Universitat de Girona
Escola Politècnica Superior

Projecte fi de grau

Estudi: Grau en Enginyeria Informàtica

Títol: Detecting colorectal polyps in colonoscopy:
What can deep learning do?

Document: Memòria

Alumne: Laura Galera Alfaro

Tutor: Xavier Lladó Bardera
Departament: Arquitectura i Tecnologia de Computadors
Àrea: Visió per computador i robòtica

Tutor extern: Pierre Baldi
Departament: Institute for Genomics and Bioinformatics
Àrea: Deep Learning

Convocatòria (mes/any): Juny 2022

Bachelor's thesis

Detecting colorectal polyps in colonoscopy: What can deep learning do?

Laura Galera Alfaro

June 2022

Bachelor's degree in Computer Engineering

Advisors:

Dr. Pierre Baldi
University of California, Irvine
Institute for Genomics and Bioinformatics

Dr. Xavier Lladó Bardera
Universitat de Girona
Departament d' Arquitectura i Tecnologia de Computadors

Amin Tavakoli, MSc.
University of California, Irvine
Institute for Genomics and Bioinformatics

Acknowledgements

I would like to express my most sincere gratitude to Dr. Pete Balsells and Dr. Roger Rangel for awarding me a Balsells Mobility Fellowship to conduct my bachelor's thesis at the University of California, Irvine. I would have never imagined myself meeting and learning from so many bright minds.

I am deeply grateful to Dr. Pierre Baldi for welcoming me at the Institute for Genomics and Bioinformatics and for his invaluable guidance. His unassuming approach to research and science is a source of inspiration. I also want to express my warmest gratitude to Amin Tavakoli, for all of the kind words and assistance he has provided. This thesis would not have been possible without them.

I would like to thank my advisor Dr. Xavier Lladó for monitoring my progress and for his valuable advice. It has been a pleasure to work under his guidance.

My sincere thanks go to the new friends I have made in California who have helped me to strike a balance with life outside the lab. I will never forget the Spanish dinners, playing flamenco songs, or the gatherings at the beach with the students of IGB.

Lastly, my parents deserve endless gratitude for their unconditional, unequivocal, and loving support. My accomplishments and success are because they believed in me. Deepest thanks to my brothers, who keep me grounded, remind me of what is important in life, and always take care of me. I also want to thank Connie and Bob for their generous hospitality.

I hope, from the bottom of my heart, that God brings joy to all them.

Contents

List of abbreviations

ADR	Adenoma detection rate
AI	Artificial intelligence
AUC	Area under curve
CAD	Computer-aided diagnosis
CNN	Convolutional Neural Network
CRC	Colorectal cancer
DNN	Deep neural network
FN	False negative
FP	False positive
ML	Machine learning
NBI	Narrow band imaging
NLP	Natural language processing
PPV	Positive Predictive Value
ROC	Receiver operating characteristic
TN	True negative
TNR	True Negative Rate
TP	True positive
TPR	True Positive Rate

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem statement

Colorectal cancer (CRC) is the third leading cause of cancer-related deaths in men and women, and the second most common cause of cancer deaths in the world when men and women are combined, with nearly 1 million patients who die every year [H.Sung 2021]. CRC is characterized by the unchecked division of abnormal cells in the colon or rectum. Most of the time it begins as a polyp, which is a noncancerous growth that develops in the mucosal layer of the colon or rectum. Once a polyp progresses to cancer, it can grow into the wall of the colon or rectum where it may invade blood or lymph vessels that carry away cellular waste and fluid (Figure 1.1). Cancer cells are then spread to other organs and tissues, forming tumors [Society 022].

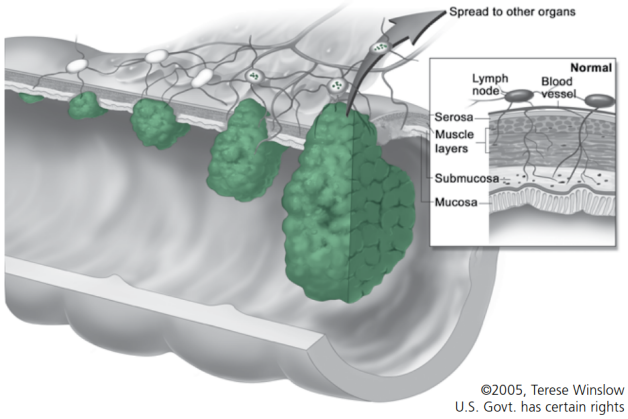


Figure 1.1: Stages of colorectal cancer growth originated by a polyp. Source: Terese Winslow, U.S. Govt.

The death rate (the number of deaths per 100,000 people per year) from colorectal cancer has been dropping for several decades as a result of performing early screening tests on potential patients [Society 022]. Colonoscopy is

the gold-standard screening procedure to inspect the large intestine. It is performed using a hand-held flexible tube device called a colonscope, which has a high definition camera mounted at the tip of the scope. The visual data that the camera feeds to the screen helps to evaluate, biopsy, and remove mucosal lessions (Figure 1.2). With such immense utility, colonoscopy has moved at the forefront of making colorectal cancer an early detected disease [Stauffer 022].

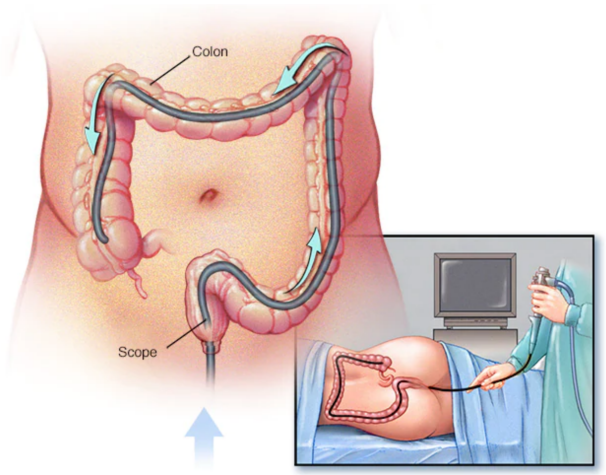


Figure 1.2: During a colonscopy, the doctor inserts a colonoscope into the rectum to check the entire colon. Source: Mayo fundation for medial education and research.

Despite these improvements, seven to nine percent of colorectal cancers still occur due to missed polyps or incompletely removed polyps during colonscopy [S.J. Winawer 1993]. Adenoma detection rate (ADR; percentage of screening colonoscopies with at least one adenoma found) is a reportable rate of the endoscopist's ability to find adenomas, the most prevalent precancerous polyp, and is inversely related to the risk of interval colorectal cancer [Liem 2018]. Unfortunately, ADR varies widely (7% - 53%) among colonoscopies [D.A. Corley 2014]. ADR depends mostly on the skill and experience of the colonoscopist, as well as characteristics of each patient, and procedural factors [Moreno 2018].

Several novel technologies have been developed to improve ADR. For instance, advancements in endoscope design, developments in accessories and new image enhancement techniques (Figure 1.3). Not only the introduction of sophisticated machinery [Bond 2015], but the developments in artificial intelligence (AI), and specially deep convolutional networks, have made computer-aided diagnosis (CAD) a promising path towards medical automation.

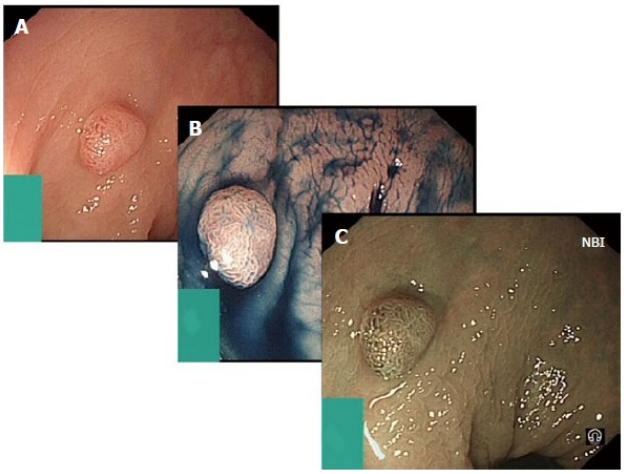


Figure 1.3: Digital chromoendoscopy. Advances in endoscope technology that manipulate wavelengths of the light to accentuate lesion characteristics. Source: World J Gastrointest Endosc.

However, there are some drawbacks that cast doubt on the capability of CAD and its adaptation to the medical system. First of all, some methods are built from a theoretical model of polyp appearance [Tajbakhsh 2015], and therefore limited to certain polyp morphologies which may not correspond to the scene where polyp appearance varies greatly. Secondly, AI systems are trained to solve only one single and narrow task. In contrast to a human endoscopist, these systems cannot use holistic information about the patient to elaborate a final diagnosis, reflecting the idea of weak AI [Wittenberg 2020]. Thirdly, many models do not consider the presence of other elements such as folds or blood vessels that can affect their performances [Bernal 2015]. Last but not least, many methods have been trained and tested on good quality image frames which might cause instability when working with real time visibility conditions.

1.2 Project objectives

The main objective of this project is to use deep learning methods to train a system capable of detecting polyps on colonoscopy images to test the ability of computer-assisted image analysis. To this end, the gradual achievements that must be accomplished are:

- Acquiring a thorough knowledge of deep learning theory and their working applications.

- Analyzing data from colonoscopies and understanding its most important features.
- Developing a CAD system to detect polyps on colonoscopy images.
- Optimizing the parameters involved in training.
- Comparing results from different approaches and combining them to get a more accurate model.

1.3 Statement of originality

The thesis presented here for examination for a BSc degree from the University of Girona is solely my own work and it was developed in collaboration with the Institute for Genomics and Bioinformatics at the University of California, Irvine. No other sources than those mentioned in the text and its references have been used in creating it.

1.4 Personal motivation

I wanted this project to be a combination of two personal interests. On one hand, I studied computer engineering because I have always been intrigued by the human reasoning, and found in machines a new paradigm for exploring this area. Also, I have been accepted to a master program in Artificial Intelligence at Stockholm University. Hence, this project was a nice way to start learning more advanced concepts of AI.

On the other hand, I am very interested in everything related to health and nutrition. I find it fascinating how the body heals and fights disease, as well as how genetic affects health. That is the reason why I chose the Institute for Genomics and Bioinformatics as the group to help me conduct this project. Besides the use big companies do of AI, I wanted to explore its applications in Science.

1.5 Outline

This project is structured into multiple chapters that provide a guidance to the problem of detecting polyps in colonscopy images:

- **Introduction:** First chapter presents the problem of detecting polyps in colonoscopy and its relation with colorectal cancer. It states the objectives expected to be accomplished, the personal motivation behind it and the structure followed in this document.
- **Feasibility study:** A hypothetical study of the technical, economical and legal part of the project.
- **Methodology**: It offers a description of the methodology used in the development of this project.
- **Thesis planning:** This chapter shows the distribution of tasks bearing in mind the available time.
- **Background theory:** The aim of this chapter is to provide a brief explanation of the theoretical concepts to understand the experiments.
- **Studies and decisions**: This chapter provides a description of the system requirements and presents the hardware and software necessary for the experiments.
- **Analysis and design**: The purpose of this chapter is to analyze the components of the research and propose a design that structures every part.
- **Implementation and results** This chapter is devoted to describe the software implementation, present the results from the tests realized and compare them among all the models.
- **Conclusions**: This chapter is a summary of the achievements, including the difficulties faced during the process.
- **Future work**: This chapter offers an overview of the future work that could be done to improve the project.
- **Installation manual**: This appendix is a guide about how the environment must be set up in order to run the software.

Chapter 2

Feasibility study

Before developing any project, it is important to analyze relevant factors to ascertain the likelihood of completing the project successfully.

The challenge of machine learning (ML) systems is the difficulty of estimating how complex it is to develop, deploy and maintain a model. Nevertheless, since this project is merely based on a research component, the study is focused on whether it is feasible to solve the problem satisfactorily using deep learning with the available data and resources.

2.1 Technical study

Everything starts with research, a living part that evolves and solidifies as further investigation is carried out. Although it is not possible to ensure that all the objectives will be accomplished, there are two critical foundations that need to be considered beforehand [Kohli 2017]:

- **Defining the use case**

  An ideal use case defines a project which is specific, measurable, and achievable, and has well-defined users and value.

  Use cases for medical purposes vary considerably. Academicians may be satisfied with finding efficient answers that result in publications and further funding. In contrast, industry desires to make ML models that work in disparate online production environments.

  The outline of this project does not include the commercialization of a product or a service. As it has been mentioned before, it has an academical outcome. Therefore, the technical study is based on analyzing which technology and equipment are necessary for carrying out experiments in the most efficient way.

- **Importance of the dataset**

  Well-annotated datasets that cover the entities described in the use case, including normal cases and those with pathology ranging from very subtle

findings to very severe are crucial to training accurate, generalizable models.

Not only the quantity but also the quality of the data plays a major role in making a project like this one succeed. It is not possible for any deep learning model to learn without data. Hence, it is crucial to spend time on acquiring an ideal image dataset which has adequate data volume, annotation, truth, and reusability.

Computer-based image recognition and analysis require high-quality data. The goal is to obtain data from a wide range of patient populations so the dataset is diverse and accurately reflects representative disease states and outcomes. Frequently, many healthcare organizations contact medical experts to review and label data to guard against inaccurate labels and ensure that a dataset is meaningful, like in this project. However, depending on the imaging modality, non-clinicians trained to spot abnormalities may label images as normal or suspect and then clinicians review only the subset of suspect images.

2.2 Economical study

2.2.1 Equipment cost

The equipment cost makes reference to the amount of money spent on any hardware and software licence required in this project.

The hardware and software components are listed in Table 2.1 along with the units and costs.

Component	Units	Unit price	Cost
Computer	1	$1,322.43	$1,322.43
Software *(TensorFlow, Sklearn, etc)*	1	$0	$0
Total			**$1,322.43**

Table 2.1: Equipment cost.

Something to bear in mind is that, in an hypothetical case, a team formed by multiple researchers would be working on this project. Consequently, more equipment would be needed. At the same time, it would be great if the group disposed of a server with GPU clusters.

2.2.2 Human resources

This section shows a hypothetical case where multiple work profiles contributed to the project and how much each service cost, even though in reality everything was done by one person.

Roles:

- **Data Analyst**

  One of the responsibilities of a data analyst is to acquire data from primary or secondary data sources and maintain databases.

  The average hourly wage for a data analyst in the United States is **$35.80**.

- **Research Scientist**

  A research scientist usually has some specialized knowledge in natural language processing (NLP), statistics, computer vision, speech, or robotics and they acquire it through a Ph.D. or extensive research experience.

  The average hourly wage for a mid-level computer and information research scientist in the United States is **$61.72**.

- **Data engineer**

  A data engineer implements, tests, and maintains infrastructural components for proper data collection, storage, and accessibility. Besides working with big data, building and maintaining a data warehouse, a data engineer takes part in model deployment.

  The average hourly pay for a data engineer in the United States is **$54.98**.

Table 2.2 shows each task, together with the profile assigned to carry it out and the estimated cost.

Task	Profile	Time	Cost
Data collection	Data Analyst	30h	$1,074.00
Data pre-processing	Research Scientist	20h	$1,234.40
Data analysis and visualization	Research Scientist	20h	$1,234.40
Model development	Research Scientist	140h	$8,640.80
Model training	Research Scientist	60h	$3,703.20
Model optimization	Research Scientist	20h	$1,234.40
Deploy the model	Data engineer	50h	$2,749.00
Documentation	Research Scientist	90h	$5,554.80
Total		430h	**$25,425.00**

Table 2.2: Human resources estimated cost.

In conclusion, the estimated total cost is $25,425.00 + $1,322.43 = **$26,747.43**.

2.2.3 Legal study

Legal issues have become a significant part of medical imaging. It is especially difficult to share and distribute medical data due to privacy concerns and potential abuse of personal information. To overcome these limitations, in the last few years, research collaborations have started to promote sharing patient data thanks to de-identification methods. However, before working on projects which involve medical imaging, it is important to analyze the obligations regarding the protection of individuals and their personal data.

Regarding this project, there was no legal regulation applied because the images used were totally anonymous. In other words, it is impossible to tell to whom the image belongs, and it is impossible to guess gender, ethnicity or age. For example, in the case of working with DNA, face pictures or bone scans, it would be required to apply de-identification methods or stipulate which legal privacy was going to be followed.

Chapter 3
Methodology

This chapter describes the methodology employed in this project. Due to the strong research component, it has followed an iterative and incremental evolution which starts with the formulation of a question, and continues with carrying out experiments based on deductions until reaching a conclusion.

3.1 Methods

In this project, we have designed and trained deep convolutional neural networks (CNN) to detect polyps using a representative set of 2,000 hand labeled images from screening colonoscopies collected from over 2,000 patients. The outputs of the best models have been combined to power up the accuracy in detecting polyps. Figure 3.1 shows in which high-level tasks the research has been split into.

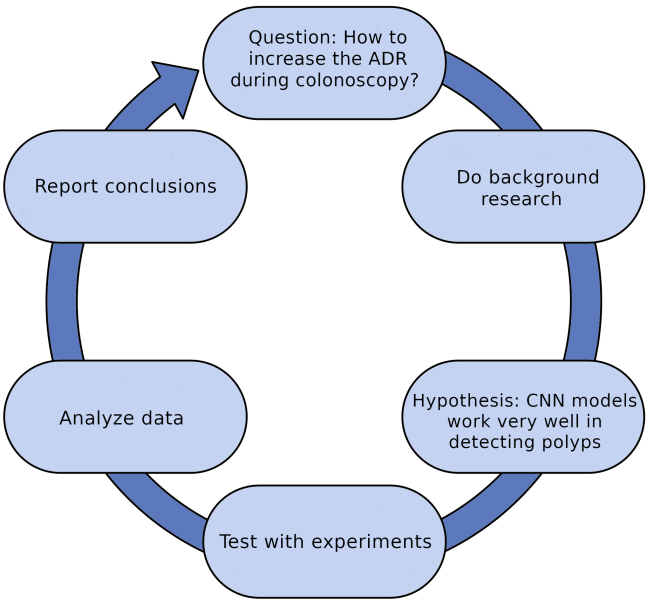


Figure 3.1: Life cycle of the research. Each phase is later concreted in multiple tasks.

3.2 Development

The methodology adhered to this project is based on developing an ongoing process which represents the previous methods. The different steps are:

1. Stating the problem and the objectives of the research.
2. Doing background research in previous publications attacking the same or similar problems.
3. Deciding on a programming language and framework.
4. Learning about the tools that are going to be used.
5. Dividing the project into smaller tasks.
6. Selecting one task.
7. Studying how to implement that part.
8. Implementing the task.
9. Questioning whether the results fulfil the expectations or not.
    - The results **are not the ones expected**. Hence, go back to point 7.
    - The results **are the ones expected**. Hence, go to point 10.
10. Storing the necessary data and representing the results in a clear and appealing way.
11. Selecting another task.
    - There **are more tasks left**. Hence, go back to point 6.
    - All the tasks **have been completed**. Hence, go to point 12.
12. Combining all the results to come up with a conclusion.
13. Writing down the report.

Figure 3.2 illustrates the previous process by making use of a diagram.

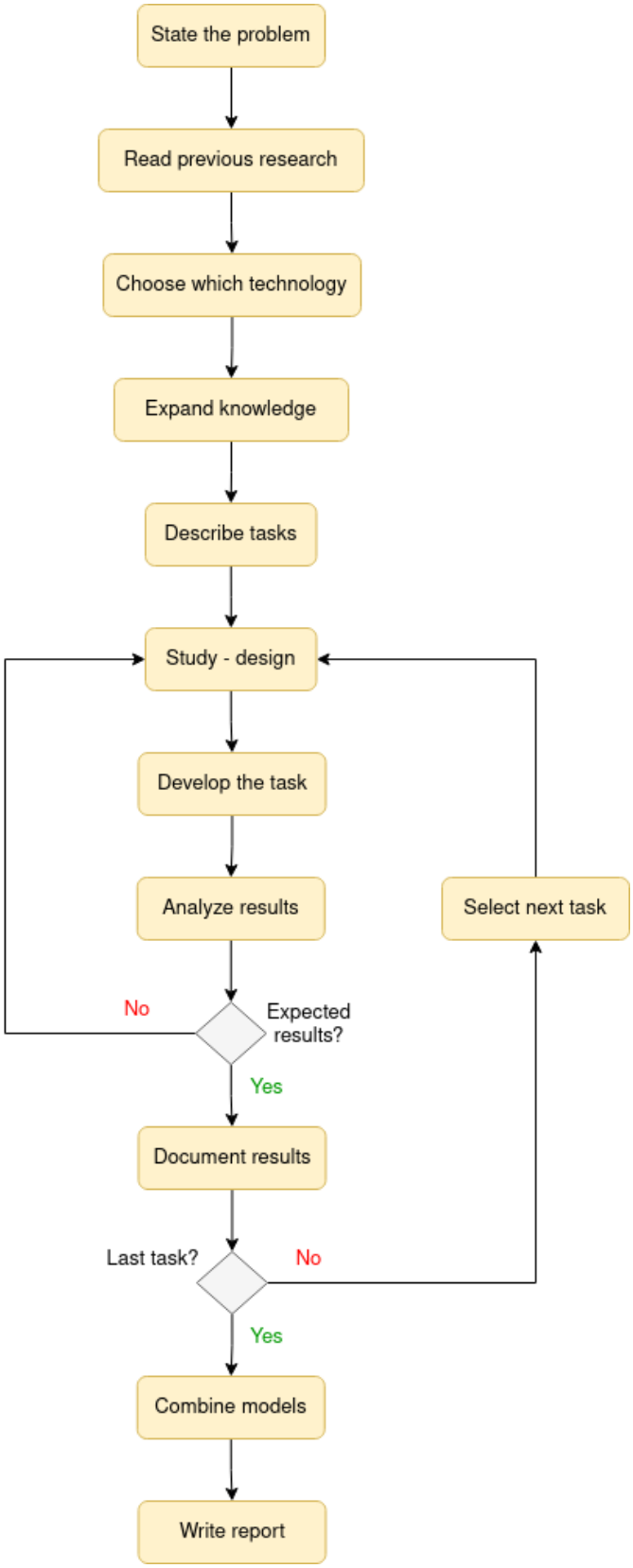


Figure 3.2: Activity diagram describing the employed methodology.

Chapter 4

Thesis planning

This thesis was developed under a Balsells Mobility Fellowship, a fellowship awarded to seven Catalan engineering students to conduct their Bachelor's or Master's thesis at the University of California, Irvine. Concretely, this research work was conducted at the Institute for Genomics and Bioinformatics, under the guidance of Dr. Pierre Baldi and Amin Tavakoli (MSc) from February 21st to June 9th, 2022.

The following subsections describe the tasks involved in this thesis and an estimated timeline table to help visualize the distribution of days in which they were expected to be completed.

4.1 Planned tasks

Searching of existing literature

Before embarking on the actual work, a week was designated to perform a thorough search of existing literature, which helped to put the proposed research in better perspective. It included reading papers and previous thesis related to polyps detection using CNN or with a similar focus.

Setting up the environment

The next week was mostly devoted to evaluate which frameworks and programming technologies could be of great use to implement deep learning models. It resulted in setting up the machine with a suitable environment before the experimental phase.

Acquiring working knowledge: TensorFlow

After deciding to use TensorFlow and Keras, it took around two weeks to get familiarized with them by customizing a CNN to classify a small batch of polyps and non-polyps images. It was a good practice to understand better the theoretical background behind CNN.

Customizing a CNN from scratch

More preliminary experiments were carried out in the following weeks of March. This was the time to improve the already built CNN and investigate ways to deal with overfitting. It was also possible to acquire more data.

Fine-tuning a pre-trained model: VGG16

The first two weeks of April were meant to fine tune the first pre-trained model; VGG16. It included analyzing which hyperparameters worked better, such as which layers unfreeze, which was the optimal learning rate, etc.

Fine-tuning a pre-trained model: ResNet50

A similar process as the one discussed before was followed, except that the model in this case was ResNet50. It took approximately the same amount of time.

Fine-tuning a pre-trained model: DenseNet121

By the beginning of May, DenseNet121 was the only pre-trained model left. Two more weeks were spent on fine-tuning that model and, at the end, some time was dedicated to present the results of all the models by elaborating confusion matrices and ROC curves.

Ensemble learning: combining pre-trained models

The aim of that week was to create an ensemble of the three pre-trained models and spend time on making a comparison of the final results.

Writing the report

Although some chapters were written along the process, the last three weeks were decisive to write the last chapters, as well as making any suggested changes by the advisors.

4.2 Estimated timeline

TASK NAME	HOURS	COLOR
Searching of existing literature	42	
Setting up the environment	36	
Acquiring working knowledge: TensorFlow	66	
Customizing a CNN from scratch	90	
Fine-tuning a pre-trained model: VGG16	108	
Fine-tuning a pre-trained model: ResNet50	78	
Fine-tuning a pre-trained model: DenseNet121	78	
Ensemble learning: combining pre-trained models	30	
Writing the report	126	

Figure 4.1: Table showing the hours estimated per each task, plus the color assigned to it.

February

Nº	S	M	T	W	T	F	S
			1	2	3	4	5
	6	7	8	9	10	11	12
	13	14	15	16	17	18	19
1	20	21	22	23	24	25	26
2	27	28					

March

Nº	S	M	T	W	T	F	S
2			1	2	3	4	5
3	6	7	8	9	10	11	12
4	13	14	15	16	17	18	19
5	20	21	22	23	24	25	26
6	27	28	29	30	31		

April

Nº	S	M	T	W	T	F	S
6						1	2
7	3	4	5	6	7	8	9
8	10	11	12	13	14	15	16
9	17	18	19	20	21	22	23
10	24	25	26	27	28	29	30

May

Nº	S	M	T	W	T	F	S
11	1	2	3	4	5	6	7
12	8	9	10	11	12	13	14
13	15	16	17	18	19	20	21
14	22	23	24	25	26	27	28
15	29	30	31				

June

Nº	S	M	T	W	T	F	S
15				1	2	3	4
16	5	6	7	8	9	10	11
	12	13	14	15	16	17	18
	19	20	21	22	23	24	25
	26	27	28	29	30		

Figure 4.2: Estimated timeline table. Each task has been scheduled on a 5-months calendar. Weeks start on Sunday.

Chapter 5

Background theory

This chapter provides an overview of the theoretical concepts that are crucial for understanding the experiments presented in posterior chapters. It has been organized in 5 sections, from introductory ideas to more specific ones.

5.1 Artificial Intelligence

Less than a decade after decrypting Enigma and helping win World War II, mathematician Alan Turing made a new question that changed everyone's life: "Can machines think?". That question was formulated in his seminal work, "Computing Machinery and Intelligence" [Turing 1950], which was published in 1950. There he described how to create intelligent machines and in particular how to test their intelligence. The Turing Test is still considered today as a benchmark to identify intelligence of an artificial system; a human should be able to distinguish in a teletype dialogue whether he is talking to a man or a machine.

The term "AI" could be attributed to John McCarthy of MIT, which Marvin Minsky defines as "the construction of computer programs that engage in tasks that are currently more satisfactorily performed by human beings because they require high-level mental processes such as: perceptual learning, memory organization and critical reasoning" (1956). From that point until the new century, AI experienced ups and downs: years of astonishing achievements, such as expert systems, in contrast with the well-known AI winters characterized by financial setbacks.

However, since 2010, a new bloom based on massive data and new computing power has boosted AI. It is no longer a question of coding rules, but of letting computers discover them alone by correlation and classification, on the basis of a massive amount of data. This situation has led AI to be applied to multiple fields, like chemistry and economics. Nowadays, AI branches out into multiple files (Figure 5.1).

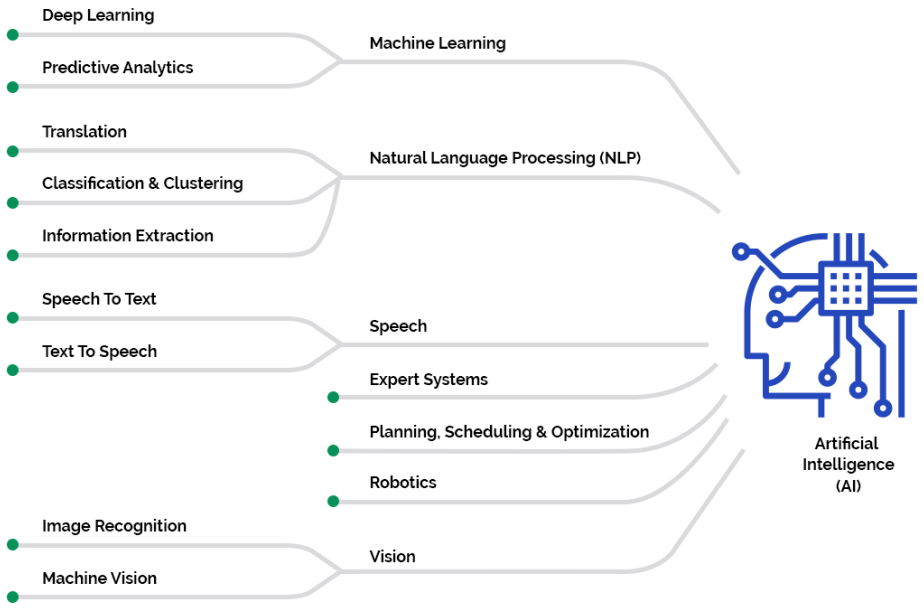


Figure 5.1: Sub-specialities of AI by 2021. Source: Bangbit technologies.

5.2 Machine learning

Machine learning is a subset of AI that focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy.

Instead of explicitly programming knowledge into computers, machine learning attempts to automatically learn meaningful relationships and patterns by observing examples. In its most basic form, the machine learning approach performs the task of acquiring domain knowledge by collecting a sufficiently large number of examples of desired behaviour for the algorithm of interest. These examples constitute the training set.

The examples in the training set (Figure 5.2) are fed to a learning algorithm to produce a trained "machine" that carries out the desired task. Learning is made possible by the choice of a set of possible "machines", also known as the hypothesis class, from which the learning algorithm makes a selection during training. An example of a hypothesis class is given by a neural network architecture with learnable synaptic weights. Learning algorithms are generally based on the optimization of a performance criterion that measures how well the selected "machine" matches the available data.

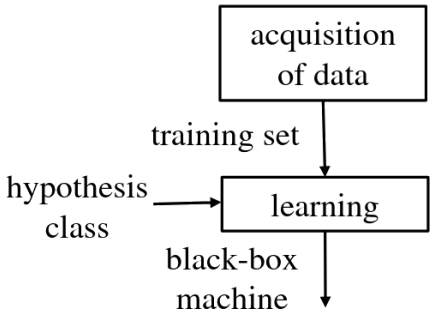


Figure 5.2: Baseline machine learning methodology.

There are three main classes of machine learning techniques (Figure 5.3):

- **Supervised learning**

  In supervised learning, the training set consists of pairs of input and desired output, and the goal is to learn a mapping function between input and output spaces. What characterizes supervised learning is that the data is labeled and the ML algorithm measures its accuracy through the loss function, adjusting until the error has been sufficiently minimized. Classification and regression are two major approaches in supervised learning.

- **Unsupervised learning**

  In unsupervised learning, the training set consists of unlabelled inputs, in other words, of inputs without any assigned desired output. This type of learning generally aims at discovering hidden patterns or data groupings without the need for human intervention. Unsupervised machine learning is mainly used in clustering, a task of dividing data into groups with similar properties.

- **Reinforcement learning**

  Reinforcement learning lies between supervised and unsupervised learning. In a certain sense, some form of supervision exists, but this does not come in the form of the specification of a desired output for every input in the data. Instead, a reinforcement learning algorithm receives feedback from the environment only after selecting an output for a given input or observation. The feedback indicates the degree to which the output, known as action in reinforcement learning, fulfils the goals of the learner.

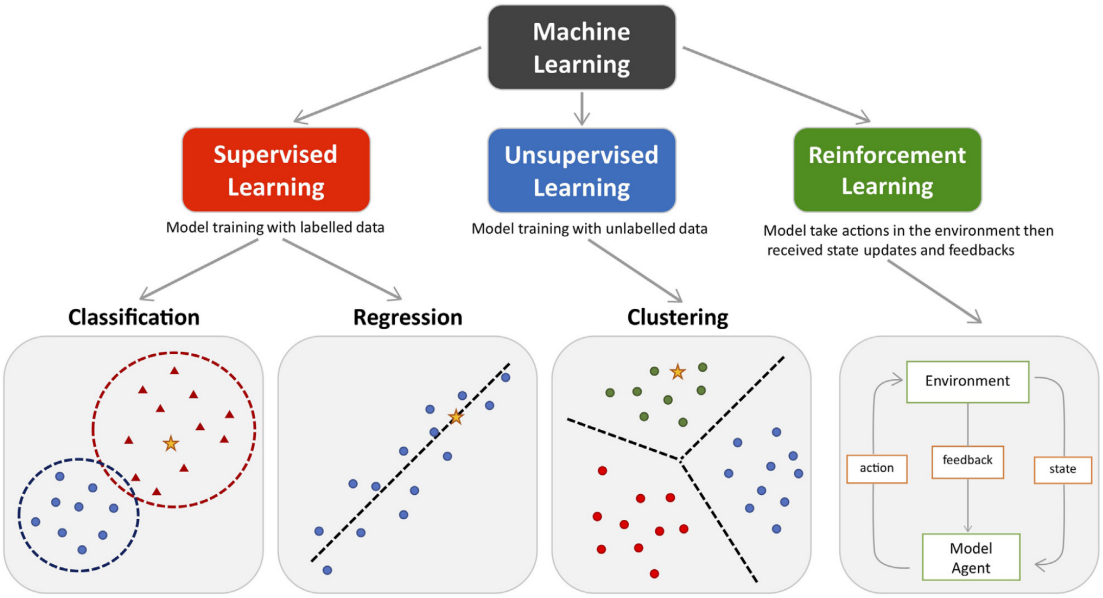


Figure 5.3: Main types of machine learning. Source: Machine Learning Techniques for Personalised Medicine Approaches in Immune-Mediated Chronic Inflammatory Diseases: Applications and Challenges.

5.3 Deep learning

Deep learning is a sub-field of machine learning that attempts to reach conclusions from analyzing data with a given logical structure. In contrast with traditional machine learning algorithms, deep learning algorithms are in a higher level of abstraction. They have the capacity of performing the laborious process of feature extraction. Each algorithm applies a nonlinear transformation to its input and uses what it learns to create a statistical model as output. It is done in multiple iterations until the output has reached an acceptable level of accuracy. The name "deep" comes from the number of layers that data has to pass through during these iterations.

Deep learning has been successfully applied to several problems; from self driving cars to speech recognition. In this particular project, deep learning is used in the task of classifying whether colonoscopy screenshots contain polyps or not. In order to achieve this, it has been necessary to understand how convolutional neural networks work, study how to combat some problems that threaten the precision of these models, and explain which metrics are mainly used to measure the accuracy of these models.

5.3.1 Deep Neural Networks

Deep neural networks (DNNs) are a set of algorithms, modeled loosely after the human brain. DNNs are represented by as a directed acyclic graph (Figure 5.4) composed by several stacked layers of neurons that attempt to approximate a certain function. They do so by transmitting a signal from the input layer to the output layer through hidden layers. A neuron on a layer has the task of detecting patterns from the incoming connections. The neuron combines input from the data with a set of coefficients, or weights, that either amplify or dampen that input. By doing that, each layer has the ability of capturing features from the data, and these features become more sophisticated the deeper the signal goes in the network.

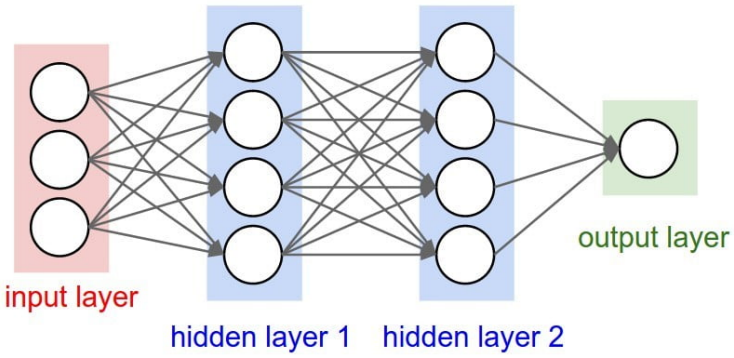


Figure 5.4: Deep feedforward neural network with 2 hidden layers. Two or more hidden layers comprise a Deep Neural Network

A unit in a layer is seen as a neuron, whereas arrows symbolize connections between neurons and each one holds a weight. Each node has a value associated and it is computed as a function of its incoming nodes and edges (Figure 5.5). It is done in the following way: the input-weight products are summed and then the sum is passed through a node's so-called activation function that squashes the resulting value between 0 and 1 to determine whether and to what extent that signal should progress further through the network to affect the ultimate outcome.

A DNN learns by optimizing a loss function that computes the distance between the current output of the algorithm and the expected output. Generally, the loss function must be differentiable, because many optimization algorithms rely on the gradient vector. These algorithms try to minimize the loss function, and Gradient Descent is the most common one.

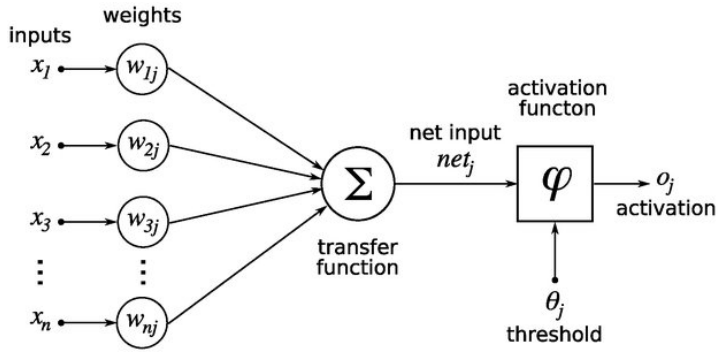


Figure 5.5: Architecture of a single neuron in a neural network.

5.3.2 Convolutional Neural Networks

A convolutional neural network (CNN) is a type of neural network commonly applied to analyze visual imagery due to its inspiration in the organization of animal visual cortex. The innovation of convolutional neural networks is the ability to automatically learn a large number of filters in parallel under the constraints of a specific predictive modeling problem, such as image classification. Hence, they are the perfect fit for this project.

A CNN is designed to automatically and adaptively learn spatial hierarchies of features through backpropagation by using multiple building blocks. In contrast to regular neural networks, CNNs use parameter sharing. All neurons in a particular feature map share weights which makes the whole system less computationally intense. Moreover, CNNs have neurons arranged in 3 dimensions: width, height, depth.

There are three main types of layers to build CNN architectures:

- **Convolutional layer**

  This layer is the first layer that is used to extract the various features from the input images, such as edges or some other feature in the image.

  The convolutional layer computes the convolutional operation of the input images using kernel filters to extract fundamental features. The kernel filters are of the same dimension but with smaller constant parameters compared to the input images. Depth must be the same for the matrix of inputs and the filter mask. The filter mask slides over the entire input image step by step and estimates the dot product between the weights of the kernel filters with the value of the input image, which results in producing a matrix called the feature map (Figure 5.6).

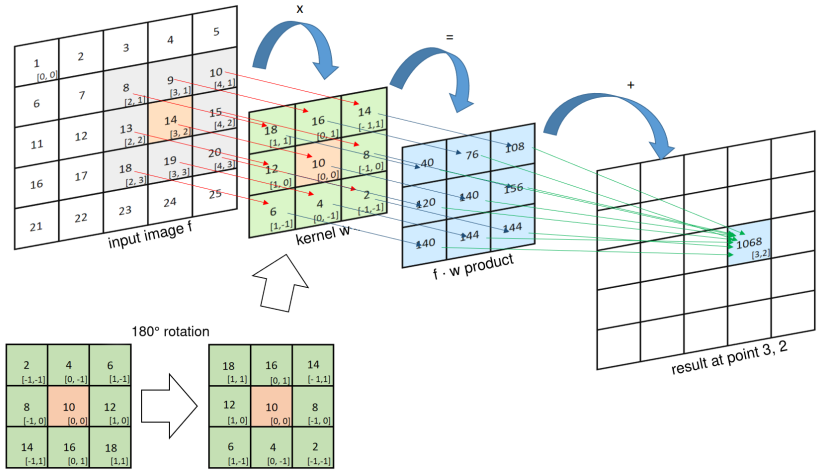


Figure 5.6: Example of convolutional operation on input image.

- **Pooling layer**

  A pooling layer is usually incorporated between two successive convolutional layers. The pooling layer reduces the number of parameters and computation by down-sampling the representation, keeping only the most important information. Pooling layers help control overfitting by reducing the number of calculations and parameters in the network.

  A form of pooling is max pooling (Figure 5.7)., which is done by applying a max filter to non-overlapping subregions of the initial representation. There are also other forms of pooling: average, general.

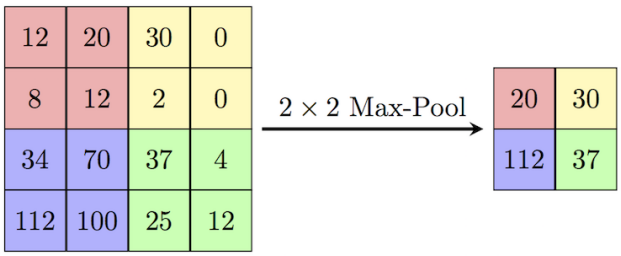


Figure 5.7: Example of max-pooling.

- **Fully-Connected layer**

  A fully-connected layer is the last layer in the convolutional neural network. It is called fully-connected because all inputs from one layer are connected to every activation unit of the next layer. It works as a classifier that determines to which class an image belongs to and it has as many neurons as number of classes. The layer receives an input vector,

and successively applies a linear combination and an activation function in order to classify the image input. Finally, the output is a vector of size corresponding to the number of classes in which each component represents the probability that the input image belongs to a class.

In this project, the last layer has a single neuron with an activation function such as Sigmoid because it is a binary problem.

Activation functions

As it was already mentioned, an activation function in a neural network defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network. Technically, the activation function is used within or after the internal processing of each node in the network, although networks are designed to use the same activation function for all nodes in a layer.

The choice of activation function has a large impact on the capability and performance of the neural network, and different activation functions may be used in different parts of the model. The most used ones are:

- **Rectified Linear Activation (ReLU)**

  ReLU performs a threshold operation to each input element where values less than zero are set to zero, otherwise, the value is returned.

- **Logistic (Sigmoid)**

  Sigmoid takes any real value as input and outputs values in the range of 0 to 1. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0.

- **Hyperbolic Tangent (Tanh)**

  Tanh takes any real value as input and outputs values in the range of -1 to 1. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to -1.0.

5.4 Methods to combat overfitting

The main goal of any machine learning model is to generalize well. It means that the target function learnt from the training data should generalize well

enough to other samples that have not been included in the modelling process. However, a model can face generalization problems when its capacity is higher than needed or when its capacity falls behind the complexity of a task. These problems are known as overfitting and underfitting (Figure 5.8) and they degrade the performance of the machine learning models.

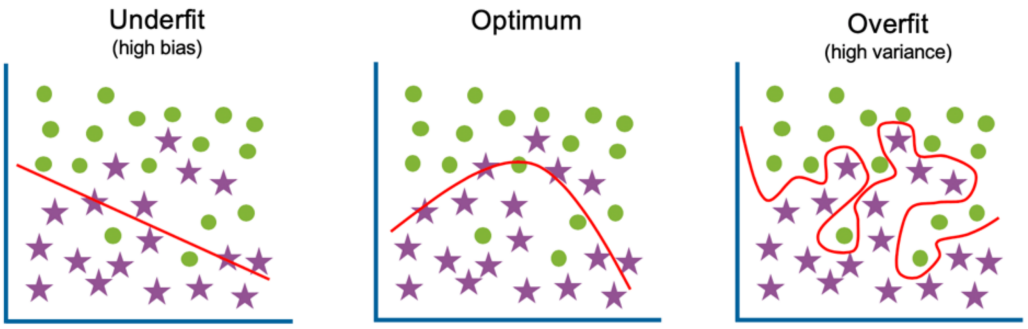


Figure 5.8: Examples of underfitting, optimum and overfitting. Source: IBM Garage Methodology.

Underfitting is a state where the model fails to significantly grasp the relationship between the input values and target variables, generating a high error rate on both the training set and unseen data. This may be the case when the model is too simple to capture patterns in the data. Therefore, the first thing to try is to increase the model complexity, for instance by adding hidden layers.

Overfitting is the opposite of underfitting. An overfit model has overly memorized the data set it has seen and is unable to generalize the learning to an unseen data set. It results in high error rates on test data. Overfitting a model is more common than underfitting one and harder to identify. The following subsections study which methods are useful to avoid overfitting when training CNNs [Grosse 2020].

5.4.1 Early stopping

Early stopping is an optimization technique used to reduce overfitting when training a learner with an iterative method, such as gradient descent. The main idea behind early stopping is to stop training before a model starts to overfit.

When training, the training error ought to continue improving. The test error generally improves at first, but it may eventually start to increase as the network starts to overfit. This suggests to stop the training at the point where

the generalization error starts to increase. Hence, the validation error has to be monitored during training to determine when to stop.

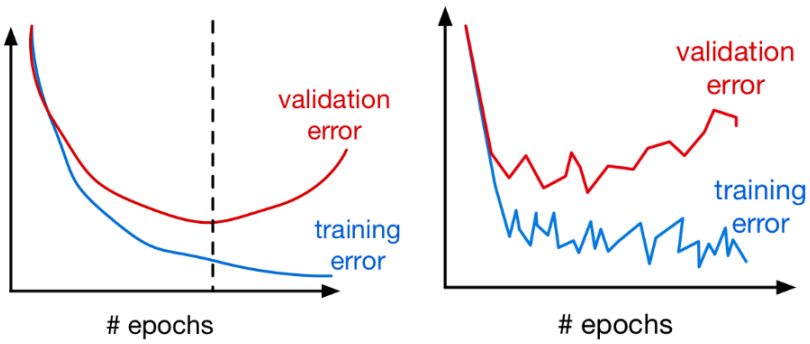


Figure 5.9: Relationship between the number of epochs and the validation and training error. **(left)** Idealized version. **(right)** Accounting for fluctuations in the error, caused by stochasticity in the SGD updates. Source: University of Toronto.

However, implementing early stopping is not so simple because the training and validation errors fluctuate during training. One common heuristic is to space the validation error measurements far apart.

5.4.2 Regularization

The basic idea of regularization through the cost function is to penalize the higher-order polynomials to the extent that the model is still able to represent the patterns in the data, but it does not get confused by noise. It consists of adding another term, called a regularization term, or regularizer, which penalizes hypotheses that are somehow pathological and unlikely to generalize well.

The total cost, then is:

$$\mathcal{E}(\boldsymbol{\theta}) = \underbrace{\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(y(x,\boldsymbol{\theta}),t)}_{\text{training loss}} + \underbrace{\mathcal{R}(\boldsymbol{\theta})}_{\text{regularizer}} \tag{5.1}$$

For instance, there are two sets of weights as shown in Figure 5.10 that make identical predictions in the training set, so they are equivalent. However, Hypothesis A is better because it is more stable. E.g., suppose the input (x1 = 1, x2 = 0) on the test set; in this case, Hypothesis A will predict 1, while Hypothesis B will predict -8.

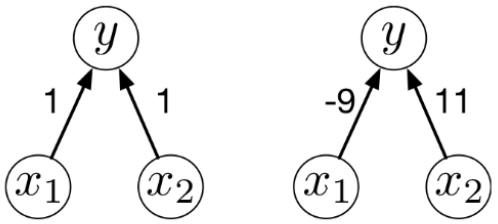


Figure 5.10: Two sets of weights which make the same predictions assuming inputs $x1$ and $x2$ are identical. **(left)** Hypothesis A. **(right)** Hypothesis B. Source: University of Toronto.

A regularizer that would favor Hypothesis A by assigning it a smaller penalty is L_2. L_2 regularization tends to favor hypotheses where the norms of the weights are smaller. It is defined as follows (The hyperparameter λ is sometimes called the weight cost.):

$$\mathcal{R}_{L_2}(\boldsymbol{w}) = \frac{\lambda}{2} \sum_{j=1}^{D} w_j^2 \tag{5.2}$$

5.4.3 Dropout layer

Dropout is a technique where randomly selected neurons are ignored during training or "dropped-out" randomly. This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. This means that the contribution of the neurons dropped-out to the activation of downstream neurons is temporarily removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

As a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features providing some specialization. Neighboring neurons rely on this specialization, which if taken too far can result in a fragile model too specialized to the training data.

Dropout is implemented as per-layer in a neural network that can be used with most types of layers. It has a hyperparameter that specifies the probability at which outputs of the layer are dropped out, or inversely, the probability at which outputs of the layer are retained.

5.4.4 Data augmentation

Another option is to artificially augment the training set by introducing distortions into the inputs, a procedure known as data augmentation (Figure 5.11). For instance, by shifting an image by a few pixels, adding noise, rotating it slightly, or applying some sort of warping. This can increase the effective size of the training set. Of course, not all transformations are useful. It depends on the task; for instance, in object recognition, it might be advantageous to flip images horizontally, whereas this would not make sense in the case of handwritten digit classification.

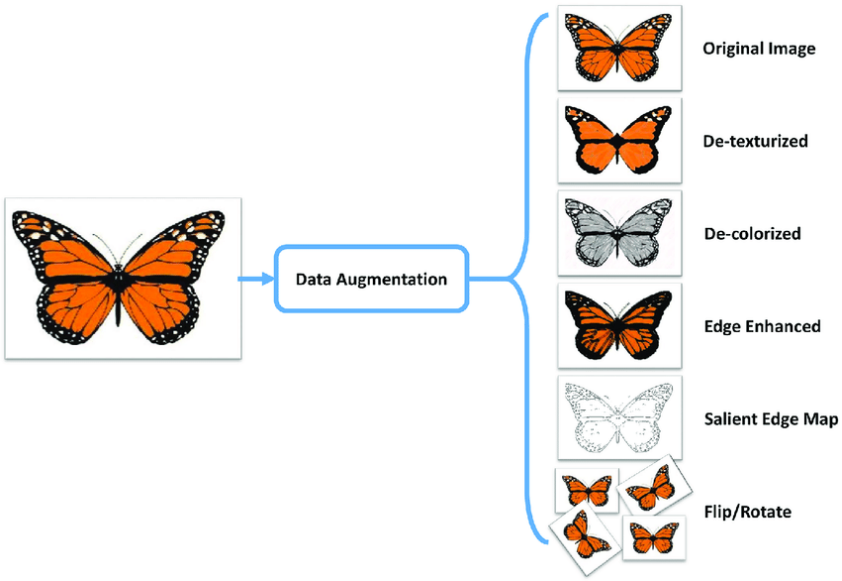


Figure 5.11: Examples of data augmentation on the same image. Source: Medium.

5.4.5 Transfer learning

Transfer learning is a machine learning technique that reuses a completed model that was developed for one purpose for a new model that accomplishes a new task. Analogous to human learning, transfer learning offers the capability of transfer knowledge across tasks instead of learning everything from scratch. The more similar the tasks, the easier is to cross-utilize knowledge. For instance, a model trained to recognize lions is likely to work well at recognizing tigers by making some changes on its inputs, outputs and layers.

There are complex tasks that require a lot of data, and are very specific. However, most deep learning models are specialized to a particular domain.

Transfer learning solves that by generalizing the features learnt by a model. It has been possible thanks to two majors reasons. Firstly, the creation of a huge dataset called "ImageNet" containing more than 14 million images hand-annotated with more than 20,000 categories. The second reason is the elaboration of advanced CNN architectures trained on datasets such as ImageNet.

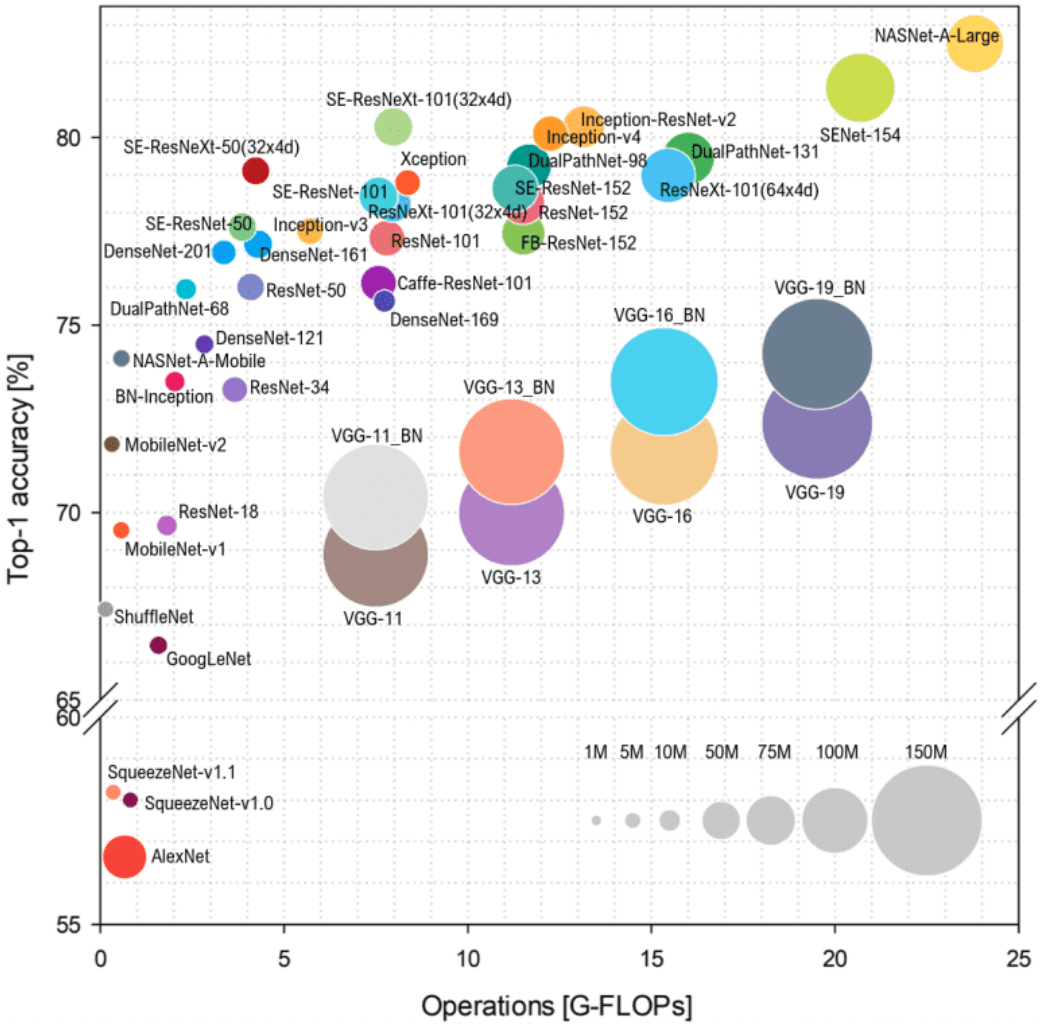


Figure 5.12: Overview of architectures until 2018. Source: Simone Bianco et al. 2018.

There are many architectures that can be pre-trained to perform a new task. Some CNN architectures are more popular than others, but it is important to consider not only their accuracy, but also their computational complexity. Figure 5.12 provides an overview of the top-performing CNNs until 2018. Moreover, the way in which these architectures are trained can be also

considered a new hyperparameter of the model. There are three strategies to fine-tune a model, which are training the entire model, training some layers and leaving the others frozen or freezing the convolutional base.

For this project, VGG16, ResNet50 and DenseNet121 have been the pre-trained models chosen for the task of detecting polyps due to their complexity-accuracy relationship.

5.4.5.1 VGG16

VGG16 is a convolutional neural network that was proposed by K. Simonyan and A. Zisserman at the University of Oxford in 2014 [Simonyan 2014]. They called it VGG after the department of Visual Geometry Group in the University of Oxford that they belonged to. The number 16 comes from the 16 layers that constitute the convolutional neural network.

As shown in Figure 5.13, VGG16 consists of a sequence of convolutional layers of filter size 3x3, stride one, and padding 1, followed by a max-pooling layer of size 2x2. The convolution stacks are followed by three fully connected layers, two with size 4,096 and the last one with size 1,000. The last one is the output layer with Softmax activation. The size of 1,000 refers to the total number of possible classes in ImageNet.

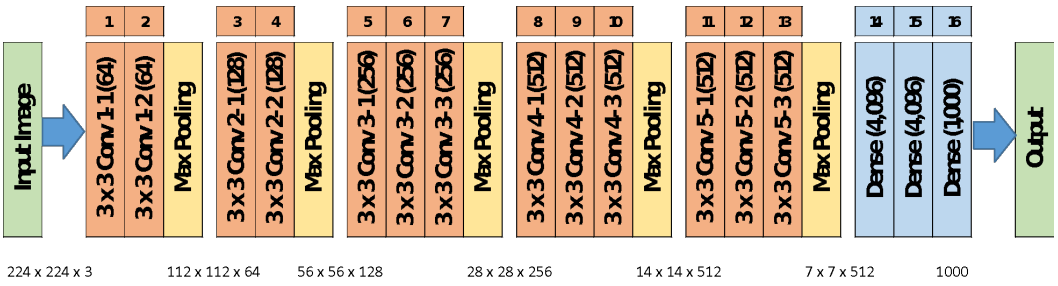


Figure 5.13: VGG16 architecture. Source: Great Learning Team.

A big difference compared to other models is that VGG uses a very small 3 x 3 receptive field (filters) throughout the entire network. However, one of the crucial downsides of the VGG16 network is that it is a huge network, which means that it takes more time to train its parameters. The total number of parameters in this model is over 138M, and the size of the model is over 500MB. This makes deploying VGG a tiresome task. Another problem is the Vanishing Gradient Problem. During backpropagation, the value of gradient decreases significantly, thus hardly any change comes to weights.

5.4.5.2 ResNet50

ResNet50 is a convolutional neural network belonging to the family of Residual networks that were introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in 2016 [He 2016].

ResNet50 is characterized by 50 layers, and has over 23 million trainable parameters. Its architecture (Figure 5.14) consists on one convolution and pooling step followed by 4 residual blocks of similar behavior. Each of the blocks, called residual blocks, follow the same pattern. They perform 3x3 convolution with a fixed feature map dimension (F) [64, 128, 256, 512] respectively, bypassing the input every 2 convolutions. These blocks are repeated [3,4,6,3] respectively. Finally, the resulting feature map goes through a global average pooling and a fully connected layer with Softmax to generate the final output.

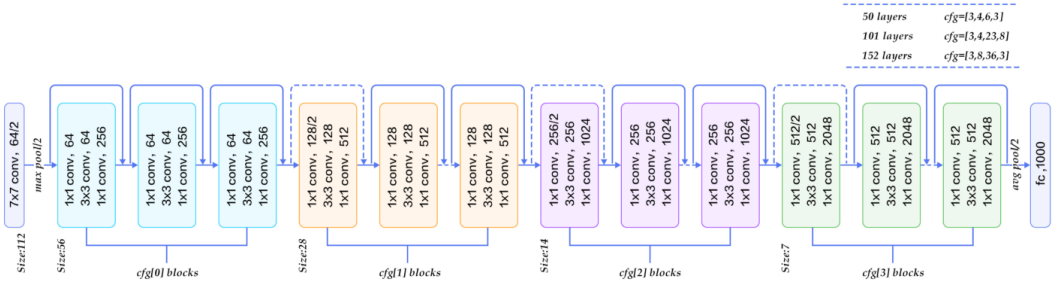


Figure 5.14: ResNet architecture. Source: Deep Residual Learning for Image Recognition.

ResNet first introduced the concept of skip connection (Figure 5.15). The idea is to connect the input of a layer directly to the output of a layer after skipping a few connections. Instead of multiplying input 'x' by the weights of the layer followed by adding a bias term, the value of 'x' is added to the output layer.

These skip connections alleviate the issue of vanishing gradient by setting up an alternate shortcut for the gradient to pass through. In addition, they enable the model to learn an identity function. This ensures that the higher layers of the model do not perform any worse than the lower layers.

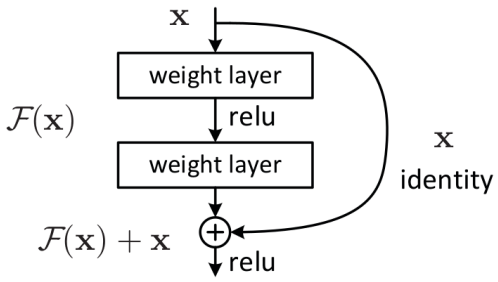


Figure 5.15: Skip connection in a residual block.

5.4.5.3 DenseNet121

DenseNet [Huang 2016] is a convolutional neural network where each layer is connected to all other layers that are deeper in the network. The first layer is connected to the 2nd, 3rd, 4th and so on. This is done to enable maximum information flow between the layers of the network.

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv; 3 × 3 conv] × 24	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48	[1 × 1 conv; 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv; 3 × 3 conv] × 16	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Figure 5.16: Dense blocks in different DenseNet architectures. Source: DenseNet paper-edited by author.

To preserve the feed-forward nature, each layer obtains inputs from all the previous layers and passes on its own feature maps to all the layers which will come after it. Unlike Resnets it does not combine features through summation but combines the features by concatenating them. Hence, the 'ith' layer has 'i' inputs and consists of feature maps of all its preceding convolutional blocks. Its own feature maps are passed on to all the next 'I-i' layers. This introduces '(I(I+1))/2' connections in the network, rather than just 'I' connections.

DenseNet121 (Figure 5.16) starts with a basic convolution and pooling layer. Then there is a dense block followed by a transition layer, repeated 3 times, and finally a dense block followed by a classification layer. A dense block has two convolutions, with 1x1 and 3x3 sized kernels, and each block is run for 6, 12, 24, 16 repetitions respectively. The transition layers reduce the number of channels to half of the existing channels applying a batch normalization, a 1x1 convolution and a 2x2 pooling layers.

5.4.6 Ensemble

Ensemble methods combine a group of predictive models to get an average prediction. It not only reduces the variance of predictions but also can result in predictions that are better than any single model. Generally, ensemble learning involves training more than one network on the same dataset, then using each of the trained models to make a prediction before combining the predictions in some way to make a final outcome.

A key part of an ensemble learning method involves combining the predictions from multiple models. It again depends on the problem. In the case of having a binary classification problem, there is what is called "voting". There are many types of voting, but the simplest one consists of selecting the label with the most votes. This only works when the number of models is odd. Other methods include majority voting, unanimous voting, and weighted voting.

5.5 Metrics and assessment

An essential part of this project is to evaluate a model against some metrics that help to monitor and measure its performance. The feedback resulting from these metrics is convenient to understand the weakness of a model and to be able to compare its power with other models.

There are various metrics used for evaluation. It also depends on which type of problem is being addressed. For classification, the most widespread metric is confusion matrix, from which you can derive significant terms also useful for assessment. In addition to confusion matrix, another popular metric is ROC (Receiver operating characteristic) curve and, in particular, the area under the ROC Curve (AUC). Added to this, performing cross validation gives robustness to the evaluation because it results in the mean and the standard deviation of the model's accuracy over k folds. That is why K-fold cross validation is considered a metric too.

5.5.1 Confusion matrix

A confusion matrix is a NxN matrix, where N is the number of classes being predicted. It shows how a classification model gets confused when it makes predictions. The matrix (Figure 5.17) is organized into columns and rows, representing the predicted classes and the actual classes respectively.

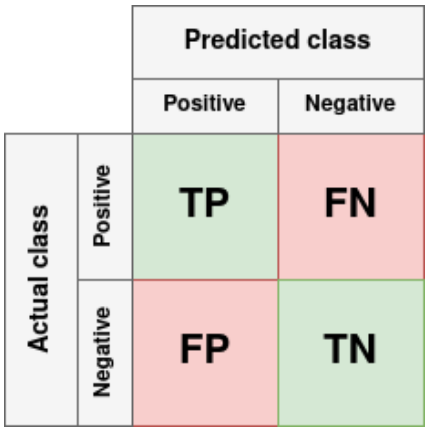


Figure 5.17: Confusion matrix

A binary confusion matrix is created thanks to four values:

- **True positive (TP)**: outcome where the model correctly predicts the positive class.
- **False positive (FP)**: outcome where the model incorrectly predicts the positive class.
- **True negative (TN)**: outcome where the model correctly predicts the negative class.
- **False negative (FN)**: outcome where the model incorrectly predicts the negative class.

Logically, summing the main diagonal gives the total of correct predictions. Whereas summing the antidiagonal gives the total of incorrect predictions.

From the previous values, there are some new metrics that can be calculated including accuracy, precision, recall, specificity, and F1 score.

Accuracy

Accuracy is the most common and simple performance metric for classification. It is the ratio of the number of correct predictions to the total number

of predictions made for a data set, or in other words, the probability of classifying an input correctly. It is especially meaningful when the data set is balanced.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Precision

Precision or positive Predictive Value (PPV) is the ratio of true positives to all the positives predicted by the model. The more False positives the model predicts, the lower the precision.

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

Recall

Recall or true positive rate (TPR) is the ratio of true positives to all the actual positives in a dataset. The more false negatives the model predicts, the lower the recall.

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

Specificity

Specificity or true negative rate (TNR) is the ratio of true negatives to all the actual negatives in a dataset. The more false positives the model predicts, the lower the specificity.

$$Specificity = \frac{TN}{TN + FP}$$

F1-score

F1-score or F-measure combines precision and recall. Mathematically, F1-score is the weighted average of the precision and recall. The classifier will only get a high F-score if both precision and recall are high.

$$F1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

5.5.2 AUC-ROC Curve

The Receiver Operator Characteristic (ROC) is a probabilistic curve that plots the true positive rate (TPR) against the false positive rate (FPR) at various threshold values (Figure 5.18). The true-positive rate is also known

as sensitivity, recall or *probability of detection*. The false-positive rate is also known as *probability of false alarm* and can be calculated as (1 - *specificity*).

The area under the curve (AUC) is a value between 0 and 1 that measures the ability of a classifier to distinguish between classes. It is used as a summary of the ROC curve. The higher the AUC, the better the performance of the model at distinguishing between the positive and negative classes.

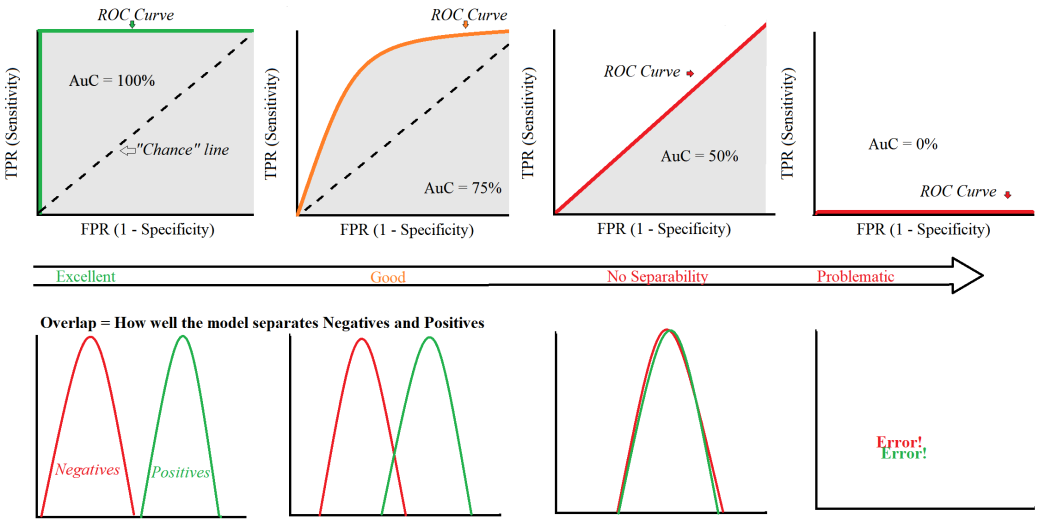


Figure 5.18: Comparison of multiple ROC curves and how much overlap there are between classes for each case. Source: Stephanie Glen.

5.5.3 K-fold Cross-validation

K-fold cross-validation (Algorithm 1) is a statistical method used to estimate the skill of machine learning models on unseen data. Particularly, this resampling procedure is of good practize when the models learn on a limited data sample. Cross-validation allows to train the model on different distributed samples and obtain the mean and the standard deviation for the metrics of interest.

It receives the name "K-fold Cross-validation" because it has a single parameter called k. This parameter makes reference to the number of groups that a given data sample is to be split into. One approach to choose k is to explore the effect of different k values on the estimate of model performance and compare this to an ideal test condition.

Algorithm 1 K-fold cross validation

Shuffle the dataset randomly

Split the dataset into k groups

for `each unique group` **do**

  Take the group as a test data set

  Take the remaining groups as a training data set

  Fit a model on the training set and evaluate it on the test set

  Retain the evaluation score and discard the model

end for

Summarize the skill of the model using the sample of model evaluation scores

Chapter 6

Studies and decisions

This chapter is devoted to study which tools are necessary to develop the project and to understand why they are considered to be of good use. The first part of the chapter is focused on explaining which requirements the system must satisfy, both, functional and non-functional. Whereas, the second part names and describes the technology chosen for the project.

6.1 System requirement

The Project Management Body of Knowledge, Seventh Edition defines requirement as *a condition or capability that is required to be present in a product, service, or result to satisfy a business need* [Institute 2021].

This section explores which requirements are necessary to detect polyps. It is important to differentiate the final system from the work behind. The process of building and training a CNN requires computational power, sophisticated libraries and, usually, a big amount of time. However, once the model has been trained and saved, it can be used anytime. Generally, the time of response is of the order of seconds.

Requirements can be classified into functional and non-functional requirements:

- **Functional requirements** are capabilities that the product must do to satisfy specific user needs.
- **Non-functional requirements** include usability, performance, reliability and security requirements.

6.1.1 Functional requirements

The functional requirements of the system are:

- Given a set of images or a single one belonging to a colonoscopy, classify whether it contains a polyp or not. It is a binary classification.

6.1.2 Non-functional requirements

Thanks to Jupyter Notebook and Python3, any computer should be able to run the code and classify images. Obviously, enough RAM is necessary to read the desired images.

6.2 Hardware

When it comes to designing and training a model, it would be wise to utilize a powerful computer. Otherwise, the computer could run out of memory in the middle of the process.

Table 6.1, 6.2 and 6.3 provide a view of the characteristics of the machine employed for the project. It is recommended to use a similar one since it has not been tested on computers with less capability.

Memory	
Total	15 GiB
Available	5,9 GiB
Swap	2 GiB

Table 6.1: Memory description of the machine employed

CPU	
Model name	AMD Ryzen 7 5800H with Radeon Graphics
Byte Order	Little Endian
Core(s) per socket	8
Thread(s) per core	2
Socket(s)	1
CPU max MHz	3200
CPU min MHz	1200
L1d cache	256 KiB
L1i cache	256 KiB
L2 cache	4 MiB
L3 cache	16 MiB

Table 6.2: CPU description of the machine employed

GPU	
Graphics Processor	NVIDIA GeForce RTX 3050 Laptop GPU
CUDA Cores	2048
Total Memory	4096 MB
Driver Version	470.42.01
CUDA Version	11.4

Table 6.3: GPU description of the machine employed

The operating system used is `Ubuntu 20.04.2 LTS`.

6.3 Software

In order to implement and execute the code of this project, it has been necessary to prepare a specific environment where the required packages have been installed. Annex Installation manual provides a detailed explanation of setting it up.

The following subsections name and describe each tool.

6.3.1 Python 3.9.7

Python (Figure 6.1) is a high-level, interpreted, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation.

Benefits that make Python the best fit for machine learning and AI-based projects include simplicity and consistency, access to great libraries and frameworks for AI and machine learning (ML), flexibility, platform independence, and a wide community.

Figure 6.1: Python logo. Source: Python Software Foundation, 2022.

6.3.2 Anaconda

Anaconda (Figure 6.2) is a distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.), that aims to simplify package management and deployment.

One of its advantages is that Anaconda makes the task of installing libraries and packages very easy thanks to its feature of creating independent environments.

Figure 6.2: Anaconda logo. Source: Anaconda, Inc.

6.3.3 Jupyter Notebook

Jupyter Notebook (Figure 6.3) is a web-based interactive computing platform. The web application can be used to create and share documents that contain live code, equations, visualizations, and text. It comes already installed with Anaconda.

Figure 6.3: Jupyter Notebook logo. Source: Project Jupyter.

6.3.4 Google Colab

Google Colab (Figure 6.4) allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education. More technically, Colab is a hosted Jupyter notebook service that requires no setup to use, while providing access free of charge to computing resources including GPUs.

Figure 6.4: Google Colab logo. Source: Google.

6.3.5 CUDA 11.4

CUDA (Figure 6.5) is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.

Figure 6.5: CUDA logo. Source: Nvidia Corporation.

6.3.6 CuDNN 8.2.4

The NVIDIA CUDA Deep Neural Network library (cuDNN) (Figure 6.6) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.

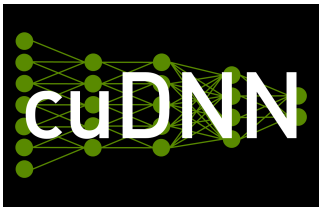


Figure 6.6: cuDNN logo. Source: Nvidia Corporation.

6.3.7 TensorFlow 2.8

TensorFlow (Figure 6.7) is a free and open-source software library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks.

The main reasons for using TensorFlow are:

- TensorFlow provides an accessible and readable syntax which is essential for making these programming resources easier to use.
- TensorFlow provides excellent functionalities and services when compared to other popular deep learning frameworks.
- TensorFlow is a low-level library which provides more flexibility.

Figure 6.7: TensorFlow logo. Source: Google Brain Team.

6.3.8 Keras

Keras (Figure 6.8) is an open-source software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library.

It contains numerous implementations of commonly used neural-network building blocks such as layers, objectives, activation functions, optimizers, and a host of tools to make working with image and text data easier to simplify the coding necessary for writing deep neural network code.

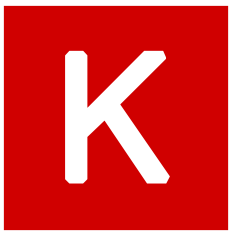


Figure 6.8: Keras logo. Source: Google Brain Team.

6.3.9 Numpy

NumPy (Figure 6.9) is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

It has been used as a data structure to store the images in the right format. Plus, it provides many functions to work with numpy arrays.

Figure 6.9: Numpy logo. Source: Community project.

6.3.10 Matplotlib

Matplotlib (Figure 6.10) is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK. There is also a procedural "pylab" interface based on a state machine (like OpenGL), designed to closely resemble that of MATLAB, though its use is discouraged.

In this project, it has been used to display the original images and its transformations after data augmentation.

Figure 6.10: Matplotlib logo. Source: The Matplotlib Development team.

6.3.11 Seaborn

Seaborn (Figure 6.11) is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. It has been used to plot the heat maps regarding confusion matrices.

It has been used to create the heat maps related to the confusion matrices.

Figure 6.11: Seaborn logo. Source: Michael Waskom

6.3.12 Pandas

Pandas (Figure 6.12) is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series. It is free software released under the three-clause BSD license.

It has been used to store the metrics of each model in a csv.

Figure 6.12: Pandas logo. Source: The pandas developers.

6.3.13 Sklearn

Sklearn (Figure 6.13) is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support-vector machines, random forests, gradient boosting, k-means and DBSCAN, and is designed to interoperate with the Python numerical and scientific libraries NumPy and SciPy.

It has been used to calculate the metrics on the test set after training the model.

Figure 6.13: Sklearn logo. Source: The scikit-learn developers.

6.3.14 cv2

cv2 is the module import name for opencv-python. OpenCV (Open Source Computer Vision Library) (Figure 6.14) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products.

It has been used to read the images, resize them and normalize them.

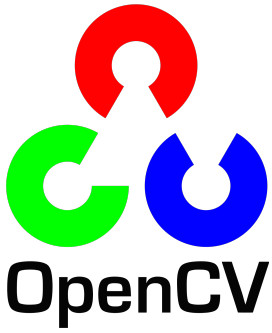


Figure 6.14: OpenCV logo. Source: OpenCV team.

6.3.15 Gimp

Gimp (Figure 6.15) is a free and open-source raster graphics editor used for image manipulation (retouching) and image editing, free-form drawing, transcoding between different image file formats, and more specialized tasks. It is not designed to be used for drawing, though some artists and creators have used it for such.

It has been used for customizing all the diagrams and other figures added to the documentation.

Figure 6.15: Gimp logo. Source: GIMP Development Team.

6.3.16 LaTeX

LaTeX (Figure 6.16) is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation. LaTeX is the de facto standard for the communication and publication of scientific documents. LaTeX is available as free software.

It has been used for writing the documentation.

Figure 6.16: LaTeX logo. Source: the LaTeX project.

Chapter 7

Analysis and design

This chapter is devoted to analyze which parts of the project are of deep importance, such as the source of data or which parameters must be taken into account for training, and to design a diagram or solution that provides answers to these matters. This chapter is strongly related to the methodology explained previously, but provides more details about the implementation.

7.1 Data set

A data set of 2,000 images (Figure 7.1) is used for training and evaluating the deep learning models presented in this project. The set of 2,000 colonscopy images contains 1,000 images of unique polyps, of all sizes and morphologies, and 1,000 images without polyps. The data set is perfectly balanced. The data set includes both white light and NBI images, and covers all portions of the colorectum, including retro-views in the rectum and cecum, appendiceal orifice, and ileocecal valve. We deliberately and randomly included features such as forceps, snares, cuff devices, debris, melanosis coli, and diverticula in both polyp and non-polyp images in a balanced fashion, to prevent the machine learning system from associating the appearance of tools with the presence of polyps. The images were stored at a resolution of 640x480 pixels.

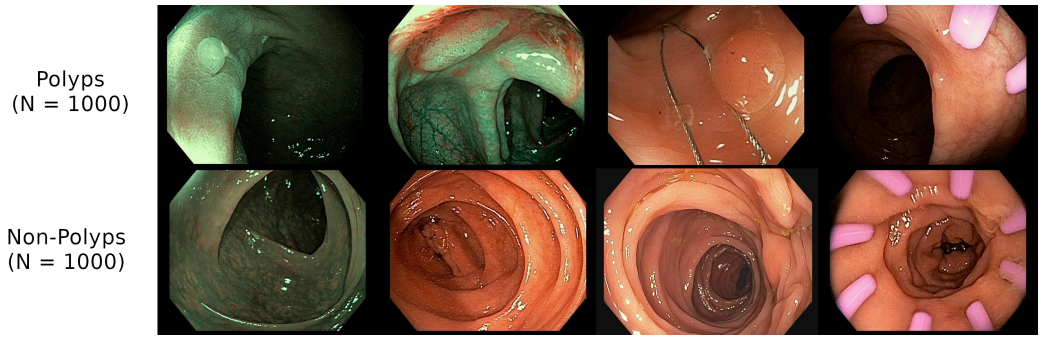


Figure 7.1: Examples of our data set. Top row: images containing a polyp, bottom row: non-polyp images. Three pictures on the left were taken using NBI (narrow band imaging) and three pictures on the right contain tools (e.g. biopsy forceps, cuff devices, etc.) that are commonly used in screening colonoscopy procedures.

In addition, pre-trained models, i.e. VGG16, ResNet50, DenseNet121, used the ImageNet challenge data set implicitly to pre-train the weights. The ImageNet challenge data set contains 1.2 million natural images of objects like boats, cars, and dogs, but no medical images. We reasoned that many of the fundamental features learnable on this data set will be transferable to the task of detecting polyps and thus use it to pre-initialize the weights of the models.

7.2 Pipeline

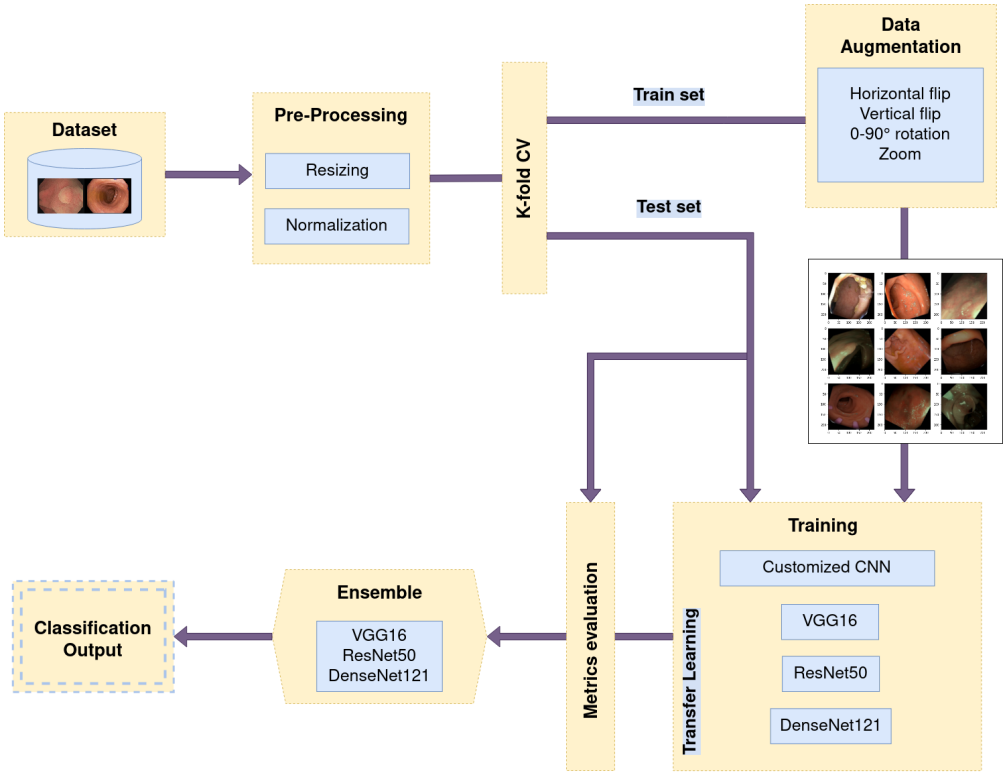


Figure 7.2: Pipeline of our system.

Figure 7.2 illustrates the pipeline we designed for this project, which was used as a guide to organize the steps of the experimental process. The different steps can be summarized in three parts. The first one is about data processing; reading the images and treating them before training. The second group encloses the training process for classifying images, which is the most elaborate part. And finally, the third one is about analyzing the results and creating an ensemble that boosts the performance of the pre-trained models.

The data set of 2,000 colonoscopy images provided by the University of California, Irvine, is stored locally, in other words, in our computer. It means that there is no existence of a data base to retrieve such images, but it is necessary to structure the data in different directories. This way is easier to load the data to the working station. Once the images are organized, the next step is to read them and pre-process them. By pre-processing we understand resizing the images and normalizing their pixels to a range of 0 to 1. Sequentially, the data must be split into train set and test set. Thus, the algorithm of K-fold cross validation is run. The k selected for cross validation is five because, despite that the default is usually ten, it was the limit threshold that the GPU memory could support.

The training consists of 4 architectures which are trained independently among the same train set under the task of classifying polyps and non-polyp images. One of these CNNs is created by scratch thanks to the layers, functions and other components provided by the Keras library that make it possible to build a model. The other three models are pre-trained by unfreezing some of their layers. These layers are then trained using the images belonging to our data set, and the models are saved locally for later. When training, the train test is expanded by applying the data augmentation technique. There are four types of transformations: vertical and horizontal flip, rotation and zoom.

The final blocks are devoted to visualize the results after evaluating the models on the test set and, if they are acceptable, create an ensemble that merges the three pre-trained models into one with better accuracy. The ways of visualizing data are the confusion matrix and the AUC-ROC curve.

7.3 Hyperparameter optimization

When training a model, the values of the hyperparameters play a big role. Hyperparameters are the variables which determine the network structure and how the network is trained. These hyperparameters are independent of the model and they cannot be directly trained from the data. They are learnt during training when we optimize a loss function. Therefore, there are techniques to optimize the hyperparameters that help to achieve better results.

There are many hyperparameters that can be chosen for optimization, and they vary depending on the type of neural network:

- **Number of neurons**

  One hyperparameter is the number of neurons in every hidden layer, except the last one, which is related to the number of classes. The number of neurons should be adjusted to the solution complexity. Adding more neurons to a layer increases the complexity of the model. However, it has two sides. On one hand, the model is prone to predict better since it has more capability. On the other hand, the number of trainable parameters can explode and make it impossible to train the model because of a matter of time and memory.

- **Activation function**

  The activation function in each layer can be also considered a hyperparameter. The input values moving from a layer to another layer keep changing according to the activation function. The activation function decides how to compute the input values of a layer into output values. Generally, the rectifier activation function is the most popular, combined with Sigmoid or Softmax in the last layer.

- **Optimizer**

  The layers of a neural network are compiled and an optimizer is assigned. The optimizer is responsible for changing the learning rate and weights of neurons in the neural network to reach the minimum loss function. The optimizer is very important to achieve the highest possible accuracy or minimum loss.

- **Learning rate**

  One of the hyperparameters in the optimizer is the learning rate. Learning rate controls the step size for a model to reach the minimum loss function. A higher learning rate makes the model learn faster, but it may miss the minimum loss function. A lower learning rate gives a better chance to find a minimum loss function. As a tradeoff lower learning rate needs higher epochs, or more time and memory capacity resources.

  As shown in Figure 7.3, a low learning rate slows down the learning process but converges smoothly. Larger learning rate speeds up the learning but may not converge.

- **Number of epochs**

  The number of times a whole dataset is passed through the neural network model is called an epoch. One epoch means that the training data

set is passed forward and backward through the neural network once. A too-small number of epochs results in underfitting because the neural network has not learned enough. The training data set needs to pass multiple times or multiple epochs are required. On the other hand, too many epochs will lead to overfitting where the model can predict the data very well, but cannot predict new unseen data well enough. The number of epochs must be tuned to gain the optimal result.

- **Batch size**

  The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters. For instance, if our model is trained on 1,000 images and the batch size is ten, it means that ten images will be passed as a group, or as a batch, at one time to the network.

  It has been observed in practice that when using a larger batch there is a significant degradation in the quality of the model, as measured by its ability to generalize [Keskar 2016].

The hyperparameters mentioned are only a few. The number of dropout layers, the weight initialization, the pool size or the configuration of early stopping are also hyperparameters that can be included in the optimization. Thus, we realize that the total number of hyperparameters, together with the values that they can adopt, result in a huge space of combinations which makes it almost impossible to do it manually.

A wise way of optimizing these hyperparameters is using an algorithm called "Grid search". Essentially, we divide the domain of the hyperparameters into a discrete grid. Then, we try every combination of values we previously chose on this grid, calculating some performance metrics using cross-validation. The point of the grid that maximizes the average value in cross-validation, is the optimal combination of values for the hyperparameters. However, Grid search is an exhaustive algorithm that spans all the combinations so it can find the best point in the domain. Hence, it is very slow, and sometimes it is more convenient to use Random search, a variation of gird search that only evaluates some random combinations.

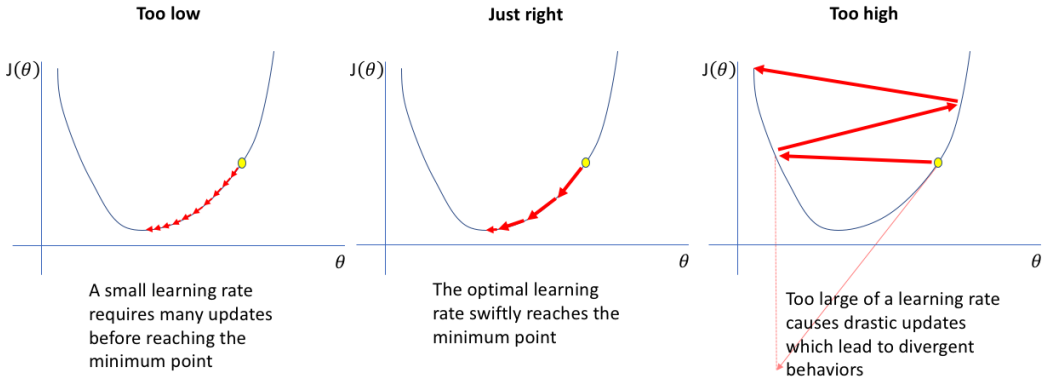


Figure 7.3: Comparison between learning rates.

Our research group created "Sherpa" [Hertel 2020], a software exclusively developed for optimizing hyperparameters. Nevertheless, tuning hyperparameters is a complex task that requires time. This time can be reduced if the algorithm is split into different processes and run in parallel. In order to run the algorithm in parallel, it is necessary to have a cluster of GPUs, something we do not have access to. Therefore, this option was discarded.

In light of the above, we decided to choose a sub set of hyperparameters and a range of values to combine them manually and test them on fold 1. Table 7.1 provides a summary of the hyperparameters for optimization and their range of values.

Hyperparameters	**Range**
Learning rate	10^{-3}, 10^{-4}, 10^{-5}
Optimizer	Adam, SGD
Batch size	8, 16, 32
Epochs	15, 30, 60

Table 7.1: Hyperparameters that were selected for optimization, together with the range of values they could take.

The resulting hyperparameters with which each model is trained are shown in Table 7.2. In contrast to the customized CNN, it is not necessary to train any of the pre-trained models until epoch 60 because they already started with a higher accuracy on both train and test set. Thus, after epoch 30, the accuracy on the test set barely improved.

Hyperparameters	own CNN	VGG16	ResNet50	DenseNet121
Learning rate	10^{-4}	10^{-4}	10^{-4}	10^{-4}
Optimizer	Adam	Adam	Adam	Adam
Batch size	16	16	16	16
Epochs	60	30	30	30

Table 7.2: Hyperparameters selected for each model

Chapter 8

Implementation and results

This chapter describes the experiments carried out with the finality of elaborating a CAD system, based on CNNs, which has the capability of detecting whether a colonoscopy image contains a polyp or not. Firstly, we introduce how the images are read and pre-processed before they are fed to a model. After that, the training implementation is explained in detail, focusing in each architecture, and describing which techniques and parameters have been chosen to improve the performance of each CNN. Finally, the results are presented in a illustrative way, and compared between all the models tested.

8.1 Data loading and preprocessing

The department of Medicine at the University of California, Irvine, provided a data set of 2,000 colonoscopy images well balanced of polyps and non-polyps [1,000, 1,000], as described in 7.1. These images were received in two directories named "polyp" and "non_polyp", and saved locally in our computer.

8.1.1 Preparing data for K-fold cross validation

The first step was to divide locally the data set into k folders containing the same amount of images from each class. The reason behind this is to train the models doing k-fold cross validation. Despite the fact that sklearn already provides some methods to do so, our purpose was to always use the same folders. Otherwise, the models would be trained using a different distribution of images for training and testing, which would make it impossible to compare models. It could have been done manually, but since we wanted to test different k in K-fold cross validation, it was clever to create a script that could do that for us.

The script receives as parameters the number of folds and the path to the data set directory:

```
./kfold_split -f 5 -d ./data
```

and it creates a fold in the father's directory of the data set with the structure shown in Figure 8.1. All the images are distributed randomly.

```
In [3]: pathCV = '/home/laura/Desktop/TFG/code/data/5_cv' #path to the dataset
        ! ./tree -d {pathCV}

        +5_cv
        +---fold1
        |   +---non_polyp 200 files 10659931 B
        |   +---polyp 200 files 9099862 B
        +---fold2
        |   +---non_polyp 200 files 9716256 B
        |   +---polyp 200 files 7895411 B
        +---fold3
        |   +---non_polyp 200 files 9010442 B
        |   +---polyp 200 files 7230386 B
        +---fold4
        |   +---non_polyp 200 files 8112496 B
        |   +---polyp 200 files 6673187 B
        +---fold5
        |   +---non_polyp 200 files 5771353 B
        |   +---polyp 200 files 5854818 B
```

Figure 8.1: Structure of the sub-directories requested to perform 5-fold cross validation. This is the result after runing the script *kfold_ split*

Considering that k-fold cross validation takes one fold as test set for each iteration, the data was distributed in the way presented in Table 8.1 and Table 8.2.

	POLYP	NON-POLYP	TOTAL
TRAIN	800	800	**1600**
TEST	200	200	**400**
TOTAL	**1000**	**1000**	**2000**

Table 8.1: Distribution of images in train and test set.

	POLYP	NON-POLYP	TOTAL
TRAIN	40%	40%	**80%**
TEST	10%	10%	**20%**
TOTAL	**50%**	**50%**	**100%**

Table 8.2: Percentage of images in train and test set.

8.1.2 Loading data

The second task was loading the data on our software. A first approach was to create a 2D array, where each row represented a fold, and each image was stored in a cell. That way, the algorithm of cross validation looped over the array and chose which fold had to be treated as test set. However, we did not have enough GPU memory for iterating and running the same model five times continually. After fold four, the kernel died every time.

```
def read_images_kfold(path, dict_labels, i):

    """ Returns four numpy arrays containg images and labels from the path directory
    First array -- images for training
    Second array -- labels mapping the images of training
    Third array -- images for testing
    Fourth array -- labels mapping the images of testing

    path -- a string encoding the path to the k-fold directory
    dict_labels -- a dictionary of the labels the data is classified into
    i - fold destined to testing
    """

    #values returned
    x_train = []
    y_train = []
    x_test = []
    y_test = []

    name_folder = 'fold'+ str(i) #test fold
    for folder in tqdm(os.listdir(path)): #for each fold
        newPath = os.path.join(path, folder) #path to the fold
        imagesFold = []
        labelsFold = []
        for group in (os.listdir(newPath)): #for each class
            value_of_label = dict_labels[group] # class name
            for file in (os.listdir(os.path.join(newPath, group))): #for each image
                path_of_file = os.path.join(os.path.join(newPath,
                                                         group), file)
                image = cv2.imread(path_of_file)
                image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
                image = cv2.resize(image, (IMG_SIZE, IMG_SIZE)) #resizing
                image = cv2.normalize(image, None, 0, 255,
                                      norm_type=cv2.NORM_MINMAX) #normalization
                imagesFold.append(image)
                labelsFold.append(value_of_label)

        if(name_folder == folder):
            x_test = imagesFold
            y_test = labelsFold
        else:
            x_train = x_train + imagesFold
            y_train = y_train + labelsFold

    return np.array(x_train), np.array(y_train), np.array(x_test), np.array(y_test)
```

Figure 8.2: Function used to load the data set of colonoscopy images

The solution was to run the same code multiple times, changing the fold that had to be selected for testing. Therefore, a new parameter “k” had to

be passed to the function, and instead of using 2D arrays, four vectors were necessary; two for a train set, and two for a test set. One vector stored the images and the other one stored the labels to know whether the image at position x had a polyp or not. Figure 8.2 presents the function for loading images. Before saving an image, it is resized to a size 224x224, and the pixels are normalized to a scale between 0 and 1. Figure 8.3 shows a grid of images from the train test after they were loaded.

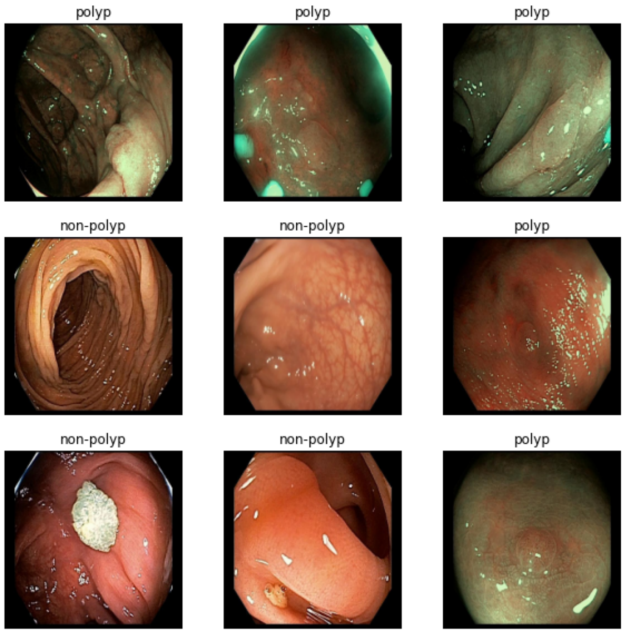


Figure 8.3: Set of images extracted from the train set after they were loaded. Each image has on top the label describing whether it has a polyp or not.

In order to reduce the time of training for each fold, we made use of four accounts in Google Colab, in addition to the environment set up in our computer. That way, we could run the same model five times in parallel and replicate the behaviour of k-fold cross validation by changing the distribution of the train and test sets.

8.1.3 Data Augmentation

As it was explained in section 5, data augmentation is a technique that creates more data by applying multiple transformations on an image. Some examples of data augmentation that can be used in Keras are:

- **Horizontal and vertical flips**: An image flip means reversing the rows or columns of pixels in the case of a vertical or horizontal flip respectively.

- **Rotations**: A rotation rotates the image clockwise by a given number of degrees from 0 to 360.

- **Zoom**: A zoom augmentation randomly zooms the image in and either adds new pixel values around the image or interpolates pixel values respectively.

Horizontal and vertical flips, rotations in the range of 90°, and 10% of zoom are the transformations that were selected (Figure 8.4). We tested other options, such as cropping the center of the image, which was not a good idea because polyps are not always in the center of the colonoscopy. Table 8.3 illustrates which transformations were approved or discarded after testing them.

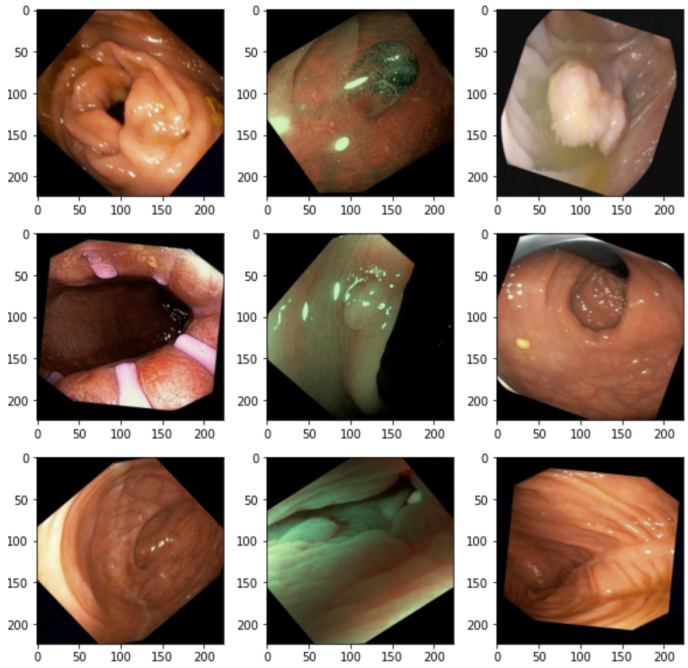


Figure 8.4: Example of images after data augmentation. Horizontal and vertical flips, rotations in the range of 90°, and 10% of zoom were applied.

	DATA AUGMENTATION		
Approved	Vert/Hor flipping	Zoom +/- 10	Rotation 0 - 90°
Discarded	Center cropping	Color augmentation	Gaussian blur

Table 8.3: List of transformations that were applied or discarded during data augmentation.

8.2 Neural Network Architectures

From the four models trained in this project, one of the models was built from scratch, whereas the other three were architectures that had been previously trained on the ImageNet data set.

8.2.1 Customized CNN

Keras provides an interface for artificial neural networks easy to use and with a big number of useful components, such as layers, activation functions, optimizers, etc. The first step is to define the structure of the CNN. It implies arranging the layers, choosing the filters or neurons per layer, deciding which activation functions must be used, etc.

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_24 (Conv2D)          (None, 224, 224, 32)      2432

 max_pooling2d_24 (MaxPoolin  (None, 112, 112, 32)     0
 g2D)

 dropout_30 (Dropout)        (None, 112, 112, 32)      0

 conv2d_25 (Conv2D)          (None, 112, 112, 64)      51264

 max_pooling2d_25 (MaxPoolin  (None, 56, 56, 64)       0
 g2D)

 dropout_31 (Dropout)        (None, 56, 56, 64)        0

 conv2d_26 (Conv2D)          (None, 56, 56, 128)       204928

 max_pooling2d_26 (MaxPoolin  (None, 28, 28, 128)      0
 g2D)

 dropout_32 (Dropout)        (None, 28, 28, 128)       0

 conv2d_27 (Conv2D)          (None, 28, 28, 128)       409728

 max_pooling2d_27 (MaxPoolin  (None, 14, 14, 128)      0
 g2D)

 dropout_33 (Dropout)        (None, 14, 14, 128)       0

 flatten_6 (Flatten)         (None, 25088)             0

 dense_12 (Dense)            (None, 512)               12845568

 dropout_34 (Dropout)        (None, 512)               0

 dense_13 (Dense)            (None, 1)                 513

=================================================================
Total params: 13,514,433
Trainable params: 13,514,433
Non-trainable params: 0
_________________________________________________________________
```

Figure 8.5: Architecture of the CNN built from scratch.

There is no strict rule that must be followed when building a CNN. It is a matter of trial and failure, but the order of the layers in a CNN must follow a logical idea (i.e., start with a convolutional layer followed by another convolutional or max pooling layer), and the complexity of the model must be equal to the complexity of the problem, otherwise it is likely to overfit or underfit.

Before reaching a final version, we started with a very simple model which had only one fully-connected layer, without any dropout layer or regularizer. The activation functions used were ReLU and Sigmoid. In total the first model had 185 k trainable parameters, which also explained why the model underfitted. That is why more layers were added with the goal of increasing the complexity of the model. It was a process of adding and removing layers, trying a different number of neurons, etc.

Figure 8.5 shows the resulting model, which has in total over 13 million of trainable parameters. In order to reduce the overfitting, we placed a dropout layer with a probability of 10% after each max pooling layer. Moreover, we introduced another dropout layer with a probability of 50% between the two fully-connected layers. A l2 regularizer was also applied in the first fully connected layer of 512 neurons.

8.2.2 VGG16

In order to use the VGG16 architecture, we had to import the model from Keras (Figure 8.6). After that, the next step was to create an instance of the model, specifying the following parameters:

- **weights**: the weights with which VGG16 is initialized. We chose ImageNet.
- **include_top**: whether to include the three fully-connected layers at the top of the network. In this case, the value is False because the top layers had to be trained according to our data set.
- **input_shape**: optional shape tuple, only to be specified if include_top is False. The shape of our images was (224, 224, 3), the same size of the images that were used to optimize VGG16.

```
#Pretrained model
from tensorflow.keras.applications import VGG16
```

Figure 8.6: Importing VGG16 model.

Before tuning the top layers, we decided to unfreeze the four last convolutional blocks of the model. The number of layers or which layers should be unfrozen can be also considered as a hyperparameter. By unfreezing layers we permit that more layers are trained using our data set, which can help to get a better feature extraction and classification. However, there is the risk of unfreezing too many layers and cause the network to overfit, leading to poor generalization.

Finally, the output had to be adapted to our data set. It included four layers: an average pooling followed by two fully-connected layers with a dropout layer between them. At the end, the number of trainable parameters was 7,605,761 out of 15,241,025. The code presented in Figure 8.7 shows the explained steps.

```
#pre-trained weights from imagenet. Load model without output layer, then input_shape required
vgg_base = VGG16(weights = 'imagenet', include_top = False, input_shape = input_shape)

# Unfreeze four convolution blocks
for layer in vgg_base.layers[:15]:
    layer.trainable = False

# Make sure you have frozen the correct layers
for i, layer in enumerate(vgg_base.layers):
    print(i, layer.name, layer.trainable)

# add a global spatial average pooling layer
x = vgg_base.output
x = GlobalAveragePooling2D()(x)
# add a fully-connected layer
x = Dense(1024, activation='relu', kernel_regularizer = 'l2')(x)
x = Dropout(0.5)(x)
# and a fully connected output/classification layer
outputs = Dense(1, activation='sigmoid')(x)

# create the full network
vgg_model = Model(inputs = vgg_base.input, outputs=outputs)
```

Figure 8.7: Fine-tuning the VGG16 model. First line is for initializing the model. The next one unfreezes the last four blocks and, after that, the output of the model is configured according to our data set.

8.2.3 ResNet50

Identically as VGG16, the first step was to import the model from Keras (Figure 8.8). The way of initializing the model and the block of output layers were both the same presented in VGG16.

```
#Pretrained model
from tensorflow.keras.applications import ResNet50
```

Figure 8.8: Importing ResNet50.

However, the number of unfrozen layers was quite different. This time we decided to unfreeze all the batch normalization layers. In addition, layer 165 to layer 174, the last 10 blocks, were also unfrozen after testing different approximations. From a total number of 25,686,913 parameters, 6,611,841 were trainable. Figure 8.9 provides the code of fine-tuning ResNet50.

```
#pre-trained weights from imagenet. Load model without output layer, then input_shape required
res_base = ResNet50(weights = 'imagenet', include_top = False, input_shape = input_shape)

res_base.trainable = False

# un-freeze the BatchNorm layers
for layer in res_base.layers:
    if "BatchNormalization" in layer.__class__.__name__:
        layer.trainable = True

# un-freeze last blocks
for layer in res_base.layers[165:]:
     layer.trainable = True

# Make sure you have frozen the correct layers
for i, layer in enumerate(res_base.layers):
    print(i, layer.name, layer.trainable)

x = res_base.output
# add a global spatial average pooling layer
x = GlobalAveragePooling2D()(x)
# add a fully-connected layer
x = Dense(1024, activation='relu', kernel_regularizer = 'l2')(x)
x = Dropout(0.5)(x)
# and a fully connected output/classification layer
outputs = Dense(1, activation='sigmoid')(x)

# create the full network
res_model = Model(inputs = res_base.input, outputs=outputs)
```

Figure 8.9: Fine-tuning the ResNet50 model. First line is for initializing the model. The next one unfreezes the batch normalization layers. The layers from 165 to the last one are also unfrozen. Finally, the output of the model is configured according to our data set.

8.2.4 DenseNet121

For DenseNet121, we also imported the model from Keras (Figure 8.10). The configuration of the model and the output layers remained the same as VGG16.

```
#Pretrained model
from tensorflow.keras.applications import DenseNet121
```

Figure 8.10: Importing DenseNet121.

We adjusted the frozen layers by unfreezing the batch normalization layers as we did in ResNet50, and this time the number of blocks that were unfrozen at the top of the model was 110, from layer 317 to 426. The number of trainable

parameters was 3,198,657 out of a total of 8,088,129. Figure 8.11 provides the code of initializing the model, freezing the desired layers and adding the output at the top.

```
#pre-trained weights from imagenet. Load model without output layer, then input_shape required
res_base = DenseNet121(weights = 'imagenet', include_top = False, input_shape = input_shape)

res_base.trainable = False

# un-freeze the BatchNorm layers
for layer in res_base.layers:
    if "BatchNormalization" in layer.__class__.__name__:
        layer.trainable = True

# Unfreeze last blocks
for layer in res_base.layers[317:]:
     layer.trainable = True

# Make sure you have frozen the correct layers
for i, layer in enumerate(res_base.layers):
    print(i, layer.name, layer.trainable)

x = res_base.output
# add a global spatial average pooling layer
x = GlobalAveragePooling2D()(x)
# add a fully-connected layer
x = Dense(1024, activation='relu', kernel_regularizer = 'l2')(x)
x = Dropout(0.5)(x)
# and a fully connected output/classification layer
outputs = Dense(1, activation='sigmoid')(x)

# create the full network
res_model = Model(inputs = res_base.input, outputs=outputs)
```

Figure 8.11: Fine-tuning the DenseNet121 model. First line is for initializing the model. The next one unfreezes the batch normalization layers. The layers from 317 to the last one are also unfrozen. Finally, the output of the model is configured according to our data set.

8.3 Training

After creating an instance of each model, we proceeded to train our four models (customized CNN, VGG16, ResNet50, and DenseNet121). The training process is based on compiling the model and then applying a fit function with the purpose of training the model in the task of finding patterns on the data (Figure 8.12).

```
modelCNN.compile(optimizer=opt,loss="binary_crossentropy", metrics=["accuracy"])

# fits the model on batches with real-time data augmentation:
history = modelCNN.fit(datagen.flow(x_train, y_train, batch_size=batch_size),
                       steps_per_epoch=len(x_train) / batch_size, epochs=epochs,
                       validation_data=(x_test, y_test),
                       callbacks = [reduce_lr, early_stopping])
```

Figure 8.12: Example of how to compile and train a model in Keras.

Once trained, the model can be used to predict to which class, polyp or non-polyp, an image belongs to. Moreover, there is information related to the training which is returned in the form of a variable by the fit function. With this variable, it is possible to plot the functions loss against epochs and accuracy against epochs for both train and test sets (Figure 8.13).

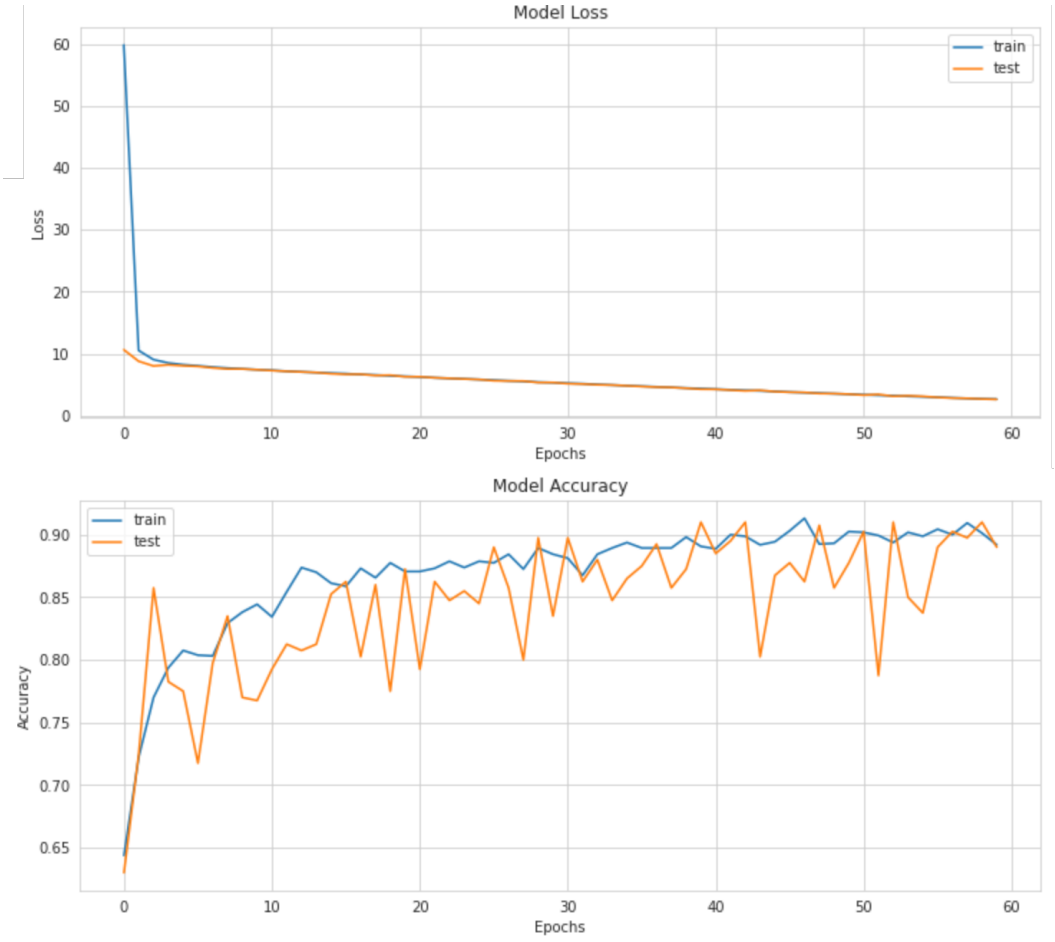


Figure 8.13: Example of plots for loss vs epochs and accuracy vs epochs when training the customized CNN (**Top**) Example of plot of loss vs epochs, (**Bottom**) Example of accuracy vs epochs.

Compiling and training functions receive a set of input parameters. These parameters are considered hyperparameters and they configure how a model must be trained. In Figure 8.12 we can appreciate which parameters we chose.

The most significant parameters are:

- **Callbacks**

  A callback is an object that can perform actions at various stages of training. They automatize tasks after every training/epoch and aid in controlling the training process. This includes stopping training when reaching a certain accuracy/loss score, saving a model as a checkpoint after each successful epoch, adjusting the learning rates over time, and more.

  As shown in Figure 8.14, we used early stopping and reduce LR on Plateau. Early stopping permits us to stop training when a monitored metric has stopped improving. The metric we selected was validation loss with a patience of eight, which means that if there are eight epochs with no improvement, the training stops.

  The second callback, reduce LR on Plateau, reduces the learning rate when a metric has stopped improving. Again, the metric selected was validation loss, with a patience of three and a reduction factor of 10^{-3}.

```
early_stopping = EarlyStopping(monitor='val_loss',
                               min_delta=0,
                               patience=8,
                               verbose=0, mode='auto')
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1,
                               patience=3, min_lr=0.001)
```

Figure 8.14: Declaration of callbacks.

- **Optimizer and learning rate**

  After testing Adam and SGD as possible optimizers, Adam was the one that showed better learning results for all models. The optimal learning rate was 10^4. This learning rate was reduced in case that the validation loss did not improve in three epochs because we used the callback reduce LR on Plateau. Figure 8.15 shows the optimizer used in the four models.

```
lr = 1e-4
opt = keras.optimizers.Adam(learning_rate=lr)
```

Figure 8.15: Declaration of Adam optimizer.

- **Batch size and epochs**

  As it was mentioned in section 7.3, the batch size was 16 for all the models, and the number of epochs was 60 for the customized CNN, and 30 for the rest.

8.4 Ensemble

The purpose of fine-tuning and then training three architectures such as VGG16, ResNet50 and DenseNet121, was to combine their results in to an ensemble. By doing so we improved the accuracy of the model by 1.51% compared to the model with better metrics. It may not seem significant, but it is very complicated to achieve higher accuracy when a model surpasses the 90% threshold.

Before ensembling, we trained each model applying 5-fold cross validation as it was explained previously, then we saved each trained model locally with the purpose of ensembling (Figure 8.16). After comparing the performance of each architecture, we decided to do an ensembling of VGG16, ResNet50 and DenseNet121, excluding the customized CNN which was merely used for understanding how CNNs and Keras work.

```
# save model and architecture to single file
name = 'modelResNet'+str(fold_test)+'.h5'
res_model.save(name)
```

Figure 8.16: Code for saving in local a model that has been trained.

The idea was to create an ensemble for each fold, particularly five ensembles, that combined the three models that were trained with the other folds. We could not create a single ensemble combining all the models, otherwise there would be images from the training set in the test set. Table 8.4 provides an overview of how the ensembling was structured.

	Ensemble 1	Ensemble 2	Ensemble 3	Ensemble 4	Ensemble 5
Models	VGG16 ResNet50 DenseNet121	VGG16 ResNet50 DenseNet121	VGG16 ResNet50 DenseNet121	VGG16 ResNet50 DenseNet121	VGG16 ResNet50 DenseNet121
Train set	Fold 2 Fold 3 Fold 4 Fold 5	Fold 1 Fold 3 Fold 4 Fold 5	Fold 1 Fold 2 Fold 4 Fold 5	Fold 1 Fold 2 Fold 3 Fold 5	Fold 1 Fold 2 Fold 3 Fold 4
Test set	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5

Table 8.4: Models, train set and test set for each of the ensembles.

The code for ensembling was divided into three parts: loading the models, combining the predictions of each model and evaluating the ensemble. The function for loading the models receives a list of strings containing the name of the models, loads them and returns a list of models. The sophisticated part is ensembling. What this function (Figure 8.17) does is predict, for each loaded model, the class of an image belonging to the test set and combines the output of the three models by applying majority voting. For evaluating, we used the confusion matrix and precision, recall and F1. All the results are explained in next section.

```
def ensemble(list_models, x_test):
    ensemble = []
    yhats = [np.where(model.predict(x_test)>0.5, 1, 0) for model in list_models]
    for i in range(len(x_test)):
        predictions = []
        for preds in yhats:
            predictions.append(preds[i][0])
        ensemble.append(mode(predictions))
    return ensemble
```

Figure 8.17: Function for ensembling an odd list of models given the list of images to predict.

8.5 Results

An important part of the research was focused on understanding the results and visualizing them in a clear way. For this purpose, we used the metrics explained in previous chapters, and we created the confusion matrix and AUC-ROC curve to better analyze those values and extract solid conclusions.

8.5.1 Metrics

After training each model, we saved the metrics of interest in an excel file, and we created a table for each CNN (Figure 8.18).

Own CNN										
	Train Acc	Test Acc	Precision	Recall	F1	AUC	TP	FP	TN	FN
fold1	0,8919	0,89	0,9875	0,7900	0,8778	0,96378	0,79	0,01	0,99	0,21
fold2	0,90	0,8894	0,9819	0,8150	0,8907	0,98293	0,815	0,015	0,985	0,185
fold3	0,9019	0,8900	0,8578	0,9350	0,8947	0,96735	0,935	0,155	0,845	0,065
fold4	0,8938	0,8725	0,8311	0,9350	0,8800	0,93913	0,935	0,19	0,81	0,065
fold5	0,9075	0,8975	0,9251	0,8650	0,8941	0,93805	0,865	0,07	0,93	0,135
Mean	89,90%	88,79%	91,67%	86,80%	88,75%	95,82%	86,80%	8,80%	91,20%	13,20%
SD	0,63%	0,92%	7,09%	5,98%	0,80%	1,93%	6,69%	8,16%	8,16%	6,69%

(a) Metrics customized CNN

VGG16										
	Train Acc	Test Acc	Precision	Recall	F1	AUC	TP	FP	TN	FN
fold1	0,981875	0,9375	0,9888	0,885	0,934037	0,99445	0,885	0,010	0,990	0,115
fold2	0,98625	0,95	0,9838	0,91	0,945455	0,991375	0,91	0,015	0,985	0,090
fold3	0,9775	0,96	0,9384	0,99	0,963504	0,99325	0,99	0,065	0,935	0,010
fold4	0,9825	0,91	0,8603	0,985	0,918415	0,978025	0,985	0,160	0,840	0,015
fold5	0,964375	0,90	0,8509	0,97	0,906542	0,977075	0,97	0,170	0,830	0,030
Mean	97,85%	93,20%	92%	95%	93,36%	98,68%	94,80%	8,40%	91,60%	5,20%
SD	0,85%	2,55%	6,59%	4,75%	2,23%	0,86%	4,75%	7,71%	7,71%	4,75%

(b) Metrics VGG16

ResNet50										
	Train Acc	Test Acc	Precision	Recall	F1	AUC	TP	FP	TN	FN
fold1	0,9931	0,8925	0,987578	0,795	0,880886	0,987175	0,795	0,01	0,99	0,205
fold2	0,9769	0,9625	0,969543	0,955	0,962217	0,995625	0,955	0,03	0,97	0,045
fold3	0,9931	0,9550	0,937500	0,975	0,955882	0,99015	0,975	0,065	0,935	0,025
fold4	0,9906	0,9250	0,879464	0,985	0,929245	0,983563	0,985	0,135	0,865	0,015
fold5	0,9862	0,8950	0,849558	0,96	0,901408	0,96605	0,96	0,17	0,83	0,04
Mean	98,80%	93,44%	92,47%	93,40%	92,59%	98,45%	93,40%	8,20%	91,80%	6,60%
SD	0,68%	3,26%	5,88%	7,86%	3,48%	1,12%	7,86%	6,84%	6,84%	7,86%

(c) Metrics ResNet50

DenseNet121										
	Train Acc	Test Acc	Precision	Recall	F1	AUC	TP	FP	TN	FN
fold1	0,9744	0,9125	1	0,825	0,9041096	0,9918	0,825	0	1	0,175
fold2	0,9831	0,9575	0,97906	0,935	0,9565217	0,9927	0,935	0,02	0,98	0,065
fold3	0,9850	0,9600	0,96939	0,95	0,959595	0,990175	0,95	0,03	0,97	0,05
fold4	0,9831	0,9325	0,93035	0,935	0,9326683	0,9839	0,935	0,07	0,93	0,065
fold5	0,9850	0,9150	0,92347	0,905	0,9141414	0,972625	0,905	0,075	0,925	0,095
Mean	98,21%	93,55%	96,05%	91,00%	93,34%	98,62%	91,00%	3,90%	96,10%	9,00%
SD	0,44%	2,26%	3,26%	5,02%	2,47%	0,84%	5,02%	3,25%	3,25%	5,02%

(d) Metrics DenseNet121

Figure 8.18: Summary of the evaluation metrics for every trained model. The metrics selected are: *Train accruacy, test accuracy, precision, recall, F1-score, AUC, True positive, False positive, True negative, and False negative.*

We calculated the mean and standard deviation of each metric because we applied 5-fold cross validation, which means that at the end a model finished having five values per metric. Moreover, we also stored the metrics of the ensemble (Figure 8.19).

Ensemble								
	Test Acc	Precision	Recall	F1	TP	FP	TN	FN
fold1	0,9225	0,994152	0,85	0,916442	0,85	0,005	0,995	0,15
fold2	0,9725	0,994764	0,95	0,971867	0,95	0,005	0,995	0,05
fold3	0,9725	0,965517	0,98	0,9727047	0,98	0,035	0,965	0,02
fold4	0,9475	0,912442	0,99	0,9496403	0,99	0,095	0,905	0,01
fold5	0,9325	0,909953	0,96	0,9343066	0,96	0,095	0,905	0,04
Mean	94,95%	95,54%	94,60%	94,90%	94,60%	4,70%	95,30%	5,40%
SD	2,28%	4,20%	5,59%	2,43%	5,59%	4,55%	4,55%	5,59%

Figure 8.19: Metrics ensemble of VGG16, ResNet50 and DenseNet121.

A helpful method of interpreting the results is comparing the test accuracy, precision, recall and F1-score between the 5 different approaches, as done in Figure 8.20[1].

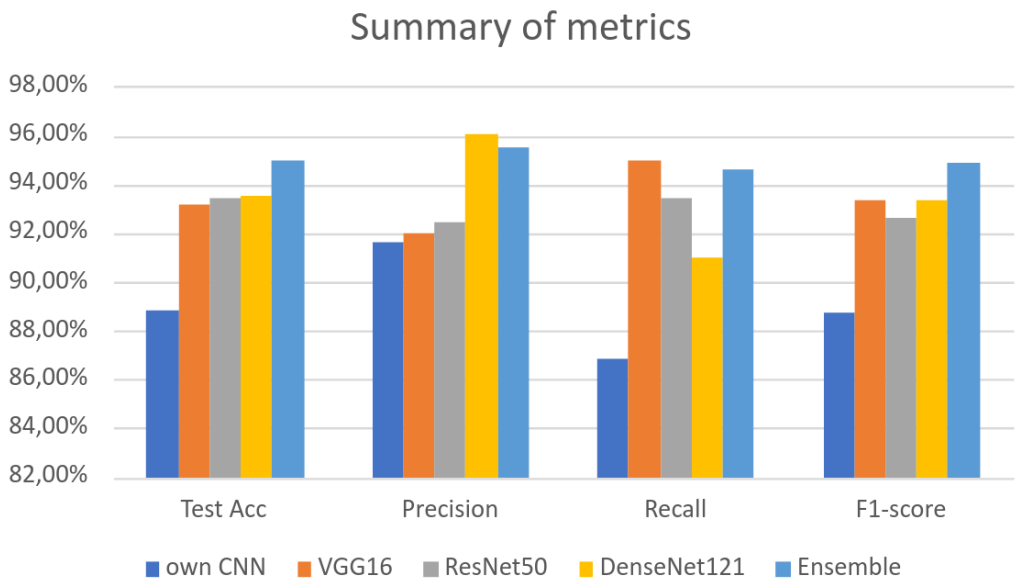


Figure 8.20: Graphic of the metrics obtained from the customized CNN, VGG16, ResNet50, DenseNet121 and the Ensemble.

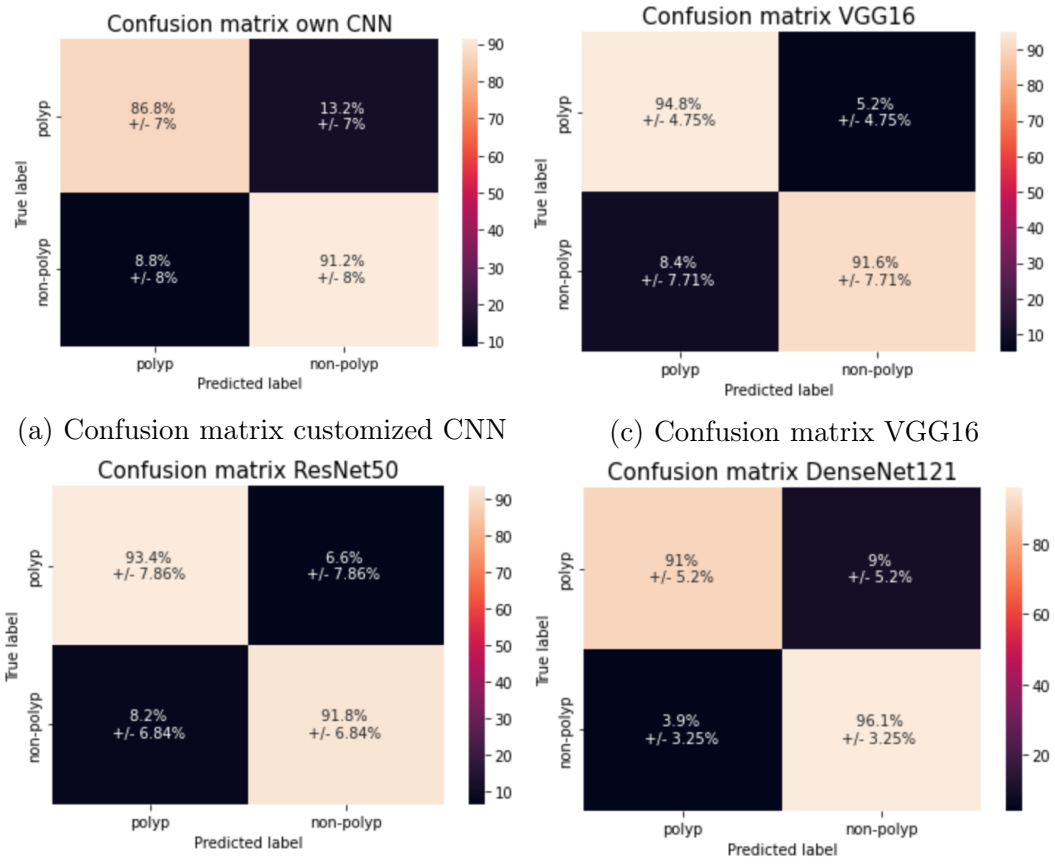
We can appreciate that the metrics of the built CNN fall behind any metric from the pre-trained models. The accuracy of the test set for our CNN is 89.9% ($\mp$ 0.63%), while for the other models it is around 93%, with some deviation. The same happens with the other metrics, for instance recall. The customized CNN struggles with detecting whether an image has a polyp when it actually does. In contrast, the ensemble resulting from combining pre-trained models

[1]The values used in the graphic represent the mean. It is important to bear in mind that the standard deviation is not represented.

works very well. It is what we expected since the ensemble is a combination of the three models with best results.

Between VGG16, ResNet50 and DenseNet151, DenseNet151 is the one with the highest test accuracy and least deviation, even though all three are in the same line. Something odd with DenseNet151 is that it has problems with detecting a polyp in an image since the recall is lower than the rest.

8.5.2 Confusion Matrix



(a) Confusion matrix customized CNN

(c) Confusion matrix VGG16

(b) Confusion matrix ResNet50

(d) Confusion matrix DenseNet121

Figure 8.21: Summary of the confusion matrix for each model that was trained

Figure 8.21 shows the confusion matrix, in the form of mean and standard deviation, of each trained model. We can observe that VGG16 is the model with the best rate of true positives, whereas the built CNN is the one with the worst, 86.8% ($\mp$ 7%). However, the rate of false positives for our own CNN is

quite good, similar to ResNet50 and VGG16.

In the case of DenseNet121, this model has 3.9% (∓ 3.25%) of false positives, which means that the cases it predicts that an image has a polyp when it does not are rare. Although DenseNet121 stands out in true negatives, it is not the same to predict that someone has a polyp and discover later that it was a false positive, compared to predicting that someone does not have a polyp when they actually do. Therefore, it is better to utilize a model with a low rate of false negatives.

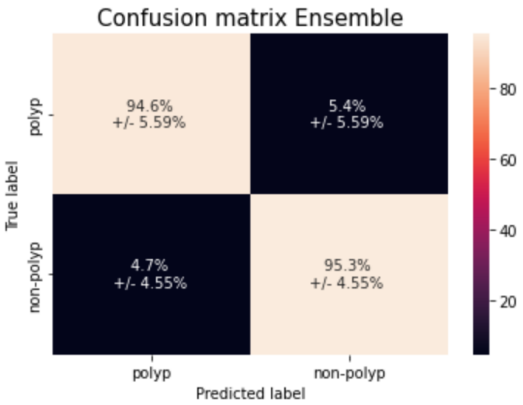


Figure 8.22: Confusion matrix of the ensemble.

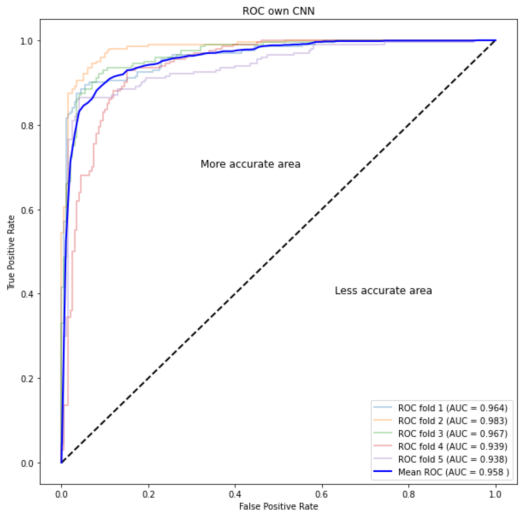
Figure 8.22 shows the confusion matrix for the ensemble of VGG16, ResNet50 and DenseNet121. On one hand, the rate of false negatives is 5.4% (∓ 5.59%), almost as low as the best result achieved by VGG16. On the other hand, the rate of false positives is 4.7% (∓ 4.55%), which is close to DenseNet121 and less than any of the other models. Therefore, the ensemble is a midpoint between the best models. It improves the weakness of one model for predicting one class by adding the vote of another model with better prediction skills.

8.5.3 AUC-ROC Curve

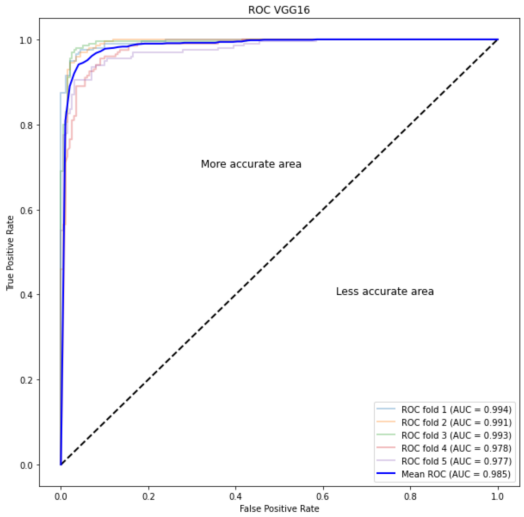
The last metrics we used were ROC curve and AUC[2]. Figure 8.23 presents the ROC curves for our own CNN, VGG16, ResNet50, and DenseNet121. Each graphic has six ROC cruves, five of them belong to the five folds that were used for testing, while the last one is the mean of the previous ROC curves. The legend provided also shows the AUC of each ROC curve.

[2]We can not provide a ROC curve for our ensemble because it is not a trained model, but a calculus. In other words, some trained models give a prediction and then these predictions are combined, that is an ensemble.

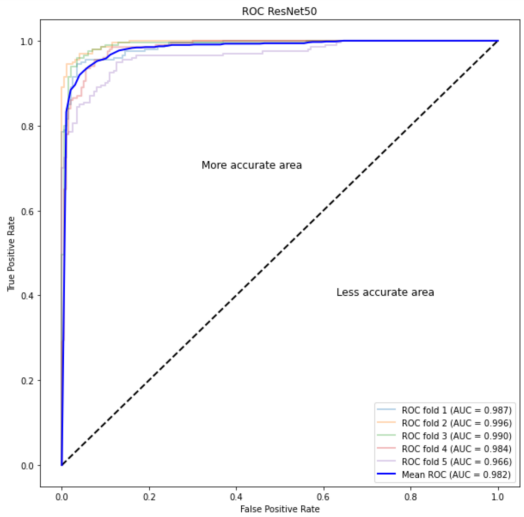
The interpretations we can make from the graphics is that our own CNN has the worst ROC curve, especially for fold four and fold five. In contrast, the rest of the models are along the same lines. All of them struggle with fold four and fold five, but the AUC for them is more than 98%.



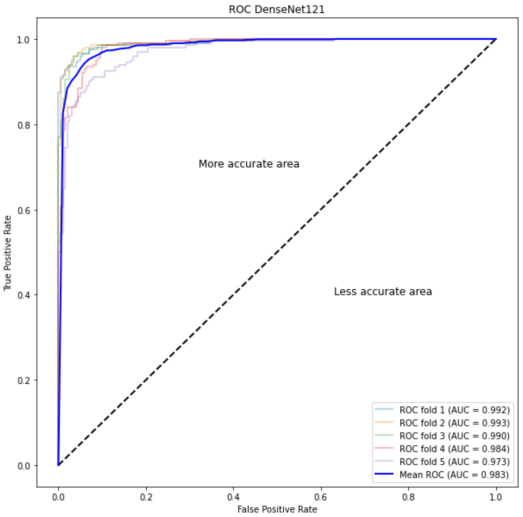
(a) ROC curve customized CNN



(c) ROC curve VGG16



(b) ROC curve ResNet50



(d) ROC curve DenseNet121

Figure 8.23: Summary of AUC-ROC curve for our own CNN, VGG16, ResNet50, and DenseNet121.

Chapter 9
Conclusions

This project had the purpose of using deep learning methods to train a system capable of detecting polyps on colonoscopy images. With that aim, we trained four different models, one of them was built from scratch, whereas the other three (VGG16, ResNet50, and DenseNet121) were pre-trained models that we fine-tuned for this particular project. From the three pre-trained models, we made a combination of their predictions into an ensemble.

After carrying out the experiments, we concluded that when working with a small data set it is better to rely on pre-trained models rather than build up a model. Creating a model took us more time than selecting an architecture and fine-tuning its layers. Even though we did not exhaust all the hyperparameter options, the set of pre-trained models achieved a better accuracy and AUC than our CNN (Table 9.1). That is why our model did not join the others when creating an ensemble. However, our CNN achieved a precision of 91.97% ($\pm$ 7.09%), not far from the 92% ($\pm$ 6.59%) of VGG16.

	own CNN	**VGG16**	**ResNet50**	**DenseNet121**
Accuracy	88.79% ($\mp$ 0.92%)	93.2% ($\mp$ 2.55%)	93.44% ($\mp$ 3.26%)	93.55% ($\mp$ 2.26%)
AUC	95.82% ($\mp$ 1.93%)	98.68% ($\mp$ 0.68%)	98.45% ($\mp$ 1.12%)	98.62% ($\mp$ 0.84%)

Table 9.1: Accuracy and AUC for our own CNN, VGG16, ResNet50, and DenseNet121.

Among VGG16, ResNet50 and DenseNet121, DenseNet121 was the model with the highest accuracy, but only by a matter of decimals. In the case of AUC, the three models were around 98%. However, the precision of DenseNet121 stood out among the rest with a value of 96.05% ($\mp$ 3.26%), and for VGG16 was 92% ($\mp$ 6.59%). In case of recall, it was the other way around and VGG16 surpassed all the others. The advantage of VGG16 is that it has a low rate of false positive, less than ResNet50 and DenseNet121. This is very important in a medical trial because the impact of predicting a false negative or a false positive is not the same. A false positive means that the model predicts a polyp when there is not, but a false negative is equal to predict that there is no polyp when there is.

It is important to consider the total number of paramaters that each of our models have and how many of them are trainable. Table 9.2 provides a summary of these values. Our own CNN was the one with more trainable parameters, but it also had the lowest accuracy. Whereas, DenseNet121 achieved the highest accuracy with the lowest number of trainable parameters. Therefore, there is no a confirmed relation between the number of parameters a model has and how effective it is.

	own CNN	VGG16	ResNet50	DenseNet121
Params	13.5M	15.2M	25.6M	8M
Trainable params	13.5M	7.6M	6.6M	3.1M
Non-trainable params	0	7.6M	19M	4.9M

Table 9.2: Comparison between parameters for our own CNN, VGG16, ResNet50, and DenseNet121.

Regarding ensemble learning, we found combing models to be a useful method to achieve a more accurate and predictive output. The accuracy of our ensemble was 94.95% ($\mp$ 2.28%), the highest among the others. Moreover, the F1-score, a metric that combines precision and recall, was also the best one with a value of 94.9%. Hence, we can say that ensemble learning was the way to go in order to compensate the weakness of a model and obtain better predictive performance than could be obtained from any of the algorithms alone.

9.1 Summary of difficulties

Even though this work shows good results for the task of detecting polyps, there are many other ideas or experiments that are worth researching. Here is a brief description of some of them:

- The first problem was setting up the environment. NVIDIA drivers generally can cause problems when they are installed in Ubuntu. After installing the latest version of our drivers, the gnome was affected and it was not possible to log in the system. Finally, after installing a the 470 version, everything worked fine.

- The main difficulty was the memory of our GPU. Even though the hyperparameters were adapted to the available memory, the kernel constantly died. We tried to do k-fold cross validation in the same session but after some epochs the process ran out of memory and stopped the execution,

making it very hard to train a model. Therefore, we had to use Google colab with the aim of achieving somehow "paralelalism".

- At the beginning we added a resizing layer on every model. We soon experimented that the model did well on classifying images belonging to train set, but quite bad when they belonged to the test set. The reason is because the images on the train set were totally different from the ones in the test set because they were resized twice.

Chapter 10

Future work

The work done in this project has accomplished the objectives proposed at the beginning. Nevertheless, there is some room left for improvement or other paths that could be explored as a continuation to the present work.

This chapter contemplates which of these improvements could be done. Here are some of them:

Firstly, hyperparameter optimization was done manually with a small range of hyperparameters and values. This cast doubt as to whether or not our selection was good enough or if there was a space of hyperparameters that could have increased the predictive capability of a model. Hence, it would be worth to spend time and resources on exploring more combinations by including a hyperparameter optimization algorithm, such as Grid search.

Secondly, one improvement could be to test the model on colonoscopy videos in a clinical setting. Despite how advanced artificial intelligence is, one of the challenges that CAD systems face is their integration in the clinical environment. Thus, an aggregation to this project could be to train the best models with more data to later be tested on real time videos. If the results were significant, the work could develop in a publication.

Finally, we selected pre-trained models that fit inside our time and computational power limitations. However, there are other architectures that have a higher top-accuracy than any of the ones we tested, like EfficentNet. A possible line to continue this research from is to investigate which architectures have been tested in similar works, and study the cost of training them versus their accuracy.

Bibliography

[Bernal 2015] J. Bernal, F. J. Sánchez *et al.* *WM-DOVA maps for accurate polyp highlighting in colonoscopy: Validation vs. saliency maps from physi- cians.* Computerized Medical Imaging and Graphics, vol. 43, pages 99–111, 2015. (Cited on page 4.)

[Bond 2015] A. Bond and S. Sarkar. *New technologies and techniques to improve adenoma detection in colonoscopy.* World journal of gastrointestinal endoscopy, vol. 7(10), pages 969–980, 2015. (Cited on page 3.)

[D.A. Corley 2014] C.D. Jensen D.A. Corley *et al.* *Adenoma detection rate and risk of colorectal cancer and death.* The New England journal of medicine, vol. 370(14), pages 1298–1306, 2014. (Cited on page 3.)

[Grosse 2020] Roger Grosse. *Lecture 9: Generalization*, February 2020. (Cited on page 25.)

[He 2016] K. He, X. Zhang, S. Ren and J. Sun. *Deep Residual Learning for Image Recognition.* 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 770–778, 2016. (Cited on page 31.)

[Hertel 2020] Lars Hertel, Julian Collado, Peter Sadowski, Jordan Ott and Pierre Baldi. *Sherpa: Robust Hyperparameter Optimization for Machine Learning.* SoftwareX, 2020. In press. (Cited on page 54.)

[H.Sung 2021] H.Sung, J. Ferlay *et al.* *Global Cancer Statistics 2020: GLOBOCAN Estimates of Incidence and Mortality Worldwide for 36 Cancers in 185 Countries.* CA: a cancer journal for clinicians, vol. 71(3), page 209–249, 2021. (Cited on page 2.)

[Huang 2016] Gao Huang, Zhuang Liu and Kilian Q. Weinberger. *Densely Connected Convolutional Networks.* CoRR, vol. abs/1608.06993, 2016. (Cited on page 32.)

[Institute 2021] Project Management Institute. Project management body of knowledge. 2021. (Cited on page 38.)

[Keskar 2016] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy and Ping Tak Peter Tang. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima.* CoRR, vol. abs/1609.04836, 2016. (Cited on page 53.)

[Kohli 2017] M.D. Kohli, R.M. Summers and J Geis. *J. Medical Image Data and Datasets in the Era of Machine Learning—Whitepaper from the 2016 C-MIMI Meeting Dataset Session.* J Digit Imaging, vol. 30, pages 392–399, 2017. (Cited on page 7.)

[Liem 2018] B. Liem and N. Gupta. *Adenoma detection rate: the perfect colonoscopy quality measure or is there more?* Transl Gastroenterol Hepatol, vol. 3, page 19, 2018. (Cited on page 3.)

[Moreno 2018] J.F. León Moreno. *ADR evaluation of screening colonoscopies during 2016-2017 in a private health clinic in Peru.* Endoscopy international open, vol. 6(11), pages E1304–E1309, 2018. (Cited on page 3.)

[Simonyan 2014] K. Simonyan and A. Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition.* CoRR, vol. abs/1409.1556, 2014. (Cited on page 30.)

[S.J. Winawer 1993] A.G. Zauber S.J. Winawer *et al. Prevention of colorectal cancer by colonoscopic polypectomy.* The New England journal of medicine, vol. 329(27), pages 1977–1981, 1993. (Cited on page 3.)

[Society 022] American Cancer Society. *Colorectal Cancer Facts Figures 2020-2022*, (Accessed: March 2022). Available at https://www.cancer.org/content/dam/cancer-org/research/cancer-facts-and-statistics/colorectal-cancer-facts-and-figures/colorectal-cancer-facts-and-figures-2020-2022.pdf. (Cited on page 2.)

[Stauffer 022] CM. Stauffer and C. Pfeifer. *Colonoscopy.* (Accessed: March 2022). Available at https://www.ncbi.nlm.nih.gov/books/NBK559274/. (Cited on page 3.)

[Tajbakhsh 2015] N. Tajbakhsh, S. Gurudu and J. Liang. *Automated polyp detection in colonoscopy videos using shape and context information.* Medical Imaging, IEEE Transactions on, vol. PP, no. 99, pages 1–1, 2015. (Cited on page 4.)

[Turing 1950] A. M. Turing. *I.—COMPUTING MACHINERY AND INTELLIGENCE.* Mind, vol. LIX, no. 236, pages 433–460, 10 1950. (Cited on page 17.)

[Wittenberg 2020] T. Wittenberg and M. Raithel. *Artificial Intelligence-Based Polyp Detection in Colonoscopy: Where Have We Been, Where Do We Stand, and Where Are We Headed?* Visceral medicine, vol. 36, pages 428–438, 2020. (Cited on page 4.)

Appendix A

Installation manual

This appendix shows how to set up the working environment to run the code developed in this project. The operating system of our machine is `Ubuntu 20.04.2 LTS`, thus, the instructions commented in the following sections are compatible with Ubuntu, but they may vary for other operating systems.

A.1 DRIVERS

The first step is to install or update the drivers of your Nvidia GPU. You can check which is the latest version compatible with your GPU on the NVIDIA official website or look for the application "Software & updates". Once the application is opened, on the tap "Additional drivers", you should be able to see a list of the available drivers you can install. In our case, we installed the release 470, one of the latest production branch releases of NVIDIA RTX Driver (Figure A.1).

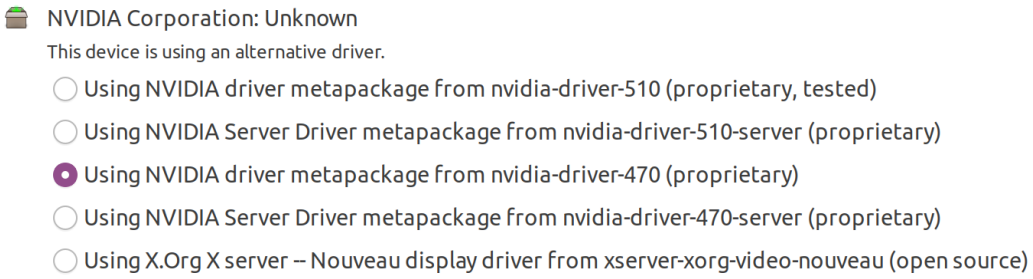


Figure A.1: Example of available drivers to install in a NVIDIA GeForce RTX 3050

After installing the drivers, you must restart your system. Then, if you open a terminal and type the command `nvidia-smi` (Figure A.2), you should be able to see which drivers are installed together with other information related to the GPU. You can also use the graphic interface "NVIDIA X Server Settings".

```
(base) laura@Cerberus:~$ nvidia-smi
Sun Jun  5 12:47:25 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 470.42.01    Driver Version: 470.42.01    CUDA Version: 11.4     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  On   | 00000000:01:00.0 Off |                  N/A |
| N/A   43C    P8     6W /  N/A |      4MiB /  3910MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

Figure A.2: Drivers installed in our system

A.2 NVIDIA TOOLKIT & CuDNN

CUDA

The next step is to install the CUDA Toolkit by accessing the NVIDIA website and selecting the options according to your operating system (Figure A.3).

Figure A.3: Options to install CUDA

After this, you should be able to see the commands that you have to enter in your terminal in order to install CUDA. To confirm whether CUDA is working, run `nvcc -version` (Figure A.4).

```
(base) laura@Cerberus:~$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Wed_Jun__2_19:15:15_PDT_2021
Cuda compilation tools, release 11.4, V11.4.48
Build cuda_11.4.r11.4/compiler.30033411_0
```

Figure A.4: Command to check whether CUDA is installed

CuDNN

Beside CUDA, you also need to install CuDNN. The first step is to go to the NVIDIA website and select "Download cuDNN". They will ask you to log in (having an NVIDIA account is a requirement to download CuDNN). Once in, select which CuDNN you want to download depending on your CUDA version. Since we have CUDA 11.4, we would download the v8.4.1, as shown in Figure A.5.

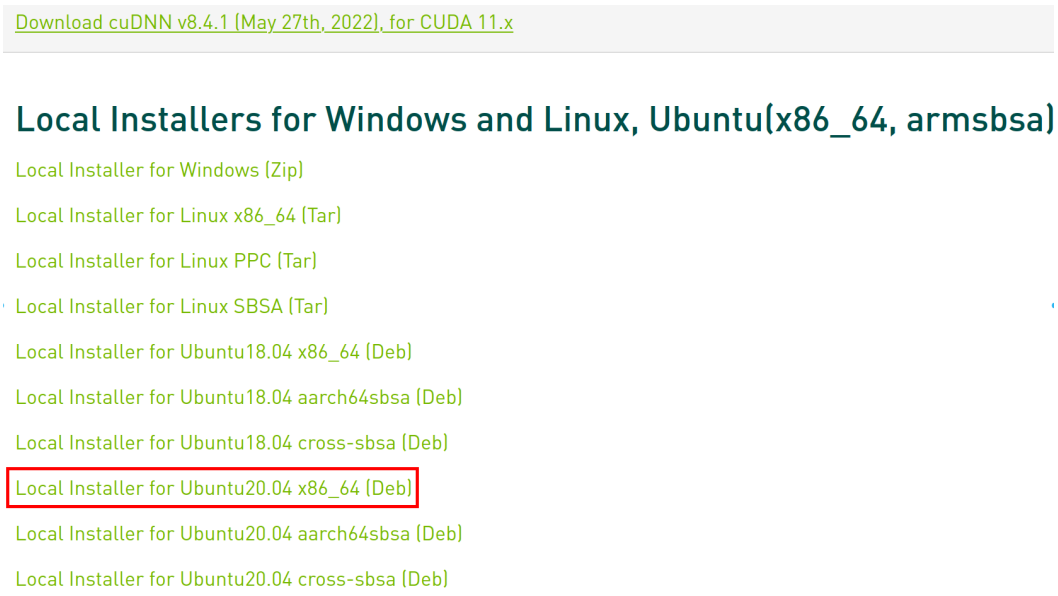


Figure A.5: Command to check whether CUDA is installed

After downloading CuDNN, unzip the file and copy the contents of the sub-folders to the path where you installed CUDA:

```
tar -xzvf cudnn-11.4-linux-x64-v8.4.1.tgz
sudo cp cuda/include/cudnn*.h /usr/local/cuda/include
sudo cp cuda/lib64/libcudnn* /usr/local/cuda/lib64
sudo chmod a+r /usr/local/cuda/include/cudnn*.h
/usr/local/cuda/lib64/libcudnn*
```

A.3 ANACONDA

We chose Anaconda because it comes with conda, a package, and environment manager. Anaconda includes Python, and over 150 scientific packages and their dependencies.

Installing anaconda is very simple. The first step is to download the installer from the Anaconda official website. Secondly, enter the following command on a terminal:

```
bash ~/Downloads/Anaconda3-2020.02-Linux-x86_64.sh
```

After this, Anaconda should be already installed. You can open the graphic interface by typing `anaconda-navigator` on a terminal. From there it is possible to administrate the environments (Figure A.6), and install any library. The same can be achieved via terminal.

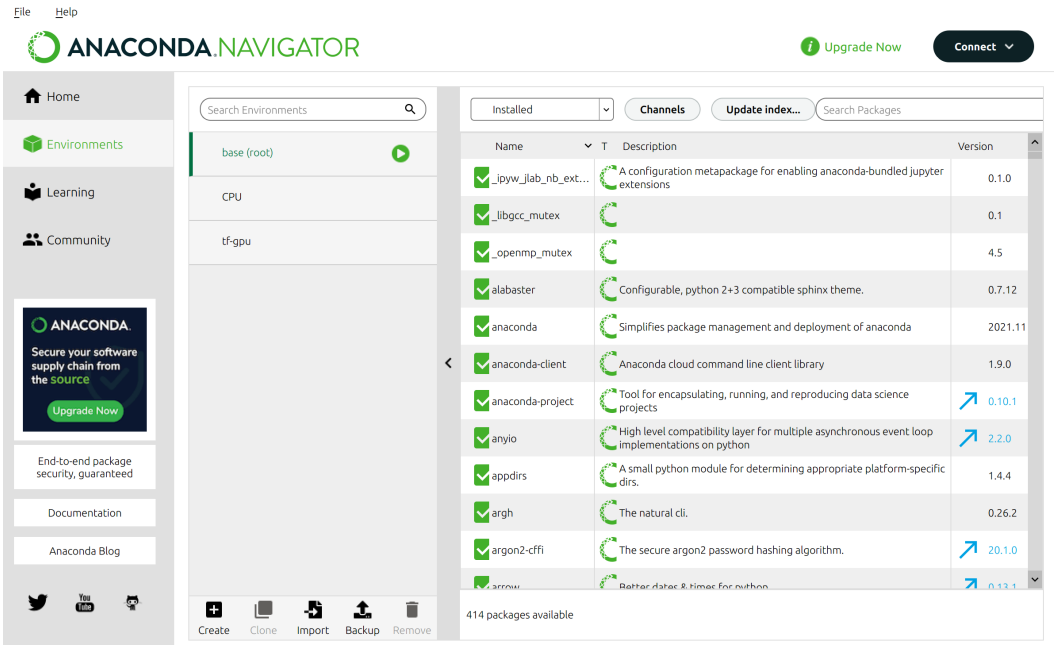


Figure A.6: Anaconda graphic interface and its feature of administrating environments

The libraries that you should have installed in an anaconda environment to run this project are:

- sklearn
- seaborn

- matplotlib
- pandas
- opencv
- tensorflow
- keras
- numpy

All that is left is open jupyter notebook, which comes with Anaconda, on a browser and choose in which environment you want to run the particular notebook.

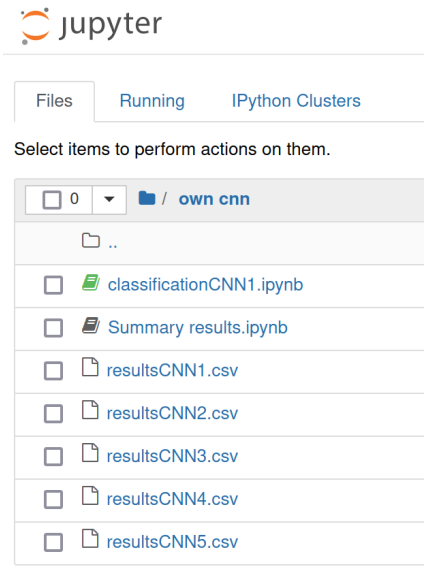


Figure A.7: Jupyter notebook opened in a browser.