What is DOCKER?* URL: <https://www.redhat.com/en/topics/containers/what-is-docker>.
- [43] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully convolutional networks for semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 3431–3440.
- [44] Mykhaylo Andriluka, Leonid Pishchulin, Peter Gehler, et al. "2D Human Pose Estimation: New Benchmark and State of the Art Analysis". In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2014.
- [45] Sam Johnson and Mark Everingham. "Clustered Pose and Nonlinear Appearance Models for Human Pose Estimation". In: *Proceedings of the British Machine Vision Conference*. doi:10.5244/C.24.12. 2010.
- [46] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation". In: *CoRR* abs/1511.00561 (2015). arXiv: <1511.00561>. URL: <http://arxiv.org/abs/1511.00561>.
- [47] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: <1502.03167>. URL: <http://arxiv.org/abs/1502.03167>.
- [48] Olav Nymoen. *Batch normalized LSTM for Tensorflow*. 2016. URL: <http://olavnymoen.com/2016/07/07/rnn-batch-normalization>.
- [49] Balázs Csanad Csaji. "Approximation with artificial neural networks". In: *Faculty of Sciences, Etvs Lornd University, Hungary* 24 (2001), p. 48.
- [50] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. "Improving deep neural networks for LVCSR using rectified linear units and dropout". In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 8609–8613.
- [51] Franck Dernoncourt, Ji Young Lee, zlem Uzuner, et al. "De-identification of Patient Notes with Recurrent Neural Networks". In: *CoRR* abs/1606.03475 (2016). arXiv: <1606.03475>. URL: <http://arxiv.org/abs/1606.03475>.
- [52] Kaggle Team. *Carvana Image Masking Challenge1st Place Winner's Interview*. Dec. 2017. URL: <http://blog.kaggle.com/2017/12/22/carvana-image-masking-first-place-interview/>.
- [53] Suki Lau. *Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning*. 2017. URL: <https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1>.