CENTRO DE INGENIERÍA

Y DESARROLLO INDUSTRIAL

UNIDAD QUERÉTARO

POSGRADO INTERINSTITUCIONAL EN CIENCIA Y TECNOLOGÍA

Characterization and estimation of lung nodule malignancy using 3D Convolutional Neural Networks in Low-Dose CT

A THESIS SUBMITTED FOR THE DEGREE OF

MASTER OF SCIENCE

Automatic Control and Dynamical Systems

PRESENTS

Eng. Jonathan Domínguez Aldana

ADVISORY BOARD

PhD Eloy Edmundo Rodríguez Vázquez

PhD Nayeli Camacho Tapia

MD Eduardo Hernández Rangel

MD Adrián Antonio Negreros Osuna

November 16, 2019

Santiago de Querétaro Qro.

Contents

List of Figures

List of Tables

Acknowledgments

To my mother, for all the effort, dedication and compromise, for all the countless hours of hard work. Thank you for being my support, not only in these last two and a half years , but since the day I was born. Thank you for everything, I love you mom.

To my advisors, Eloy and Nayeli, for all their patience, for all their support and for their countless advice. Not only did I find in you two wise teachers, but also great human beings and friends. Thank you for being the two pillars on which this project was supported.

On the other hand, I would like to thank my advisor Eduardo, for giving me an opportunity to collaborate with me even without knowing me. I thank you for all those long distance calls, for helping me remember that all we do is to help others, a vision I will always try to remember. I thank Adrián for his trust and knowledge. Above all, I thank you for that time when we met in Boston, and you told me that Mexicans can also compete with the best people in the world, it's just a matter of changing our mind set.

I would like to extend my gratitude to Iván, for sharing his experience in the branch of artificial vision, for being my external advisor and my friend. Also, I would like to tell you that thanks to

your classes, I decided to focus my professional career in the area of machine learning.

Also, I give special thanks to all the medical doctors who helped me in the construction of the CT database in Querétaro: Ricardo Veloz, Ricardo López, Andrea Aguilera, Martha Lidia Elizalde, Néstor Piña, Rafael Francisco Páramo, Violeta Cortés, Lilian Camacaro, Ernesto Cordero, Miguel Isaías Paredes, Jesús Salas, Paulina Rojas, Francisco González and Mario Rosas. Without your support and advice, this project would not have an impact on Mexican patients.

I thank my father and my brothers for all their understanding and patience. In particular, I thank you for all your love and support in these two and a half years. I love you so much.

Finally, I would like to thank my friends for understanding my absence at most meetings. I will always thank you for your friendship, you know that you will always be my second family.

Agradecimientos

A mi madre, por todo el esfuerzo, dedicación y entrega, por las incontables horas de trabajo arduo. Gracias por ser mi soporte, no solo en estos últimos dos años y medio de maestría, pero en todo momento desde el día en que nací. Gracias por todo, te amo mamá.

A mi asesores, Eloy y Nayeli, por toda su paciencia, por todo su apoyo y por sus incontables consejos. No solo encontré en ustedes dos sabios profesores, pero también grandes seres humanos y amigos. Gracias por ser los dos pilares en los que se soportó este proyecto.

Por otra parte, agradezco a mi asesor Eduardo, por darme una oportunidad de colaborar conmigo aún sin conocerme. Te agradezco por todas esas llamadas a larga distancia, por ayudarme a recordar que todo lo que hacemos es para ayudar a los demás, una visión que siempre trataré de recordar. A Adrián le agradezco su confianza y su conocimiento. Sobre todo, te agradezco aquella vez que nos vimos en Boston y me dijiste que los mexicanos también podemos competir con los mejores del mundo, sólo es una cuestión de cambiar nuestra mentalidad.

Quisiera extender mi gratitud a Iván, por compartirme su experiencia en la rama de visión artificial, por ser mi consejero externo y mi amigo. Así mismo, aprovecho la oportunidad para

expresarte, que gracias a tus clases, decidí enfocar mi carrera profesional en el área de aprendizaje automático.

De igual forma, quisiera dar agradecimientos especiales a todos los médicos que me ayudaron en la construcción de la base de datos tomográficos en Querétaro: Ricardo Veloz, Ricardo López, Andrea Aguilera, Martha Lidia Elizalde, Néstor Piña, Rafael Francisco Páramo, Violeta Cortés, Lilian Camacaro, Ernesto Cordero, Miguel Isaías Paredes, Jesús Salas, Paulina Rojas, Francisco González y Mario Rosas. Sin su apoyo y sus consejos, este proyecto no tendría un impacto para los pacientes mexicanos.

Agradezco a mi padre y a mis hermanos por toda su comprensión y paciencia. En especial, les agradezco todo su cariño y soporte en estos dos años y medio. Los quiero mucho.

Finalmente, quisiera agradecer a mis amigos por entender que muchas veces no podía estar presente en las reuniones. A ustedes siempre les agradeceré su amistad, saben que siempre serán mi segunda familia.

Abstract

Lung cancer has posed a major challenge for health institutions all around the world. Economic and social implications derived from this disease result in efforts to reduce mortality rates and establish better diagnostic procedures. Specifically, technologies such as low-dose computerized tomography (CT) have been implemented to increase early detection accuracy of lung cancer. Even though, there are still major challenges to improve sensitivity and specificity rates of lung cancer prognosis using CT. More precisely, false positives and negatives are still present in the prognosis procedure. In consequence, there are psychological, economic and social problems associated with false positive and negative rates. Some of these problems include economic costs for families and health institutions, patient anxiety, and potential risks of morbidity and/or mortality. Moreover, false negatives represent the main problem due to the potential irreversible consequences that could arise, where survival rates decrease considerably and paliative care is the only alternative to reduce patient suffering. To address these issues, several research studies have developed different Computer Aided Diagnostic (CAD) tools. Thus, the objective of this study is to investigate all related work to develop a better CAD system for automatic lung cancer detection that will help radiologists with CT assessment; and ultimately, will reduce the number of false positives and negatives.

By analyzing all related studies, three main constraints have been identified: 1) few studies have focused on the entire process of lung cancer detection, 2) the best results for each stage of the estimation process were obtained using a lot of computational resources, and 3) state of the art models (for each stage) implement ensemble methods containing a lot of parameters, where many Convolutional Neural Networks (CNNs) are trained, thus increasing model complexity. Therefore, a sequential 3D Convolutional Neural Network (CNN) architecture was proposed to make an end-to-end lung cancer estimation. Four principal components comprised the algorithm: 1) a nodule detector, 2) a malignancy classifier, 3) a morphological nodular predictor, and 4) the final cancer estimator. Only one classic CNN was trained for each one of the first three stages, while the last component was implemented using a classic Deep Neural Network. Moreover, all training was performed on constrained hardware. Specifically, a NVIDIA GeForce GTX 1060 GPU, 24 GB of RAM and an Intel Core 17-7700HQ CPU were used.

A particular Hyperparameter Tuning technique, the ”Tree-structured Parzen Estimator”, was used for selecting the appropriate network topology for each component. Data for training and validation was obtained from the open source Lung Image Database Consortium (LIDC), while independent data was collected from three hospitals located in the state of Querétaro, in Mexico. Several standard preprocessing techniques were applied before training and validating the networks. Also, augmentation techniques were implemented during training, such as 3D lossless transformations and ”Generative Adversarial Networks”. A 10-cross fold validation was defined to evaluate the network’s performance at each stage, where four metrics were specified: sensitivity, specificity, F1 score, and ROC curve.

Results for the nodule Candidate Generation CNN were successful giving a final sensitivity of 94.1% and an average of 450 FP /CT with a binarization threshold of 0.1. On the other hand, the FPR model did not perform as expected, giving as a result a sensitivity of 0.25 and an average of 41.8 FP /CT. Therefore, efforts must focus in optimizing and improving the FPR model performance. For the spiculation and lobulation estimators, the results were promising giving final sensitivities of 0.9965 and 0.8358 respectiveley, as well as final specificities of 0.9984 and 0.9240. Another

successful result was given by the malignancy classifier, which reported a sensitivity of 1 and a specificity of 1. Although these last results sound promising, further testing must be done to verify them. Finally, the cancer estimator reported a sensitivity of 0 and specificty of 1.

As illustrated in the results, the main problem that was encountered during the research was the development of the false positive reduction stage. Final experiments showed that the best FPR models were obtained when heavy data augmentation was applied to the data, specifically when random nodular translations were included. An important remark resides in the probability estimations computed by almost all of the developed FPR models. Most of the highest detection rates per CT slice were obtained outside of the CT scan. It is possible that with the addition of preprocessing algorithms, all undesired CT regions could be removed to avoid these false positive findings. Poor results may also be attributed to the exclusion of repeated annotations of the same nodule provided by all four radiologists from database A. Thus, it is possible that the restrictions imposed for defining a true positive were too flexible.

An important highlight is to illustrate the sequential design of the algorithm workflow. Due to this sequentiality, many of the poor results obtained in previous steps are reflected in further stages of the algorithm. Specifically, the results obtained for the cancer estimator are explained by this architecture design. Also, one of the main limitations of this research study was the hardware. Although many of the trained models were inspired on more complex architectures, such as ResNet, Inecption Net, Vgg and Inception-Resnet, their abstraction capabilities cannot be compared to the original ones. Even though, this limitation was seen as another contribution of this thesis, to achieve similar results by training models with fewer parameters.

This works has proved the value of CAD models to aid in the detection of lung cancer with some degree of accuracy. It is expected that better technologies will be developed in the future, helping radiologists detect lung cancer in early stages to provide immediate healthcare to patients.

CHAPTER 1

Introduction

1.1 Problem statement

Based on estimates from GLOBOCAN 2018 there are 18.1 million new cancer cases and 9.6 million deaths worldwide associated to the disease. Specifically, lung cancer is the most commonly diagnosed carcinoma in the world representing the highest mortality rate for both genders [1]. In Mexico, since 1998 lung cancer has occupied the first position in mortality rate with respect to other cancers. Fig. 1.1 illustrates different cancer death statistics in Mexico since 1998 [2]. Furthermore, based on the World Cancer report 2014, diagnostic procedures for lung cancer are of great relevance for reducing deaths and increasing patient survival rates. In this case, Low-Dose Computerized Tomography (CT) of chest is the preferred choice to make an initial prognosis [3, 4].

The National Lung Screening Trial (NLST) made a comparison in diagnostic accuracy between the CT and chest X-ray. A sample of 53,454 male and female smoker patients, between 55 and 74 years old, was gathered to make the comparison between both technologies. Results showed a mortality rate decrease in lung cancer of 16% using CT, with respect to chest X-ray [5]. Also,

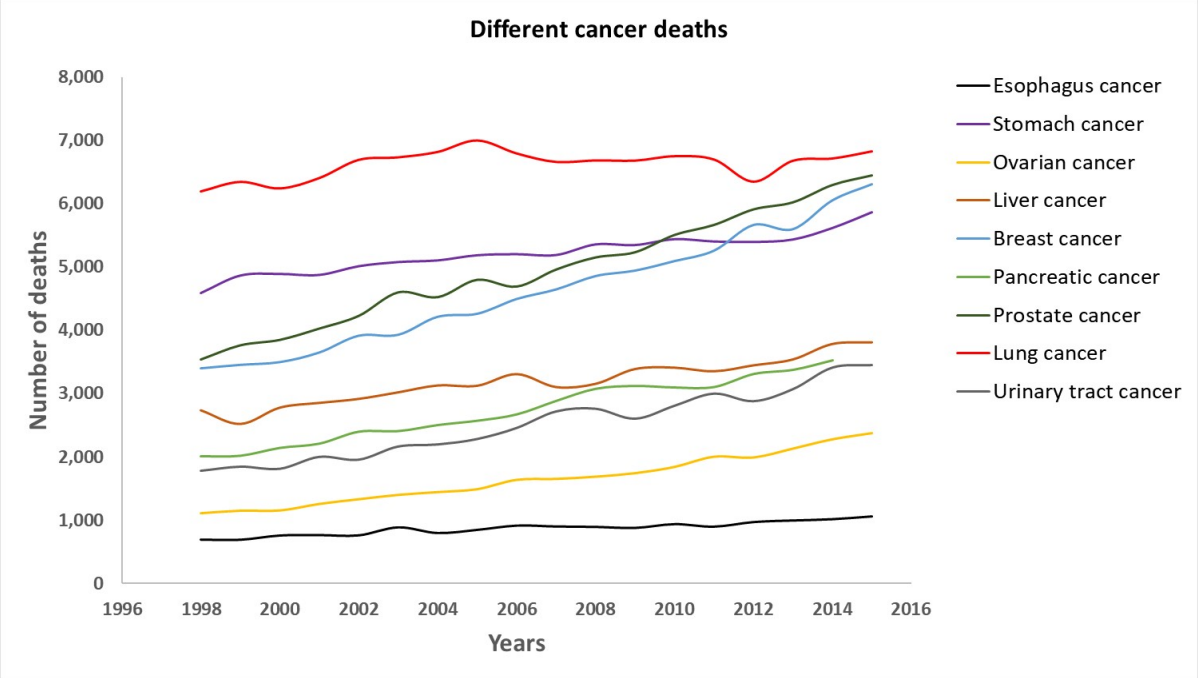

Fig. 1.1. Cancer death statistics in Mexico for several types of carcinomas from 1998 to 2015.

a research study showed the presence of 23% false positives (FP) and a 0-20% range of false negatives (FN) produced by the CT evaluation in the NLST assessment. A specificity of 73.4% and a sensitivity of 93.8% were obtained during the research [6]. Another study exposed that approximately 35.6% of patients will receive at least one FP screen in a three years examination period, of whom 18 will have an invasive procedure [7].

Besides false positives, it is well known that diagnostic accuracy is also related to the radiologist's background and years of expertise. For example, two radiologists can give different diagnosis results even if they analyzed the same CT; a problem referred as inter-grader variability [8]. One of the possible reasons related to these differences consists in the radiologist fatigue, although further research is needed to confirm it. Consequently, there can also be an increase in the false negative rate if the medical doctor is tired [9].

1.2 Motivation

Patients diagnosed with lung nodule false positives may experience anxiety. Furthermore, false positives represent a risk increase in morbidity and/or mortality due to unnecessary diagnostic tests and treatments, deriving in economic implications that directly affect the patient [7].

A statistical research was designed to analyze a sample of 1087 patients, from the "PLCO Screening Trial", to assess the medical and non-medical costs associated with false positive rates presented in lung, prostate, colorectal, and ovarian cancer screens [10]. In this study, 43% of subjects presented at least one FP, from which 83% received follow-up care. The adjusted mean difference of costs between patients, who presented at least one FP and the ones who didn't, was estimated to be $424,539 US dollars. As a result, there is a pressing need to increase sensitivity and specificity of lung cancer assessment. A potential solution is to develop a Computer Aided Diagnostic (CAD) tool.

There are three main reasons to conduct the research and develop a CAD system:

- Reduction of lung nodule false positive and negative rates encountered in low-dose CT.
- Develop an objective framework to avoid subjective assessments in the diagnosis of lung cancer.
- Reduction in the time invested by the radiologist to detect and characterize lung nodules in low-dose CT.

1.3 Research questions

Three main research questions are formulated to guide this study:

1. What morphological nodular characteristics will allow the automatic detection and characterization of pulmonary nodules using a machine learning algorithm?

2. Would the automatic detection and characterization of lung nodules help the radiologist reduce false positives and negatives while assessing a CT?

3. Would dividing the malignancy estimation problem in different stages result in better performance of the machine learning algorithm?

1.4 Hypothesis

If a sequence of interconnected 3D Convolutional Neural Networks is developed to detect and characterize pulmonary nodules in low-dose CT, with a sample size of at least 1000 CT for training and testing, then at least 90% sensitivity and 90 % specificity of nodular malignancy estimation can be attained.

1.5 Objectives

1.5.1 General objective

To detect and characterize lung nodules to estimate nodular malignancy probability in low-dose CT with a specificity and sensitivity of at least 90% using a machine learning algorithm based on 3D Convolutional Neural Networks.

1.5.2 Specific objectives

1. Preprocess all computerized tomography.

2. Define the pipeline architecture in two stages and develop the mathematical model:

   (a) Design and implement the nodule detector.

   (b) Develop the nodular characteristics estimators.

3. Validate and compare the results with other studies using four different metrics: sensitivity, specificity, F1 score, and ROC curve.

1.6 Scope

The study was developed for a time period of two years at ”Centro de Ingeniería y Desarrollo Industrial” (CIDESI), in collaboration with Universidad Anáhuac Querétaro. All computation tasks were done using the following hardware specifications: a Graphics Processing Unit (GPU) NVIDIA GTX 1060, an Intel Core i7-7700 HQ Central Processing Unit (CPU), and 32 GB of Random Access Memory (RAM). A sample size range, between 500 and 2,000 CT, was defined for analysis. Two types of patients were considered to be included in this research study:

1. Patients with lung nodules greater than 3 mm in maximum diameter that were assessed as benign by expert radiologists (minimum 6 years of experience).

2. Patients with primary lung tumors and/or metastatic carcinomas present in the lungs. Specifically, selection criteria was specified using the clinical staging conventions provided by the International Association for the Study of Lung Cancer (IASLC) [11]. Therefore, Stages 0, IA, IA1, IA2, IA3, IB, IIA, IIB, and IIIA were included in the analysis. More advanced lung cancers were not considered, since such lesions are evident and easily identified in the CT.

Clinical specifications for both patients were defined as: either sex, gender, and ethnic origin. The selection criteria was chosen to generalize detection capabilities, thus allowing the algorithm to be implemented in any clinical context.

CHAPTER 2

Theoretical framework

2.1 Lung cancer and computerized tomography

2.1.1 An overview of lung cancer

The origin of the term cancer goes back to Ancient Greece with Hippocrates, who defined the concept as *carcinos* to describe tumor formation due to ulcers. Such description is attributed to the crab's structure whose projections are similar to the morphology presented by the disease [12]. In general, cancer refers to any malignant tumor formed by unusual cell division, which can penetrate their surrounding organs; a process denominated as metastasis. Generally, tumors can be classified into two types: malignant and benign. Any tumor is considered benignant if it is enclosed and isolated from its surroundings. On the other hand, malignant tumors are characterized by a fast unregulated cell multiplication, which can invade adjacent nearby tissues [13].

It is well known that unregulated cell division of cancer is caused by mutations in specific genes, therefore all cancers are referred to as diseases of the genome [3]. Specifically, it has been reported

that 50% of all cancers are caused by mutations in the p53 tumor suppressor gene [14].

Etiology of the disease is extremely relevant to understand mutation causes. There are several risk factors associated to cancer development, where tobacco smoking represents the most important factor up to date. Other factors include chronic infections, diet, alcohol intake, sun exposure, environmental pollution, and exposure to hazardous environments [3]. The World Health Organization has classified risk factors into two categories: genetic and external. The latter has been subsequently classified into physical, chemical, and biological carcinogens [13].

As defined previously, smoking is considered the most important risk factor related to different types of cancer development. Furthermore, cigarette smoking has been proven to be the main cause for lung cancer in developed countries [3]. Based on The World Cancer Report 2014, lung cancer is one of the most aggressive human cancers presenting a 5-year overall survival of 10-15%. Also, lung cancer has a high incidence rate in men and women presenting 18.1 million new cancer cases and 9.6 million cancer deaths worldwide just in 2018 [1, 3].

All classification procedures of the disease have been based primarily on morphological features. Reports have divided lung cancer into four distinct histological types: adenocarcinoma, squamous cell carcinoma, small cell carcinoma, and large cell carcinoma [3]. Also, several efforts have been focused in differentiating disease staging. Therefore, the International Association for the Study of Lung Cancer (IASLC) has published the eighth edition for lung cancer staging, where all classification results with their respective descriptors are shown in Table 2.1. The research article provides a more detailed explanation of the chosen descriptors (N=node, T=tumor and M=metastasis) and the selected criteria for classifying lung cancer staging [11].

As reported by the IASLC, nodule evaluation is a crucial task for making an initial assessment of the patient. In consequence, the American College of Radiology has published an assurance tool for evaluating lung nodules in CT [15]. Generally, all evaluation efforts are focused in evaluating solitary pulmonary nodules (SPN) because the presence of multiple nodules is evidence of metasta-

Table 2.1. Staging procedure for lung cancer development based on three different descriptors tumor, node and metastasis.

Staging Group	T	N	M
Occult carcinoma	TX	N0	M0
Stage 0	Tis	N0	M0
Stage IA1	T1a(mi)	N0	M0
	T1a	N0	M0
Stage IA2	T1b	N0	M0
Stage IA3	T1c	N0	M0
Stage IB	T2a	N0	M0
Stage IIA	T2b	N0	M0
Stage IIB	T1a-c	N1	M0
	T2a	N1	M0
	T2b	N1	M0
	T3	N0	M0
Stage IIIA	T1a-c	N2	M0
	T2a-b	N2	M0
	T3	N1	M0
	T4	N0	M0
	T4	N1	M0
Stage IIIB	T1a-c	N3	M0
	T2a-b	N3	M0
	T3	N2	M0
	T4	N2	M0
Stage IIIC	T3	N3	M0
	T4	N3	M0
Stage IVA	Any T	Any N	M1a
	Any T	Any N	M1b
Stage IVB	Any T	Any N	M1c

sis, which indicates a high probability that the tumor does not have its origins in the lungs [16].

A concise definition of a solitary pulmonary nodule is provided in [16], specifying it as a single lesion in a round or oval shape with a diameter ≤ 3 cm in lung parenchyma, surrounded entirely by gas-containing lung tissue. Any lesion having a diameter greater than 3 cm is considered a pulmonary mass. To differentiate a SPN as benign or malignant, there are five factors to be considered [16–18]:

- **Clinical factors:** all solitary pulmonary nodules in non-smoker patients, younger than 35 years old, and without cancer history are considered benign lesions: granuloma, hamartoma (benign malformations) or inflammatory lesion [17].
- **Growth Pattern:** growth development of a SPN is measured in *doubling time*, which is the time it takes for a nodule to double its volume. Any SPN with *doubling time* less than 1 month (infectious lesions) or greater than 2 years (hamartomas) have a high probability of being benign [17]. On the other hand, a lesion presenting a *doubling time* between 30 to 400 days tends to be malignant [16].
- **Size:** a study showed that the probability of malignancy is less than 1% if the nodule diameter is less than 4 mm, 0.9% if it is between 4 and 7 mm, 18% if the size oscillates in a range of 8-20 mm, and 50 % for diameters between 20 and 30 mm [16].
- **Border Characteristics:** malignant nodules are usually associated with irregular, spiculated, and lobulated contours. For example, a research defined a logistic regression model that estimated a probability between 88% and 94% that a nodule is malignant given that it was lobulated [16]. With respect to spiculation, the term *corona radiata* has been used to describe it as a set of linear densities that radiate from the edge of a nodule into the adjacent lung [17]. Moreover, SPN that are neither spiculated or lobulated present a high probability of being benign [17].
- **Density:** nodule calcification is an important characteristic for differentiating between malignant and benign lesions [16, 17]. A research conducted a study with a sample of 504 patients

with calcified nodules. Results comprised 97% and 3% of benign and malignant lesions respectively. The same study, using a sample of 1109 patients with non-calcified nodules, reported 29% benign lesions and 71% of malignant anomalies [19]. Moreover, there are 5 different types of calcification patterns to detect a benign lesion: central, complete, concentric, laminated, and pop-corn shape [17].

2.1.2 Computerized tomography

Many factors and parameters must be evaluated to make the best possible assessment of SPN, and the best tools must be used to achieve this goal. As described previously, CT has been the preferred choice over chest radiography for lung cancer risk assessment based on the NLST study that compared mortality rates for both technologies [20]. The history of computerized tomography goes back to 1967 with Godfrey Hounsfield, an engineer that developed the first modern CT scanner [21].

Basic principles behind CT screening define a subject to be scanned as being divided into axial slices [22]. A typical CT scanner consists generally of two components that are rigidly linked: an X-ray tube and an X-ray detector located in the opposite side. Both components scan across the subject with a linear translation movement. Many X-ray transmission measurements, at different locations of a specific slice, are performed while both components are moving linearly. When translation is finished for that specific slice, the system is rotated by some angle around the subject. Next, the translation movement is repeated over the patient for the same slice. A complete slice scan is finished when the system has reached 360°. Finally, the process is repeated for each slice comprising the patient [22].

Nowadays, there is a wide variety CT scanner models with different slice thickness resolutions and clever algorithmic solutions for image reconstruction. Even though there have been technology advancements for lung cancer screening, there are still challenges for improving diagnostic sensitivity and specificity [5, 6, 8]. Several solutions have been proposed to address these issues and are defined as Computer Aided Diagnosis (CAD) tools.

2.2 Deep learning and neural networks

With the advent of big computational power resources, availability of a huge amount of data, and better algorithms, deep learning has been a great success in many areas where pattern recognition is required [23–27]. As a consequence, deep learning has been proposed as an excellent CAD tool for the evaluation of medical images. Actually, this area of study has been applied recently to tackle many challenges posed in the prognosis of several diseases using medical imaging techniques [23, 28–30].

2.2.1 Machine learning and neural networks

Deep learning is one of the most recent topics developed in machine learning, which is considered one of the seven disciplines of artificial intelligence [31]. Defining precisely the concept of machine learning is difficult because there have been slight changes in its definition throughout history. For example, Peter Norvig and Stuart J. Russell conceive it as the ability of a computer to adapt to new circumstances, to detect and extrapolate patterns [31]. As exposed by Arthur Samuel, machine learning can also be defined as the field of study that gives computers the ability to learn without being explicitly programmed [32]. A definition that is more suitable for recent times is the one provided by Stephen Lucci and Danny Kopec, who defined it as the process by which a computer distills meaning due to exposition of training data [33].

Within machine learning there are three major areas: supervised, unsupervised and reinforcement learning. Given a training set $(\vec{x}^{(1)}), y^{(1)}), (\vec{x}^{(2)}, y^{(2)}), ..., (\vec{x}^{(m)}, y^{(m)})$ where each $y^{(i)}$ was generated by an unknown function $y = f(x)$ and $x^{(i)} \in \mathbb{R}^n$ represent the vector of n inputs for the i^{th} training example, the main objective of supervised learning is to find a function $h(x)$ that approximates $f(x)$ [31]. This problem can be stated in two different tasks depending on the outputs $y^{(i)}$. If the output vector $\vec{y}$ is comprised by a finite number of discrete categories, then approximating f is stated as a classification problem. On the other hand, if the output vector has continuous variables as components, then approximating f becomes a regression task [34]. For this thesis, unsupervised and reinforcement learning are irrelevant, but more information is covered in [31, 34, 35].

Several algorithms have been developed in the supervised learning regime, such as linear and logistic regression, neural networks, support vector machines (SVM), k-nearest neighbors, the perceptron, kernel methods, among others [31, 34, 35]. Specifically, neural networks (NN) are of great relevance in the context of this thesis.

History of NN goes back to 1943 with Warren McCulloch and Walter Pitts, who presented the first theoretical model of a neuron in their paper "A logical calculus of the ideas immanent in nervous activity" [36]. Such model computes a binary sum of the inputs contained in $\vec{x}^{(i)}$ and outputs a predicted value $\hat{y}^{(i)} \in [0, 1]$ depending if a certain threshold is reached. Thus this artificial neuron has the ability to represent any of the three logical operators: AND, OR and NOT.

Later on, in 1949, psychologist Donald Hebb posed the basis for neural learning by stating [37]: *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."*

Thanks to Hebb's Rule and the theoretical model of McCulloch and Pitts, Frank Rosenblatt developed the basic artificial neuron: The Perceptron [38]. Hebbian learning was represented by weighting each feature contained in a single training example $\vec{x}^{(i)}$ and adjusting them based on a simple update rule: the proportional difference between the estimated output and the observed value $e = y^{(i)} - \hat{y}^{(i)}$, which represents the miss-classification error for the i^{th} training example.

Therefore, the basic model of the perceptron consists on a "squashing" function $f(\vec{x})$ (applying a threshold) computed over a weighted sum of inputs. Moreover, the weighted sum can be computed as the dot product between a vector of neural weights $\vec{w}$ and the input vector for a specific training example $\vec{x}^{(i)}$:

$$z = \vec{w}^T \cdot \vec{x}^{(i)} \tag{2.1}$$

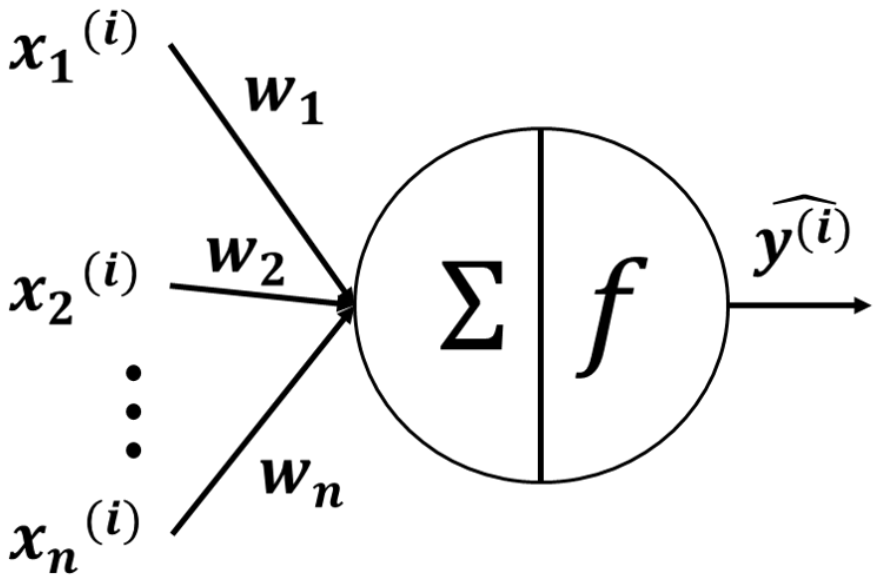

Fig. 2.1. Representation of the perceptron that computes the weighted sum of the input vector components of the i^{th} training example, to make a prediction $\hat{y}^{(i)} \in [0, 1]$. [38].

A simple representation of the perceptron is described in 2.1 where f represents the activation function (threshold computation) and $\sum$ refers to the weighted sum of the inputs $x_1^{(i)}, x_2^{(i)}, ..., x_n^{(i)}$ contained in $\vec{x}^{(i)}$.

One problem with this approach is the inability to compute non-linearly separable functions, such as the XOR boolean operator. As stated by Minsky, computing non-linear functions would require extra layers of several perceptrons, also defined as a hidden layers [39]. Furthermore, the learning rule provided by Rosenblatt do not work because there is no way to adjust weights in the previous hidden layers. In other words, the update rule provides information for the error made only by the last perceptron.

Several years passed until Geoffrey Hinton *et al.* provided an efficient and automatic way for updating all weights in previous layers: the backpropagation algorithm [40]. The learning procedure was defined using the chain rule from calculus, where error derivatives would be computed for each weight in the multi-layer perceptron. In consequence, all weights would be updated based

on how much each one of them contributed to the final prediction error. To make this learning procedure feasible, the activation f was changed for a sigmoid function described in (2.2), which is both non-linear and differentiable. In general, any function f that has a bounded derivative will work using this learning regime [40, 41]. It is important to note that f corresponds to the prediction $\hat{y}^{(i)}$ made by the multi-layer perceptron.

$$f = \frac{1}{1 + \exp^{-z}} \tag{2.2}$$

Therefore, the main objective for training a multi-layer perceptron consists of finding the best combination of neural weights that minimizes the error between all observed values $y^{(i)}$ and predictions $\hat{y}^{(i)}$. Depending on how the error is calculated, the objective can be stated as a regression or classification problem. If the error term is calculated using the averaged squared error over the m training examples as expressed in (2.3), then it is a regression problem. On the other hand, if E is computed using the average log-losses over all training examples as defined in (2.4) it is expressed as a classification task. The expression defined by E is also denoted as the cost function in the machine learning jargon.

$$E = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2 \tag{2.3}$$

$$E = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}log(\hat{y}^{(i)}))) + (1 - y^{(i)})log((1 - \hat{y}^{(i)})) \tag{2.4}$$

To update the neural weights, the chain rule of calculus is needed to compute the partial derivatives of the cost function E with respect to each of the weights w_j in the network. Because the objective is to minimize E, an iterative optimization technique defined as Gradient Descent (GD) is applied to update weights after computing the error over all training examples. This update rule is defined in (2.5), where α defines the learning rate (step size) and w_t defines the value of a specific weight in iteration t.

$$w_{t+1} := w_t - \alpha\frac{\partial E}{\partial w_j} \tag{2.5}$$

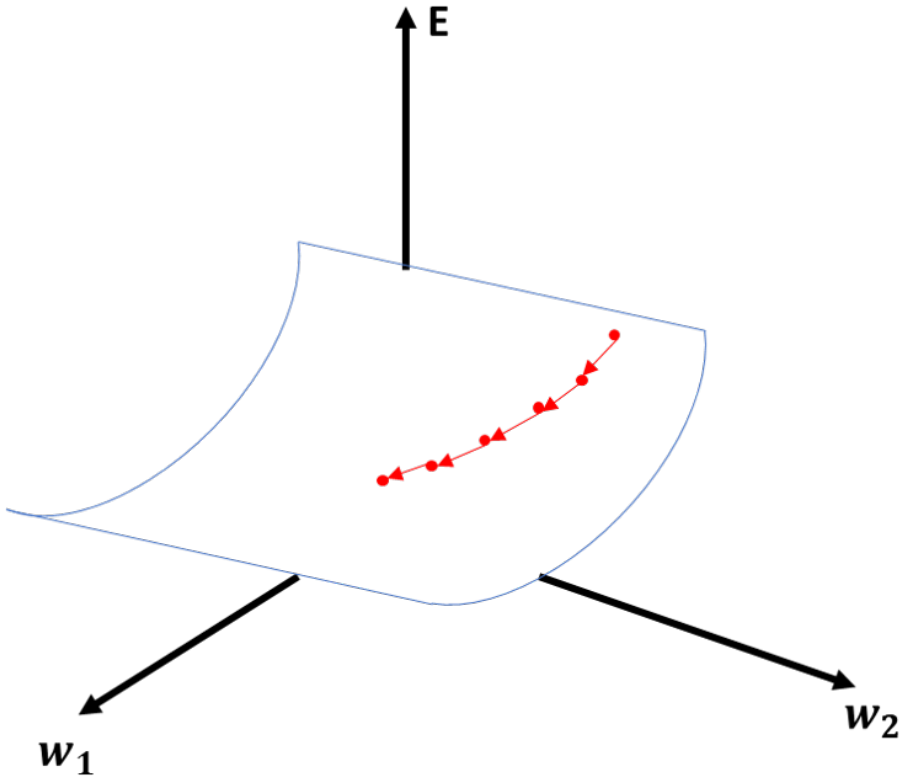

Fig. 2.2. Illustrative example of Gradient Descent optimization to find the minimum of the error surface E that is parameterized by two weights w_1 and w_2.

From a geometric perspective, at every iteration GD is calculating the steepest negative direction of the n dimensional error surface from a point in space, which corresponds to the current combination of weights. Therefore, after every iteration, the algorithm tries to converge to a minimum of the error surface E. An example of this learning procedure is illustrated in 2.2, where the error surface E is parameterized by only two weights w_1 and w_2. The red line describes the trajectory of both weight values as they are updated with each iteration of Gradient Descent.

With this approach, multi-layer perceptrons started to be trained efficiently. Later on, in 1989, Kurt Hornik would prove mathematically that neural networks (NN) could approximate theoretically any function given enough time and computational power [41].

2.2.2 Training issues of neural networks

Even with the beautiful mathematical proof stating NN as universal approximators, Yoshua Bengio *et al.* encountered a problem when training large neural networks: gradients that propagated throughout the network would vanish or explode [42]. Several solutions were proposed to overcome the difficulties of training deep neural networks (DNN): unsupervised training for initializing weights at each layer, weight sharing using different network architectures such as convolutional neural networks (CNN), and using graphics processing units (GPUs) [43–45].

A thorough analysis was conducted by Yoshua Bengio and Xavier Glorot to identify the main causes that resulted in poor performance of the backpropagation algorithm [46]. Two principal findings were reported:

1. The non-linear sigmoid function was a poor choice because it presents saturation during training. This resulted in slow learning, which was reflected on several plateaus of the error surface.

2. Choosing random weights, without taking into account to which layers they correspond, is a bad decision. As a consequence, subsequent multiplication of derivatives resulted in the exploding/vanishing gradient problem (non constant variance). The solution consisted in random weight initialization with scaling, which resulted in faster training convergence.

As presented in the original paper, normalized initialization, or commonly known as "Xavier initialization", was computed to initialize weights considering the specific layer within the network and performing scaling accordingly. Equation (2.6) defines initialization from a uniform distribution over a symmetric interval considering the number of incoming connections n_j and outgoing connections n_{j+1} of a specific layer within the network.

$$W \sim U[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}] \tag{2.6}$$

To address the problem of the sigmoid activation function, several efforts were done to search for a better non-linear activation [47, 48]. Results showed that the best f is the simple rectified

linear activation function (ReL), which is expressed in equation (2.7). Actually, any neuron that applies the ReL activation is defined as a Rectified Linear Unit (ReLU).

$$f(x) = max(0, x) \tag{2.7}$$

Although the ReL activation function presents a discontinuity at $x = 0$, it has proven to have several advantages over the logistic sigmoid activation [49]:

1. One main advantage is that the ReL function can represent sparse representations because there is no need for all neurons to be activated simultaneously (many units could output zero). Thanks to this property computation is faster because neurons that are not activated can be ignored.

2. It presents a solution to the vanishing gradient problem because the derivative is either 0 or 1. Therefore, backpropagation runs without getting stuck in plateaus.

3. Also, more distributed representations are obtained because it does not focus on single neurons, but rather in a multiple combinations of them.

Besides all these ingenious solutions to tackle the vanishing/exploding gradient issue, there is another problem that has been present in any machine learning model: overfitting. A phenomenon that has been defined as: *"having more parameters in the model that can be justified by the data"* [50]. Several techniques have been implemented to reduce its effects during training, and improve model generalization. For example, Geoffrey Hinton *et al.* implemented a very basic, but powerful idea: the dropout technique. It consists on "removing" temporarily some random amount of neurons during training, thus avoiding complex and unnecessary neural adaptation. Therefore, only the most important features are learned by each neuron while activated during the optimization process [51].

A more classical approach in machine learning consists in adding a penalty term to the error function E, a method known as regularization. There are two main types of regularization [52]:

1. L1 (lasso) regularization: it consists on adding the sum of the absolute values of the weights in the NN to the cost function E. Expression (2.8) denotes a classical L1 regression problem.

2. L2 (Ridge or Tikhonov) regularization: also called weight decay, it consists on adding the sum of the squared values of the weights in the NN to the cost function E. Expression (2.9) denotes a classical L2 regression problem.

Both types of equations have a regularization parameter $\lambda > 0$ that controls the amount of penalty that is added to the cost function E and is scaled by the amount of training examples m with the factor $\frac{1}{2m}$.

$$E = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2 + \frac{\lambda}{2m}\sum_{w}|w_j| \tag{2.8}$$

$$E = \frac{1}{m}\sum_{i=1}^{m}(y^{(i)} - \hat{y}^{(i)})^2 + \frac{\lambda}{2m}\sum_{w}w_j^2 \tag{2.9}$$

Even when overfitting has been reduced considerably, there is another issue that is present: long convergence times to an optimal solution during training. Changes in the probability distributions of activations computed in each layer during training have been reported as one of the main causes for this slow down; a problem that has been defined as internal covariate shift [25]. A solution proposed by Christian Szegedy and Sergey Ioffe was to normalize all layer's inputs x_i to adjust the distributions, such that even if the network parameters change during training, the activations will maintain the same distributions [25]. To achieve this goal, equations (2.10) and (2.11) corresponding to the mini batch mean and variance respectively, are computed.

$$\mu_B = \frac{1}{n}\sum_{i=1}^{n}x_i \tag{2.10}$$

$$\sigma_B^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_B)^2 \tag{2.11}$$

Normalization is performed on single scalar features producing a distribution with mean 0 and variance 1. Therefore, using the mean and variance of equations (2.10) and (2.11), normalization of a mini batch is calculated with expression (2.12). The variable ϵ is added to avoid singularities during computation.

$$\hat{x}_i = \frac{(x_i - \mu_B)^2}{\sqrt{\sigma_B^2 + \epsilon}} \tag{2.12}$$

One important thing, as reported by the authors, is that layer representations may change because of these transformations. To avoid this, scale and shift computations are introduced to recover the layer representation power. Equation (2.13) defines both trainable parameters γ and β to recover such representations. This step ensures that the transformation performed over the mini batches are able to represent the identity transform.

$$y_i = \gamma \hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \tag{2.13}$$

2.2.3 Optimization algorithms for neural network training

To speed up training times, efforts have focused on making adjustments to the learning rule defined in equation (2.5). A classical variant of this algorithm is Stochastic Gradient Descent (SGD), which computes the gradient of E for single training examples instead of all the data contained in the training set, as specified in (2.14). Also, computation of the gradient of E has been defined for batches of training data, a variant called Batch Gradient Descent (BGD) [53].

$$w_{t+1} := w_t - \alpha \frac{\partial E(x^{(i)}, y^{(i)})}{\partial w_j} \tag{2.14}$$

A faster optimization algorithm is GD with momentum [53]. The basic idea consists on calculating an exponential weighted average of the past gradients and adding it to the update rule. This helps to damp the oscillations generated in SGD and BGD, thus creating faster update steps in the learning procedure and achieving lower convergence times. Both calculations, for the exponential weighted average and the final update rule, are depicted in equations (2.15) and (2.16), respectively. In equation (2.15), the hyper parameter $\gamma \in [0, 1]$ adjusts the amount of weight assigned to previous gradient results, while v_{t-1} and v_t correspond to the gradients computed in the last and current iterations.

$$v_t = \gamma v_{t-1} + \nabla_w E \tag{2.15}$$

$$w_{t+1} := w_t - \alpha v_t \tag{2.16}$$

Another optimizer that has shown promising results is the Adaptive Moment Estimation (Adam) learning algorithm [54]. This approach builds upon GD with momentum and the RMSprop algorithms by computing the exponential weighted averages of past gradients and their squared magnitudes; followed by their corresponding bias corrections. Equations (2.17) and (2.18) define the calculations needed to compute the exponential weighted averages for both, the gradients and their squared magnitudes. Bias corrections are implemented in expressions (2.19) and (2.20). Finally, the learning update rule defined by Adam, using both bias corrections, is written in equation (2.21).

$$v_t = (\beta_1)v_{t-1} + (1 - \beta_1)\nabla_w E \tag{2.17}$$

$$m_t = (\beta_2)m_{t-1} + (1 - \beta_2)(\nabla_w E)^2 \tag{2.18}$$

$$\hat{v_t} = \frac{v_t}{1 - \beta_1^t} \tag{2.19}$$

$$\hat{m_t} = \frac{m_t}{1 - \beta_2^t} \tag{2.20}$$

$$w_{t+1} := w_t - \alpha \frac{\hat{v_t}}{\sqrt{\hat{m_t}} + \epsilon} \tag{2.21}$$

Other optimizers that have been developed throughout the history of NN include RMSprop, Adadelta, Adagrad, Nadam, and AdaMax [53–55].

2.2.4 Hyper parameter optimization

It is important to highlight the large number of hyper parameters that need to be tuned in a DNN to achieve acceptable results. Actually, finding the best set of hyper parameters for a DNN is a difficult task to do by hand because the search space is really large. As exposed by James Bergstra *et al.*,

"hyper parameter optimization is the problem of optimizing a loss function over a graph-structured configuration space" [56].

There have been efforts to create algorithms that search efficiently for the best set of hyper parameter values. One typical approach is grid search, in which all possible combination of values of the hyper parameters are explored in a hyper dimensional "grid". The problem with this strategy comes inevitably from the *curse of dimensionality*, in which the search space size increases exponentially as the number of hyper parameters is increased [57].

Another solution for hyper parameter tuning is random search, which has proven to have better results compared to grid search [56, 57]. In this technique, all values for each hyper parameter are independently sampled from a uniform density from the same space, as the one spanned by regular grid search.

Both techniques are really powerful when the evaluation of the learning algorithm function f is cheap. In other words, when the amount of resources and time required to learn f are low. For DNNs that is not the case, because a lot of time and resources are needed to train them. In such cases, when learning the function $f : X \rightarrow Y$ is really expensive, Sequential Model-Based Global Optimization (SMBO) algorithms are the preferred choice to make the hyper parameter tuning [56]. SMBO models try to approximate f with a surrogate that is cheaper to evaluate. Two famous SMBO algorithms are the hierarchical Gaussian Process and the tree-structured Parzen estimator (TPE). Both of them restrict configuration spaces to tree structures [56].

2.2.5 Convolutional neural networks

Until now, the discussion has centered in DNN, but another topic that is of great interest for this dissertation are Convolutional Neural Networks (CNN). As reported in the literature, the first CNN was developed by Kunihiko Fukushima and was named: the *Neocognitron* [58, 59]. A more recent CNN, which resembles most modern architectures and incorporated the backpropagation algorithm,

was implemented by Yann LeCun *et al.* [60].

As their name suggests, these networks substitute the multiplication operator for the convolution computation in equation (2.1). In a formal mathematical sense, this linear operation defines an integral that expresses the amount of overlap of one function g as it is shifted over another function f. Its identification symbol is the asterisk $*$ and is explicitly stated in expression (2.22) [61].

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.22}$$

In the deep learning jargon, function f is designated as the *input*, function g as the *kernel*, and the output function as the *feature map* [62]. This definition applies for continuous valued functions that are usually stated in time-domain problems. For the case of CNNs, the discretized convolution operation is implemented over several N dimensions as stated in equation (2.23).

$$(f * g)(t_1, ..., t_i, ..., t_N) = \sum_{\tau_1} ... \sum_{\tau_i} ... \sum_{\tau_N} f(\tau_1, ...\tau_i, ..., \tau_N)g(t_1 - \tau_1, ..., t_i - \tau_i, ..., t_N - \tau_N) \tag{2.23}$$

Because CNNs are frequently implemented for pattern recognition in images, the input is usually expressed as I and the kernel as K. In practice, when dealing with spatial domain applications, the *cross-correlation* operation is used instead, which is the same computation as convolution but without flipping the kernel as defined in (2.24). Therefore, this operation is computing the similarity between both signals, the kernel and a specific portion of the image, as the kernel is superimposed over all the image domain. Similar to DNNs, the objective of convolutional neural networks consists in learning the parameters within all kernels K_j, that are convolved with the input I.

$$(I * K)(t_1, ..., t_i, ..., t_N) = \sum_{\tau_1} ... \sum_{\tau_i} ... \sum_{\tau_N} I(t_1 + \tau_1, ..., t_i + \tau_i, ..., t_N + \tau_N)K(\tau_1, ...\tau_i, ..., \tau_N) \tag{2.24}$$

A very famous application of the convolution operation is the Sobel operator for edge detection [63]. This specific algorithm implements 3x3 kernels that are convolved with images to

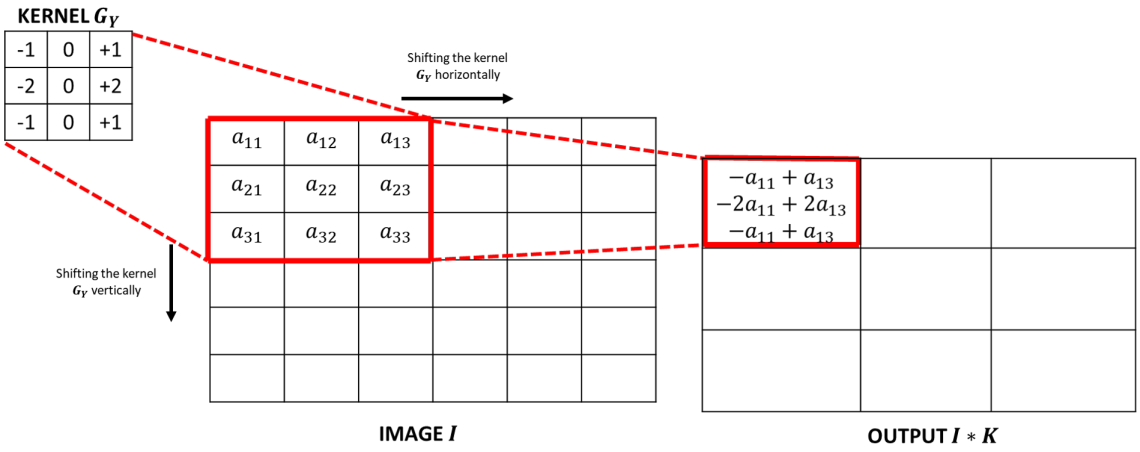

Fig. 2.3. Example of the convolution operation using the Sobel operator as kernel.

calculate derivative approximations in the horizontal and vertical directions. Figure 2.3 illustrates the convolution operation implemented using the Sobel operator to detect horizontal edges using a kernel G_y, superimposed over a two-dimensional image I. In this case, a 3x3 kernel G_y is convolved with a 6x6 image I to produce a 3x3 matrix as a result. Only the first convolution computation is shown.

The motivation for using CNNs with respect to classical DNNs resides on three main advantages, sparse interactions, parameter sharing, and equivariant representations [62]. Sparse interactions arise from convolving kernels that are much smaller in size than the input I, which derives into fewer parameters associated to each output. On the other hand, DNNs have all input neurons connected to all output units. As a result, DNNs contain a large number of parameters compared to CNNs. Therefore, thanks to sparse interactions, CNNs can build complex structures based on a smaller amount of parameters.

The second advantage of CNNs, parameter sharing, means that a single parameter may be used more than once in the CNN. The reason behind this principle lies in shifting the kernel over the input domain, which forces the application of the same kernel weights (parameters) to different spatial locations within the input I.

Finally, equivariant translation, results naturally from the parameter sharing property. Even if the input I is translated, the representations that are built by the CNN will stay the same. In order words, even if an object is shifted within the image, the CNN will still have the ability to detect it. Thus, these three properties of CNNs provide a powerful framework for object detection and define a clear advantage over classic DNNs.

Modern CNN layers have a different structure compared to classic DNNs. Each layer in a CNN is composed by three different transformations [62, 64]:

1. Convolutional step: a set of m kernels K_i with equivalent sizes (kxk) are convolved with the n dimensional input I in parallel, giving as a result different tensors ρ_i (with the bias term b_i included) as expressed in equation (2.25). Finally, the different results ρ_i are concatenated in a multi-dimensional array or volume ρ.

$$\rho_i = I * K_i + b_i \tag{2.25}$$

2. Detection step: a non-linear function f, such as the ReL activation defined in equation (2.7), is applied to each component of the concatenated output of convolutions ρ, to give as a result, a volume γ with components γ_i; as described in (2.26).

$$\gamma_i = f(I * K_i + b_i) \tag{2.26}$$

3. Pooling step: it is defined as a function that replaces the output γ_i of the net, at a certain location, with a summary statistic of the nearby outputs δ_i [62]. By adding this step, the CNN keeps the most important abstractions and discards the irrelevant ones (invariance feature extraction), thus reducing computational resources and the number of parameters that need to be trained. There are different types of pooling operations applied to the nearby outputs:

   (a) Max pooling: calculates the maximum output of a rectangular neighborhood.

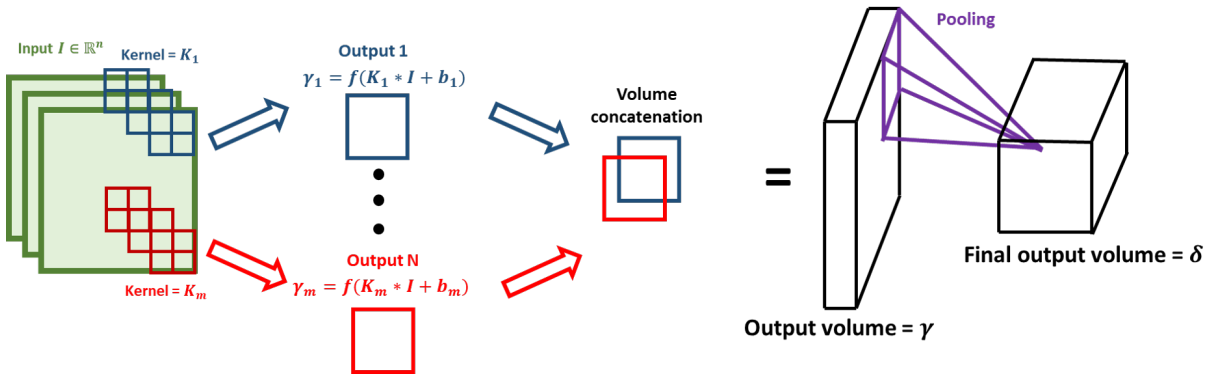

Fig. 2.4. Example of a convolutional layer where m kernels are applied to an n dimensional input I. The resulting volumes ρ_i from the convolutional steps are omitted.

(b) Average pooling: computes the average value of a rectangular neighborhood.

(c) L^2 pooling: computes the L^2 norm of a rectangular neighborhood.

(d) Global weighted average pooling: computes a weighted average based on the distance from the central pixel.

An illustrative example of a CNN layer is depicted in Figure 2.4, where each kernel K_i could be seen as the analogous of a neuron in a classical DNN. In this type of layer, m kernels are applied to an n dimensional input I, resulting in ρ_i tensors. Then, a non-linear function f is applied to each convolution result ρ_i resulting in a tensor γ_i. All tensors γ_i are concatenated to produce a final volume γ. Finally, a pooling layer computes the summary statistics at each output neighborhood resulting in the output tensor δ of the layer.

It is important to see that the input volume is reduced after being processed by each CNN layer. Because of this shrinkage effect, some padding operations are also computed in the layers, thus maintaining the volume size and allowing the CNN to generate more powerful abstractions.

Although this is the typical architecture of a CNN layer, more operations can be defined within each layer, such as normalization layers, loss layers, or even fully connected (FC) layers (typical layers of a DNN) [64]. By using the pooling and convolution operations, the CNN becomes a

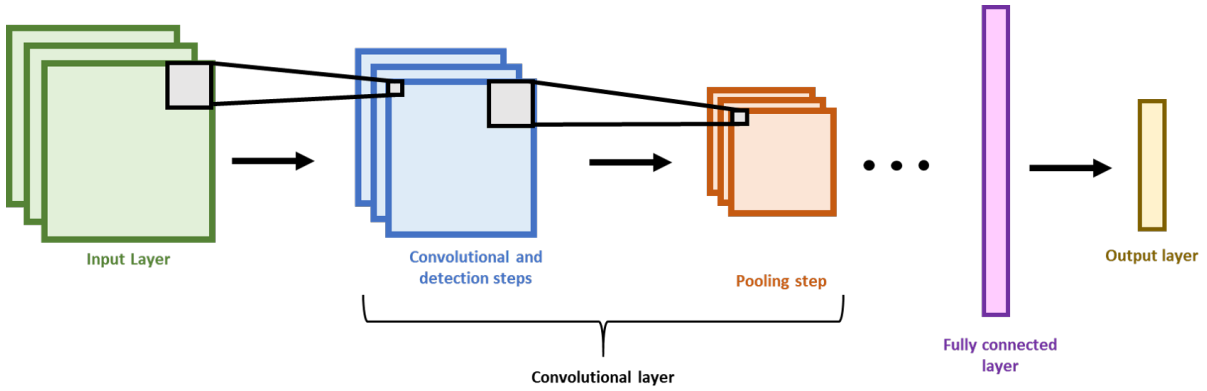

Fig. 2.5. Model of a classic Convolutional Neural Network architecture.

powerful algorithm that can learn features that are invariant not only to translations, but also to rotations and more complex spatial transformations [62].

An important aspect of CNNs is that the learning procedure is based on the same backpropagation algorithm used by classic DNNs [40]. Therefore, all the techniques described before, to improve and solve different training issues of classic DNNs, can also be applied to CNNs. A typical CNN architecture is shown in Figure 2.5, although variants of this network configuration have also been investigated [24–26, 65, 66]. The classical architecture comprises four main portions, 1) the input layer, 2) a set of convolutional layers, where a typical single layer is illustrated in Figure 2.4, 3) a fully connected layer, and 4) an output layer.

2.2.6 Deep learning and medical imaging

Thanks to the ability of DNNs and CNNs to learn complex patterns within data, they have become the preferred choice to solve image recognition tasks. Many researchers have focused on developing medical tools based on these algorithms to assist in *"image registration, anatomy localization, lesion segmentation, detection of objects and cells, tissue segmentation, and computer-aided detection"* [23]. Specifically, computer-aided diagnosis (CAD) is the topic of main relevance for this dissertation.

In the literature, the typical workflow for designing a CAD system is defined in two steps, 1) candidate generation algorithms and 2) classifiers that output the most probable regions having the disease. As reported by Dinggang Shen *et al.*, CNNs have become the algorithm of choice for developing any of the two aforementioned steps [23]. As it has been stated, the motivation of using deep learning techniques for detecting abnormalities in medical images resides in reducing error rates during evaluation; mainly the number of false positives (FP) and false negatives (FN) [6–8, 23].

Although DNNs and CNNs have reported promising results for detecting abnormalities in medical images, its implementation has become a major challenge due to the limited amount of medical data available for training. Even so, many strategies to prevent overfitting have been developed such as data augmentation, transfer learning, and using patches of samples instead of full-sized images [23, 67–70]. It is evident that more efforts are required to automate the generation of available medical data to improve the performance of deep learning CAD systems. A problem that depends on the national context of each country and deals with other issues, such as legal and administration matters.

2.3 Literature review

Several CAD tools have been developed to reduce false positive rates and assist radiologists in the evaluation of lung cancer in CT. Most CAD systems implement any of the following four steps: 1) preprocessing of CT, 2) lung segmentation, 3) lung nodule detection, and 4) lung nodule malignancy classification [70, 71]. Recent studies have focused mainly on either lung nodule detection or malignancy classification [28, 68, 70–80]. Table 2.2 gives a brief summary of all related research.

2.3.1 Nodule detection algorithms

Generally, for nodule detection algorithms there are two main steps: 1) candidate selection (CS) and 2) false positive reduction (FPR) [72, 79]. A thorough analysis has shown that Convolutional Neural Networks (CNN) outperform any other type of classifiers in the FP reduction and nodule candidate

Table 2.2. Research studies focused on both stages of the nodule assessment process: nodular detection and malignancy classification.

Authors	Year	CT	EVALUATION METRICS						IMPLEMENTED METHODS				
			ACC	SNS	SPC	F1	ROC	Log Loss	Nodule detector	Malignancy classifier	Lung cancer estimator with combined nodular characteristics	Direct lung cancer estimator	Optimization
Kuruvilla, J. et al.	2013	155	0.91	-	-	-	-	-	-	-	-	• **Architecture:** NN with variable α, β. • **Data:** geometric parameters of 2D lungs	**GD con ADAM**
			0.93	0.91	1	-	-	-	-	-	-	• **Architecture:** NN with variable β. • **Data:** geometric parameters of 2D lungs	**GD with modified ADAM**
Kumar, D. et al.	2015	157	0.75	0.83	-	-	-	-	• **Architecture:** autoencoder • **Data:** 2D nodules.	• **Architecture:** Binary Decision Tree. • **Data:** vector with 200 characteristics.	-	-	**L-BFGS**
Madero, O. et al.	2015	151	0.82	0.91	0.74	-	0.81	-	-	• **Architecture:** SVM with radial basis function. • **Data:** vector with 11 characteristics obtained with wavelets..	-	-	**Mult. Lagrange**
Dou, Q. et al.	2016	888	-	0.92 with 8 FP/scan	-	-	-	-	• **Architecture:** three 3D CNN with different receptive fields. • **Data:** 3D nodules.	-	-	-	**SGD with momentum**
Kuan, K. et al.	2017	2702	-	0.667	0.77	0.598	-	0.53	• **Architecture:** 3D CNN – ResNet. • **Data:** 3D CT volume portion.	• **Architecture:** two 3D CNN - ResNet in parallel with variable α. • **Data 1° network:** 3D CT volume portion. • **Data 2° network:** nodules detected by nodule detector.	• **Architecture:** NN with 2 layers. • **Data:** weighted characteristics obtained from previous networks.	-	**GD with ADAM**
Hammack, D. et al.	2017	2500	-	-	-	-	0.85	0.40	• **Architecture:** Ensemble learning with several 3D CNN – ResNets / VGGNets • **Data:** 3D CT volume portion.	• **Architecture:** Ensemble learning with several 3D CNN – ResNets / VGGNets • **Data:** : nodules detected by nodule detector.	• **Architecture:** Ensemble learning with 3D CNN – ResNets. Final classification was done with a Binary Decision Tree. • **Data:** nodules detected by nodule detector.	• **Note:** 17 CNN were implemented for all stages in the algorithm.	**GD with ADAM and Nesterov momentum**
Setio, A. et al	2017	888	-	0.95 and < 1 FP/scan	-	-	-	-	• **Architecture:** Ensemble learning with 5 different 3D CNN. • **Data:** varies according to architecture.	-	-	-	**Multiple**
Liu. et al.	2017	2608 nódulos	-	-	-	-	0.780	-	-	• **Arquitectura:** Ensemble learning with 2D and 3D CNN. • **Data:** nodules with malignancy pathologic confirmation.	-	-	**SGD con momento**
Ding, J. et al.	2017	888	-	0.922 with 1 FP/scan	-	-	-	-	• **Architecture:** Faster Region-based CNN.	-	-	-	-
Ross, G. et al.	2018	888	-	0.8929 with 1.789 FP/scan	-	-	-	-	• **Architecture:** 3D Inception-ResNet.	-	-	-	**GD with ADAM**
Dai, C. et al.	2018	1000	-	0.91 with 32 FP/scan	-	-	-	-	• **Architecture:** 3D Inception-ResNet.	-	-	-	**GD with ADAM**
Shen, S. et al.	2018	897	0.842	0.705	0.89	-	0.856	-	-	• **Architecture:** Hierarchical Semantic CNN with 2 subnetworks: 1° to estimate morphological characteristics and 2° for malignancy classification.	-	-	**GD with ADAM**
Ardila, D. et al.	2019	42,920	-	-	-	-	0.944	-	• **Architecture:** 3D RetinaNet without feature pyramid network.	-	• **Architecture:** CNN that combines outputs from nodule ROI detector and direct lung cancer estimator.	• **Architecture:** 3D inflated Inception V1 Network.	**GD with Focal Loss**

ABBREVIATIONS

- α = *learning rate.*
- **ACC** = *accuracy.*
- **ADAM** = *adaptive moment estimation.*
- β = *learning momentum.*
- **CNN** = *convolutional neural network.*
- **FP** = *false positives.*
- **F1** = *F1 score.*
- **GD** = *gradient descent.*
- **LBFGS** = *BFGS algorithm with limited memory.*
- **Log Loss** = *logarithmic los.*
- **Mult** = *multipliers.*
- **NN** = *neural network.*
- **ResNet** = *residual network.*
- **ROC** = *receiver operating characteristic.*
- **SGD** = *stochastic gradient descent.*
- **SNS** = *sensitivity.*
- **SPC** = *specificity.*
- **SVM** = *support vector machine..*
- **CT** = *computerized tomography.*

selection tasks [72, 79]. An example is illustrated in the LUNA 16 challenge, where more than 95% detection sensitivity was achieved with less than 1 FP/scan using the results of 5 different CNNs and 888 CTs [72]. This research has proved to be the gold standard for the nodule detection task.

Isotropic re-sampling of the CT is a common task to do before nodule detection. Revised literature has defined voxel cubes of $1mm^3$ as a standard choice for uniform volumes [68, 81]. Also, lung segmentation is performed to reduce computational expense and focus only on the desired region of interest, thus decreasing analysis time [28, 68, 72, 81].

One research used all of the aforementioned steps: candidate selection, false positive reduction, isotropic re-sampling, and lung segmentation [81]. For candidate generation they used a 3D U-Net-inspired DNN architecture with a sliding window. Centroids from labeled volumes were computed for generating candidate nodules. With respect to false positive reduction, they used nodule volumes as the main feature and a size threshold of $8mm^3$, where any candidate with a lower size was treated as a false positive. An adapted Inception-ResNet architecture [82] was implemented using scaled exponential linear units. The training was performed using a 10 fold cross-validation, where all nodules generated from validation bins were used for the FPR task. Adam optimizer was implemented for optimizing the objective function. The results defined an averaged validation sensitivity of 89.29% with 1.789 FP/scan.

Another study developed a similar approach also using the 3D Inception-ResNet architecture [69]. A data set of 1000 CTs, from the Tianchi Medical Competition, was used for training, validation, and testing, where inputs to the network comprised normalized 64 x 64 x 64 CT patches. Also, both, candidate generation and FP reduction, were performed on the data set. An area under the ROC curve of 78% was obtained as the final result.

For the false positive reduction task, Dou Q. *et al.* developed three typical 3D CNN with different contextual information schemes and several max-pooling layers between convolutional ones [79]. Three receptive fields were proposed: (20x20x6), (30x30x10), and (40x40x26) to capture different

surrounding spatial patterns. The training was performed using Stochastic Gradient Descent and the LUNA 16 Challenge Database comprised by 888 CTs [72]. The results indicated a validation sensitivity of 92.2% with 8 FP/scan.

Finally, Ding, J. *et al.* focused on both, the candidate detection challenge and the false positive reduction task [83]. A Faster Region-based CNN was developed for the candidate detection stage and a classical 3D CNN was designed for the FP reduction step. All the evaluation was performed using the LUNA 16 Challenge Database. A sensitivity of 94.6% with 15 FP/scan was obtained for the candidate detection task. For the FP reduction task, an average FROC-Score of 0.893 was obtained; therefore, attaining the best performance in the LUNA 16 Challenge.

2.3.2 Malignancy classification algorithms

With respect to cancer estimation, a study obtained 93.3% accuracy, 100% specificity and 91% sensitivity using an Artificial Neural Network (ANN) trained with Adaptive Moment Estimation (Adam) and 155 CTs to determine the presence of cancer in a specific CT slice [77]. Another study focused in lung nodule malignancy classification achieving 75.01% accuracy, 83.35% sensitivity and 0.39 FP/scan using a five-layered auto-encoder trained by L-BFGS and 4,323 nodules [78]. To improve malignancy classification of nodules, Shen *et al.* implemented a CNN trained with Adam stochastic optimization algorithm and 897 LDCT. The CNN architecture consisted of two level outputs: 1) low-level radiologist semantic features and 2) a high-level malignancy classification. Results comprised 85.6% Area Under the Curve (AUC), 70.5% mean sensitivity, 88.9% mean specificity, and 84.2% mean accuracy [70]. Also, a research conducted by Madero *et al.* used 151 CTs and applied three wavelet transforms db1, db2 and db3 to extract and select 11 characteristics. A Support Vector Machine (SVM) was developed to perform malignancy classification resulting in 82% accuracy, 90.90% sensitivity and 73.91% specificity [71].

One of the main issues with malignancy classification is the lack of an open source data base with sufficient histopathologic labels. Therefore, few studies have reported results based on tissue

confirmation and have focused only in radiologic features. One of these studies trained an ensemble of 2D and 3D CNNs with 2608 nodules and their respective pathologic confirmed labels, thus obtaining an area under the ROC curve of 0.78 [29].

Finally, based on all revised studies, the best results for malignancy classification have been reported by researchers at Google [28]. They used 42,920 CTs from 14,851 patients contained in the NLST study. Using a similar framework to the one proposed in this project, they implemented a two block modular approach. The first block consisted of a Region of Interest (ROI) detection model, where the selected CNN architecture was RetinaNet. All the detected ROIs were obtained either from one or two CTs corresponding to the same patient submitted for follow-up examination. The second block used an end-to-end inflated Inception V1 network, trained with $1.5mm^3$ voxel size volumes to predict cancer within 1 year. Outputs from both blocks were used as inputs to train a single CNN.

There are two major differences in this study with respect to the others. First, they used pathology-confirmed cancer labels as opposed to radiologic assessment scores. Second, they implemented comparison techniques between prior and current CT, to improve malignancy assessment. The final results comprised a 94.4% AUC calculated on a test set containing 6,716 CTs, where predictions were thresholded at three different levels to match the evaluation metrics specified in Lung-RADS [15]. Up until the writing of this thesis, the research published by Google poses the new gold standard for an end-to-end lung cancer risk assessment on CT.

CHAPTER 3

Methods

3.1 Study design

3.1.1 Research population

Population characteristics for this study were defined for the algorithm to be implemented in any Mexican hospital and clinical context. Thus, the following inclusion criteria was specified:

1. Gender: either.
2. Age: between 18 and 95 years.
3. Ethnics: either.
4. With or without smoking history.

More precisely, two types of patients were considered in this research as specified in 1.6. Only nodules $\geq$ 3 mm in diameter were considered, as any smaller lesion is defined as irrelevant based on the Guidelines for Management of Incidentally Detected Pulmonary Nodules in Adults [18].

Determining the sample size for a machine learning algorithm, for both training and validation, is a difficult task. Actually, it is still a major area of research in Computer Science. However, there has been evidence that with more training data CNNs can achieve better performance [84]. Because data is limited in medical applications, there is a major challenge to address when developing deep learning models for CAD systems [23]. Therefore, based on data acquisition limitations rather than sample size calculation; we specified a range with a lower bound of 1000 CT for both training and validation, with no upper bound limit for the required number of CTs. If available in records, histologic studies were also included for cancer label confirmation.

3.1.2 Type of study

The research study was defined by four distinct characteristics:

1. Observational: there was no direct or indirect impact on the health of patients when doing the computational analysis of pulmonary nodules. No clinical, surgical, or invasive procedures were needed for this study.

2. Retrospective: all analyzed CT scans were taken before this research study was conducted. No specific time interval in the past was specified for collecting CT studies.

3. Cross-sectional: it was decided to store and process the CT scan data in a 5 year time period since the start of this project: 01 September 2019 to 01 September 2024.

4. Descriptive: there was no comparison between CT scans from different patients or the same patient contained in the sample that was collected.

3.1.3 Research duration

The research was developed in two years: from 20 September 2017 to 01 November 2019. Specifically, CT scans and pathological information were acquired from 05 February of 2019 to 25 August of 2019. Changes to the schedule were due to several complications encountered in the research execution.

3.2 Data information sources and manual for data acquisition

3.2.1 Data sources

1. **Data base A:** this source of information comes from the Lung Image Database Consortium Image Collection (LIDC-IDRI) [85]. Seven academic centers and eight medical imaging companies worked together to build this data base comprised by 1010 CT scans and XML files with annotations done by four experienced thoracic radiologists in a two phase annotation process. In the first blinded-read stage, each radiologist evaluated independently each CT scan and classified all lesions into three categories: "nodule $\geq$ 3 mm," "nodule $\leqslant$ 3 mm", and "non-nodule $\geq$ 3 mm". In the second phase, all radiologists reviewed their own annotations along with the marks made by the other three. Apart from evaluating nodule locations, eight subjective radiologic characteristics were assessed in a discretized score scale ranging from 1 to 5: subtlety, internal structure, spiculation, lobulation, sphericity, solidity, margin, and likelihood of malignancy.

2. **Data base B:** all low-dose CT scans contained in this data source come from three hospitals located in the State of Queretaro, Mexico; Hospital Star Médica, Hospital H+, and Hospital Ángeles. Several expert radiologists annotated each CT scan providing the bounding box coordinates for locating nodular lesions. Following a similar approach to the LIDC-IDRI database, each radiologist assessed three subjective radiologic characteristics: likelihood of malignancy, spiculation, and lobulation. Nodular assessment was made using the same discretized score scale ranging from 1 to 5. All annotations were made using the Mango software developed by the Research Imaging Institute UTHSCSA and copyrighted by the University of Texas [86]. Also, Excel files were generated to save the marks made by each expert radiologist. Slice thickness of each CT scan was defined to be $\leqslant$ 2.5 mm; any thickness greater than this value was discarded.

   (a) Data Base B1 (Hospital Star Médica): this data base consists of 73 CT scans annotated by an expert radiologist with 5 years of experience in diagnostic radiology and a nuclear radiologist with 7 years of experience in oncological cases. The scanner model used

was a Siemens SOMATOM Definition with a maximum resolution of 64 slices.

(b) Data Base B2 (Hospital H+): this source comprises 33 CT scans annotated by an expert radiologist with 6 years of experience in interventional radiology and another expert radiologist with 7 years of experience in diagnostic radiology . The scanner model used was a GE Optima CT660 with a maximum resolution of 128 slices.

(c) Data Base B3 (Hospital Ángeles): this data base constitutes 13 CT scans annotated by an expert radiologist with 20 years of experience in diagnostic radiology and another expert radiologist with 1 year of experience in diagnostic radiology and Positron Emission CT. The scanner model used was a GE LightSpeed with a maximum resolution of 64 slices.

3. **Data base C:** this data base comprises three main information sources: pathology confirmation labels to verify the presence or absence of lung/metastatic cancer and the corresponding associated CT scan. All information was obtained from the LIDC-IDRI data base and two hospitals in the State of Queretaro, Mexico; Hospital H+ and Hospital Ángeles.

   (a) Data Base C1 (LIDC-IDRI): this data base consists of 130 CT with confirmed diagnosis at patient level and was defined into three categories: 1) benign/non-malignant disease, 2) malignancy that is a primary lung cancer and 3) a metastatic lesion that is associated with an extra-thoracic primary malignancy (the ”unknown” category was excluded). For the diagnosis method all categories were included.

   (b) Data Base C2 (Hospital H+): this source comprises 8 CT scans from patients with lung/metastatic cancer confirmed with biopsy. The scanner model used was a GE Optima CT660 with a maximum resolution of 128 slices.

Due to the retrospective nature of the study, large amounts of required data and no impact exerted directly or indirectly to the patients it was not mandatory to obtain informed consents from patients. However, in case that the Medical Institution and ethics committee required an informed consent, it was included in Appendix A.

All acquisition protocols were conducted based on the legal guidelines stipulated in *"apartado 5 de la NORMA Oficial Mexicana NOM-004-SSA3-2012, Del expediente clínico y en el apartado 5 de la NORMA Oficial Mexicana NOM-035-SSA3-2012, En materia de información en salud"*.

3.2.2 Data acquisition and safety protocols

All the steps followed in the acquisition protocol are detailed in this, except for data bases A and C1. It is important to express that strict patient confidentiality was mantained during the research.

1. CT acquisition methodology: patient in supine position with elevated arms and maximum inspiration during the acquisition, simple study without intravenous contrast.

2. Anonymization: all patient information that is not relevant to this research was removed and sensible data was anonymized.

3. Data protection: all information was stored in a hard drive and encrypted using a 256 bits Advanced Encryption Standard (AES).

3.3 Programming frameworks

All scripts and programming were developed using the Python programming language version 3.7.3. Several Python libraries were used to implement all algorithms reported in this dissertation and are enlisted below:

1. Tensorflow: an interface for designing machine learning algorithms and an implementation for executing such algorithms [87].

2. Keras: a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed to enable fast experimentation [88].

3. Pydicom: a pure Python package for working with DICOM files such as medical images, reports, and radiotherapy objects [89].

4. Numpy: the fundamental package for scientific computing with Python [90].

5. Scipy: a Python-based ecosystem of open-source software for mathematics, science, and engineering [91].

6. Scikit-learn: a Python package containing simple and efficient tools for data mining and data analysis [92].

7. Pandas: an open source, BSD-licensed library providing high-performance, easy-to-use data structures, and data analysis tools for the Python programming language [93].

8. Matplotlib: is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms [94].

9. Hyperopt: is a Python library for serial and parallel optimization over awkward search spaces, which may include real-valued, discrete, and conditional dimensions [95].

3.4 Preprocessing techniques

Based on several research studies, providing a homogeneous and standardized framework for data analysis is crucial to improve performance of deep learning models applied to medical imaging [68, 72]. An efficient programming framework was implemented to automate the extraction of nodules annotated by expert radiologists. An Object-Oriented Programming (OOP) approach was developed to extract all information from data bases A and B, where nodules were annotated.

Different scripts were written according to the database where the CT scan was located. An important thing to highlight is that these scripts allow the extraction of nodules with any receptive field specified by the user. Thus, the scripts could be used to develop further research in the future and try similar approaches as the ones reported by Dou Q. *et al.* [79]. Appendix B contains all the scripts developed for each data base.

Normalizing the inputs of any DNN or CNN is a crucial task to accelerate training times, an issue that has been discussed with batch normalization in 2.2.6 [25]. Therefore, two main steps were performed, pixel intensity rescaling and spatial re-sampling. Both preprocessing steps have been implemented by several authors across the literature with slight variations [68, 70, 76, 81, 96]. Appendix C contains all the scripts developed for each data base to perform both steps, pixel intensity spacing and spatial re-sampling.

3.4.1 Pixel intensity rescaling

The Hounsfield scale, named after Godfrey N. Hounsfield, has been used to measure radio density of different materials within the human body. These measurements reflect x-ray attenuation, which is proportional to the physical material density [97]. Typical values include -1000 Hounsfield units (HU) for air and +1000 HU for bones. Values of 1828.50 $\pm$ 60.421 HU and 344.45 $\pm$ 20.531 HU have been reported for cortical and cancellous bone respectively [98]. Studies have reported radio density levels of lung tissue depending on its level of inflation. Both limits were specified as 100 HU and -1000 HU for non inflated and overinflated lung tissue [99].

Generally, CT scan pixels are not represented in this scale. However, scanner companies provide the rescaling slope s and intercept b in each DICOM file to convert from pixel intensities to HU by using equation (3.1). This expression computes a linear transformation applied to a 3D pixel value intensity matrix $\boldsymbol{I}_{\boldsymbol{pix}}^{(i)}$, which corresponds to the i^{th} CT scan of a set of m scans comprising a certain database.

$$\boldsymbol{I}_{\boldsymbol{HU}}^{(i)} = s(\boldsymbol{I}_{\boldsymbol{pix}}^{(i)}) + b \tag{3.1}$$

After applying equation (3.1) to all 3D matrices $\boldsymbol{I}_{\boldsymbol{pix}}^{(i)}$, a rescaling operation was performed to define a new range of pixel values that varied from 0 to 1. Based on the radio density values of different structures, air and cancellous bone were defined as the upper and lower bounds of the new range of pixel intensities. Therefore, equation (3.2) was applied to all transformed matrices $\boldsymbol{I}_{\boldsymbol{HU}}^{(i)}$, where $HU_{air} = -1000HU$ and $HU_{bone} = +400HU$ depict the values for both, air and cancellous bone, respectively. A bigger value was chosen to guarantee that all relevant structures, such as lung

tissue, are kept after normalizing.

$$\boldsymbol{I}^{(i)}_{\boldsymbol{rescaled}} = \frac{\boldsymbol{I}^{(i)}_{\boldsymbol{HU}} - HU_{air}}{HU_{bone} - HU_{air}} \tag{3.2}$$

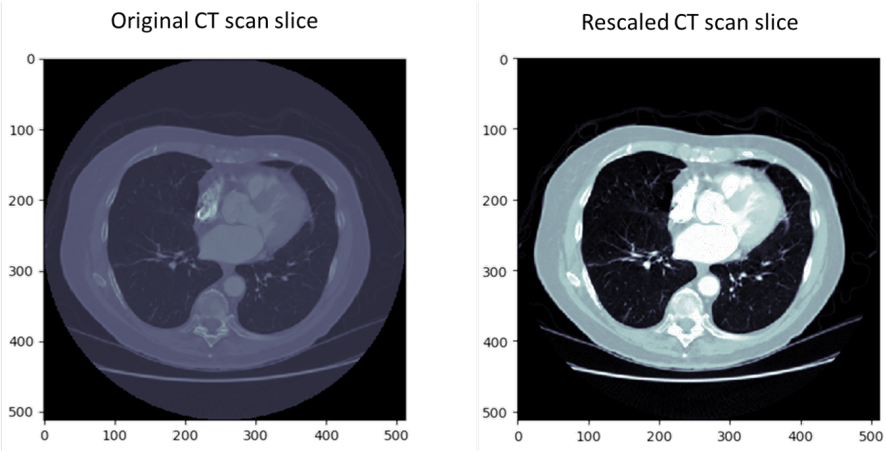

Fig. 3.1. Result of pixel rescaling on a specific CT slice obtained from the LIDC-IDRI database.

Finally, any pixel value > 1 was defined as 1 and any pixel value < 0 was specified as 0. Figure 3.1 illustrates the result of rescaling pixel intensities of the middle CT slice of the first patient of the LIDC-IDRI data base.

3.4.2 Spatial re-sampling

Spatial variations have been observed when analyzing scan resolutions for all CT scans included in the LIDC-IDRI database. Figure 3.2 shows the histograms for all spatial pixel resolutions in each CT scan dimension. Means and standard deviations were calculated for each spatial resolution, which are illustrated in Table 3.1. Furthermore, if the final objective is to implement the algorithm in any clinical context, these spatial resolution differences will be always encountered. Thus, it is important to provide a consistent method for adjusting spatial dimensions to have a standard voxel size, where a voxel is defined as 3-dimensional box of material and is represented by a single pixel value within the pixel matrix that composes the CT scan [22].

A target voxel size of 1 mm^3 was proposed based on implementations developed by other researchers [68, 76, 81, 96]. Therefore, all CT scans were re-sampled to have a voxel size of 1 mm^3,

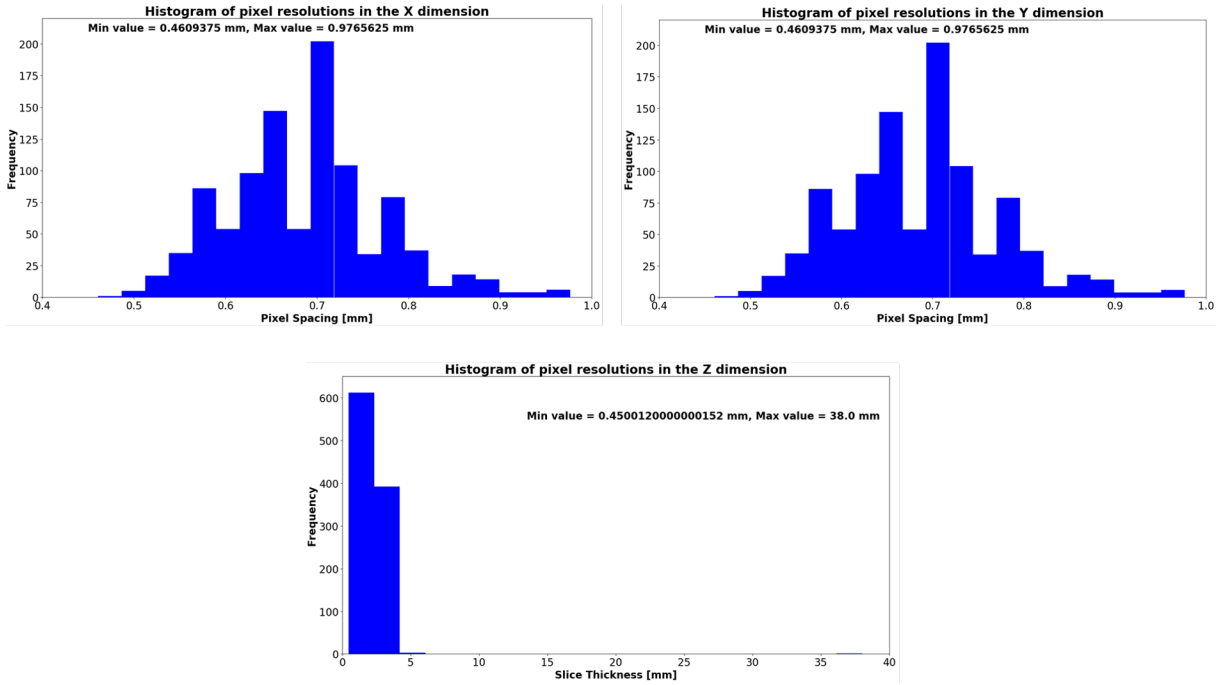

Fig. 3.2. Histograms corresponding to spatial resolutions of each CT scan dimension for a sample size of 1010 patients contained in the LIDC-IDRI database.

Table 3.1. Mean and standard deviation calculations of spatial resolutions corresponding to each CT scan dimension.

Dimension [mm]	Mean [mm]	Standard Deviation [mm]
X	0.687859598	0.084697435
Y	0.687859598	0.084697435
Z	1.765376993	1.413438331

using the nearest-neighbor interpolation algorithm with the Scipy programming framework [91]. The nearest neighbor method simply chooses the nearest pixel value to the desired location in the new interpolated matrix [100]. To implement scan rescaling, a resizing factor r must be calculated for each dimension of the scan $\boldsymbol{I}_{pix}^{(i)}$ based on the desired size of the new re-sampled scan $\boldsymbol{I}_{resampled}^{(i)}$.

Equation (3.3) describes the calculation for obtaining the rescaling factor r_D for any dimension D of a specific CT scan $\boldsymbol{I}_{\boldsymbol{pix}}^{(\boldsymbol{i})}$ by computing the size ratio between the new re-sampled scan size $I_{resampled\ sizeD}^{(i)}$ and the old size $I_{pix\ sizeD}^{(i)}$. Figure 3.3 illustrates the result of re-sampling a single CT scan slice from the LIDC-IDRI database, using the nearest neighbor algorithm, to obtain a voxel volume of 1 mm^3.

$$r_D = \frac{I_{pix\ sizeD}^{(i)}}{I_{resampled\ sizeD}^{(i)}} \tag{3.3}$$

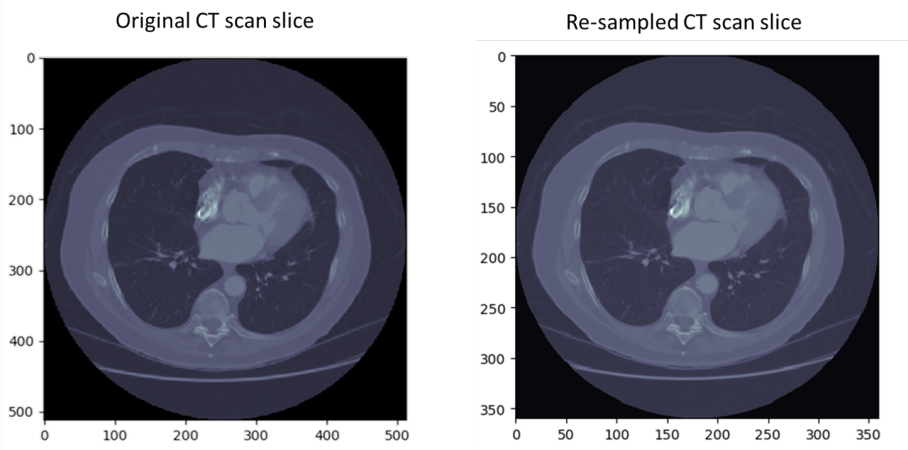

Fig. 3.3. Original and re-sampled slices of a CT scan obtained from the LIDC-IDRI database.

3.5 The algorithm workflow

Estimating the presence or absence of lung cancer is a difficult task. Therefore, the problem was divided in four specific sub-problems, defined as: 1) detecting nodules in the CT, 2) estimating morphological characteristics of nodular findings, 3) calculating the likelihood of nodule malignancy based on radiologic assessments, and 4) estimating the absence or presence of lung cancer based on different diagnosis methods: biopsy, surgical rejection, progression, response, or a 2-year nodular radiologic review. The algorithm workflow is similar to the end-to-end strategy developed by Ardila *et al.* [28]. A schematic illustrating the algorithm workflow is shown in Figure 3.4. All programming frameworks used for training the CNNs include: Tensorflow, Keras, Hyperopt, and

Numpy.

The process starts by extracting a sub-volume $\boldsymbol{I_V} \in \mathbb{R}^3$ within a normalized CT scan $\boldsymbol{I^{(i)}_{norm}}$, with a specific receptive field $r_{field} = (n_x \text{ x } n_y \text{ x } n_z)$. The first stage comprises a CNN receiving a CT portion I_V, which outputs a conditional probability for nodular presence $\hat{y_{nodule}}$, as specified in equation (3.4).

$$\hat{y_{nodule}} = P(y_{nodule} = 1/\boldsymbol{I_V}) \tag{3.4}$$

Based on a specific threshold thr_{nodule} , the algorithm decides if the input I_V corresponds to the presence of nodule ($y_{nodule} = 1$) or the absence of it ($y_{nodule} = 0$). If the CNN decided that there was no nodule on $\boldsymbol{I_V}$, it would slide the window to extract the next CT portion. Otherwise, if it predicted that a nodule was present, a second and third CNNs would receive as input the same sub-volume $\boldsymbol{I_V}$. The second CNN comprises several sub-networks j, where each one of them would estimate the presence or absence of the j^{th} morphological characteristic $morph_j$ in the previous detected nodule $\boldsymbol{I_V}$, as defined in equation (3.5).

$$\hat{y_{morph_j}} = P(y_{morph_j} = 1/\boldsymbol{nodule}) \tag{3.5}$$

On the other hand, the third CNN will output the conditional probability of the detected nodule being malignant or benign, as illustrated in equation (3.6).

$$\hat{y_{malignant}} = P(y_{malignant} = 1/\boldsymbol{nodule}) \tag{3.6}$$

Similar to the process implemented in the first CNN, the second and third CNNs implement thresholds thr_{morph_j} and $thr_{malignant}$ to decide if their respective anomalies are present in $\boldsymbol{I_V}$. By sliding the window over the CT scan, different portions $\boldsymbol{I_V}$ are extracted and processed using the process described previously. As a result, the algorithm generates $n_o = j + 1$ output matrices, where $\boldsymbol{O_{malignancy}}$ and $\left\{\boldsymbol{O_{morph_1}}, \boldsymbol{O_{morph_2}}, ..., \boldsymbol{O_{morph_j}}\right\}$ correspond to estimation matrices for the presence of both, malignancy and the set of j morphological characteristics.

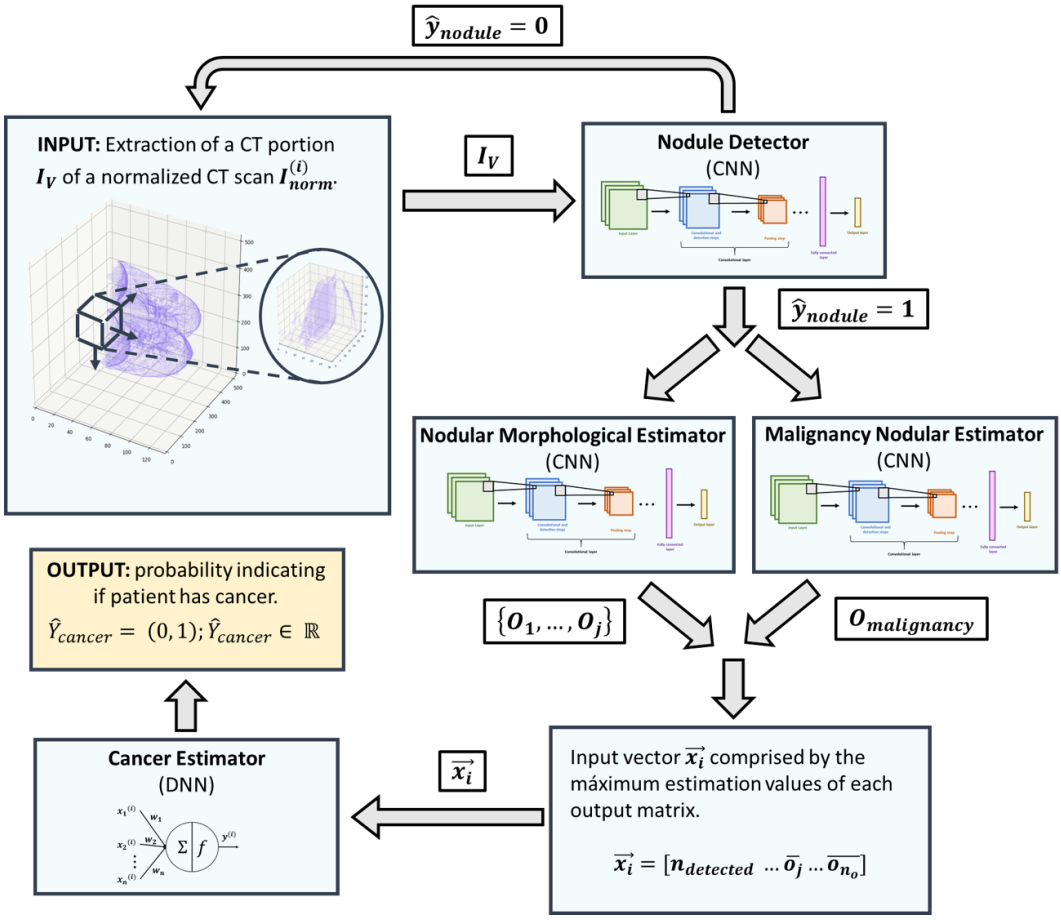

Fig. 3.4. Schematic describing the inputs of each stage of the algorithm, as well as the outputs estimated by each CNN.

Each component o_i inside the n_o matrices corresponds to a single estimation performed by the respective CNN, which defines if the corresponding anomaly is present in a scan portion $\boldsymbol{I_V}$ given that it was previously defined as a nodule by the first CNN. Because the main interest is to locate nodules and define a probability for the presence of malignancy and a specific set of morphological characteristics, an intersection probability is computed using Equation (3.7), where $P(nodule)$ is calculated with expression (3.4), and $P(anomaly/nodule)$ refers to either the probability computed in Equation (3.6) or the calculation performed using expression (3.5).

$$P(anomaly \bigcap nodule) = P(anomaly/nodule)P(nodule) \tag{3.7}$$

As a result, all components comprising matrices $\boldsymbol{O_{malignancy}}$ and $\left\{\boldsymbol{O_{morph_1}}, ..., \boldsymbol{O_{morph_j}}\right\}$ are changed from conditional probabilities to their respective intersections using Equation (3.7). Finally, a component max estimation is computed for each one of the matrices $\boldsymbol{O_{malignancy}}$ and $\left\{\boldsymbol{O_{morph_1}}, \boldsymbol{O_{morph_2}}, ..., \boldsymbol{O_{morph_j}}\right\}$ using Equation (3.8). In this expression, $\bar{o_j}$ represents the maximum value of all the k estimations o_i comprising a specific output matrix j computed by the malignancy classifier or a particular morphological estimator.

$$\bar{o_j} = max_{o_i} O_j \tag{3.8}$$

A classical DNN would receive as input a vector $\vec{x_i} \in \mathbb{R}^{n_o+1}$ that corresponds to the i^{th} patient, which is also represented in Equation (3.9). The first component of $\vec{x_i}$ represents the number of detected nodules $n_{detected}$ by the nodule detector, while the remaining components correspond to all the max estimated values $\bar{o_j}$ that were calculated with Equation (3.8).

$$\vec{x_i} = \begin{bmatrix} n_{detected} & \dots & \bar{o_j} & \dots & \bar{o_{n_o}} \end{bmatrix} \tag{3.9}$$

Therefore, the classical NN collects the input vectors $\vec{x_i}$, and produces a final estimation $\hat{y_{cancer}}$, indicating wether the patient might have cancer. Equation (3.10) defines the conditional probability computed by the last CNN.

$$\hat{y_{cancer}} = P(y_{cancer} = 1/\vec{x_i}) \tag{3.10}$$

3.6 Nodule detector

As specified in Section 3.5, the first step consists in creating a CNN that will take sub-regions $\boldsymbol{I_V}$ from a normalized CT scan $\boldsymbol{I^{(i)}_{norm}}$ as input, and outputs a number $y_{nodule} \in [0, 1]$ where 0 represents "no nodule" and 1 defines "nodule is present". Specifically, this network was divided into two sub-networks: 1) a candidate nodule selection network and 2) a false positive reduction (FPR) architecture; a common approach developed by other researchers [72, 79]. Appendix D contains all the scripts programmed for the nodule detector algorithm.

3.6.1 Candidate nodule detection CNN

The candidate detection (CD) stage consists in detecting *"nodule candidates at a very high sensitivity, which typically comes with many false positives"* [72]. Data bases A and B (see Section 3.2.1) were used for extracting all the nodules used for training, validating, and testing the algorithm. Only data base A was required for the training and validation stage. All nodule extractions were performed by identifying automatically all labels assigned by at least one expert radiologist, and only nodules $\geq$ 3 mm were considered as relevant findings; as specified by the Guidelines for Management of Incidentally Detected Pulmonary Nodules in Adults [18].

Additionally, several experiments were performed with different inclusion criteria. More precisely, several experiments included only labels assigned to a specific instance, in which three different levels of agreement were considered: 1) all four radiologists agreed in that specific annotation, 2) at least two experts assigned the same label for a particular nodule, and 3) at least one radiologist labeled the particular instance as nodule.

To verify that multiple labels assigned by each radiologist corresponded to the same instance, the percentage of intersection was computed for all pair combinations of volume labels $V_{radiologist\,i}$ and $V_{radiologist\,j}$ using equation (3.11). Any percentage greater than 10% was considered as the same true positive label. With these specifications, a total of 896, 1,374, and the same 1,374 positive instances $n_{pos}^{(i)}$ were collected, for the corresponding three levels of agreement previously defined.

$$Inodule = \frac{V_{radiologist\,i} \bigcap V_{radiologist\,j}}{V_{radiologist\,i}} \tag{3.11}$$

Because of inconsistencies with the specified nodular receptive field and their corresponding nodule annotations, only 1,347 nodules $n_{pos}^{(i)}$ were considered for building the data set, in which a 10 cross-fold validation was performed. Many CNNs were trained and validated using all the collected data: 896 and 1,347 positive instances, respectively, where several Python generators were built and a classic train/test split of 70%/30% was implemented due to memory constraints.

For defining the receptive field size, three histograms corresponding to each dimension size were computed for all nodules $\geq$ 3 mm and $<$ 3mm. All dimensions were obtained after applying both normalization transformations to each CT, as defined in Section 3.4.2 Figure 3.5 illustrate each one of the histograms computed for each nodular size. All sizes were collected after the normalization step was applied (refer to Section 3.4.2). As a result, the optimal receptive field was specified as $(32$ x 32 x $32)$, which encompasses 97% of all nodule sizes contained in data base A. For future work, multiple receptive fields could be implemented following a similar approach as the one developed by Dou *et al.* [79]. A sample of CT slices containing a nodule is displayed in Figure 3.6.

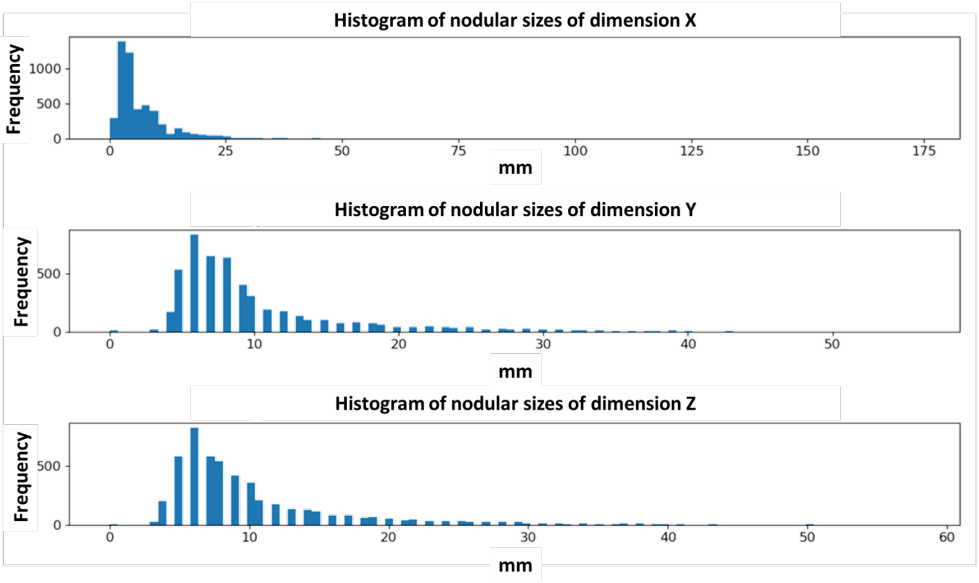

Fig. 3.5. Histograms representing nodular sizes in each dimension for all nodules contained in data base A.

Because of the limited amount of data, four data augmentation techniques were developed: 1) nine lossless information rotations were applied to each instance: 1) 90°, 180° and 270° rotations with respect to each main axis of the CT, 2) Generative Adversarial Networks (GANs) were developed to generate more artificial nodules [101, 102], 3) nodule rescaling using the nearest neighbors algorithm, and 4) seven random image translations with respect to each possible combination of axes (X, Y, Z, XY, XZ, XYZ) with a maximum translation value of 10 pixels. CNNs were the

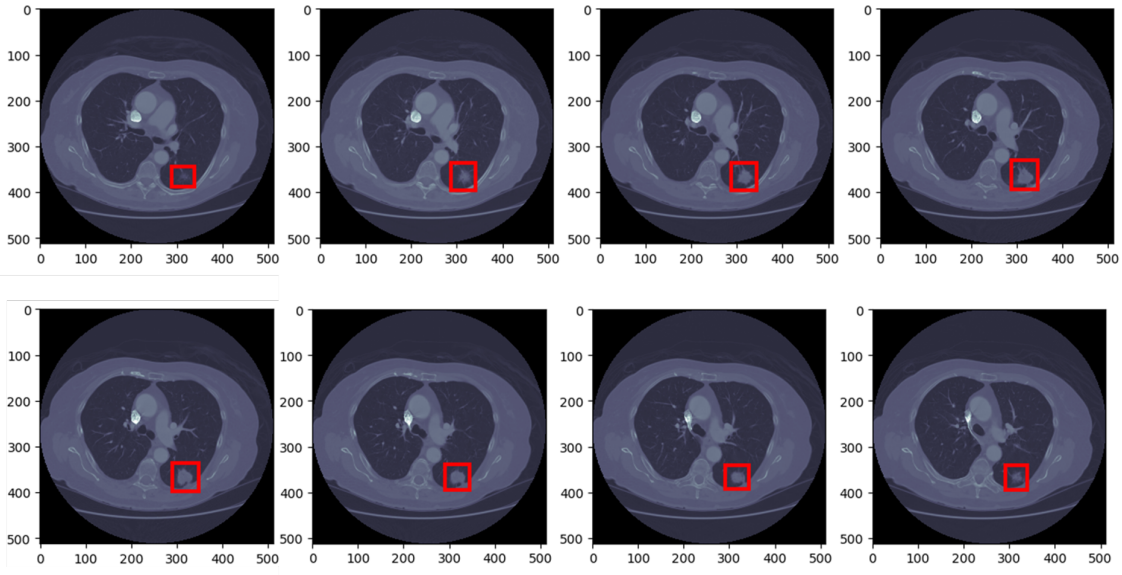

Fig. 3.6. Sample of CT slices where different red bounding boxes indicate where the nodule is located within the scan.

preferred architecture for both, the discriminator and generator, when implementing GANs. A receptive field of $(32 \text{ x } 32 \text{ x } 32)$ was specified for the discriminator input size and the generator output size. The Adam optimization technique, defined in Equations (2.17) - (2.21), was used for training both CNNs. Adam parameters were set to $\beta_1 = 0.5$, $\beta_2 = 0.999$ and $\alpha = 0.0002$. Also, Xavier initialization was implemented by using Equation (2.6).

Due to hardware constraints, a batch size of 32 instances was set for each training step and 3000 epochs (iterations) were specified. An output example generated by the trained GANs can be seen in Figure 3.7, which illustrates a sample of 100 artificial generated nodules. Each artificial nodule has a volume equal to $V_{nodule} = 32 \text{ x } 32 \text{ x } 32$ pixels. Scripts related with GANs can be seen in Appendix D, where each programming implementation was obtained from [103] and adapted to meet the specifications of this research.

Using both, data rotations and GANs, $n_{pos}^{(i)}$ nodules were augmented to build the entire set of training positive examples, where the number of nodules $n_{pos}^{(i)}$ varied depending on which training

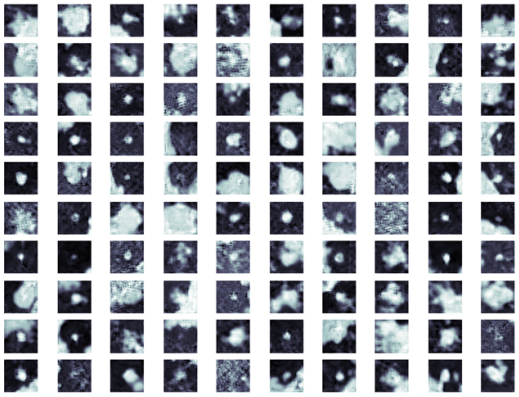

Fig. 3.7. Sample of 100 artificial nodule middle slices generated by the trained GANs after 3000 epochs.

strategy was implemented: a classic train/test split or a 10-cross fold validation. With respect to negative instances $n_{neg}^{(i)}$, random crops not containing nodules were sampled from each normalized CT scan. Also, for some models, several instances labeled as "non-nodule $\geq$ 3mm" were collected from Data Base A.

Also, in many experiments different data augmentation techniques were applied, where three possibilities were considered: 1) only the nine lossless information rotations, 2) only GANs, and 3) both augmentation techniques. As a result, the training set size varied in the number of samples $n^{(i)}$ due to both, the three data augmentation implementations and the three radiologist levels of agreement.

Hyper-parameter optimization was the most difficult challenge to overcome in this research project because of the large number of parameters that needed to be tuned. The tree-structured Parzen estimator (TPE), a SMBO algorithm, was the preferred strategy to search for optimal parameters in the hyper-dimensional configuration space [56]. Specifically, the Hyperopt Python library was used to approximate the surrogate function based on previous observations [95, 104]. Even with this algorithm, there was another major problem to address: selecting the best network configuration to specify the search space of hyper-parameters. As it has been reported in previous

studies, many CNN architectures have been tested for different pattern recognition tasks, such as ResNet, Inception Network, LeNet, Vgg, and Inception-ResNet; to name a few [24–26, 65, 66]. Therefore, several trials were performed with variations of the aforementioned configurations using TPE, where the objective metric was defined as the cost function specified in equation (2.4) specifying the validation loss. Specifically, between 10 and 30 trials (model variations) were evaluated for each of the proposed CNN architectures depending on the complexity and size of the network. For each trial of TPE, a 10-fold cross-validation procedure was chosen for training and validating the nodule candidate generation CNN. Nevertheless, other models used a classic 70%/30% train/test split during the hyper-parameter optimization process. For the 10 fold cross-validation a range of 5 to 20 epochs was chosen because of expensive computation times. On the other hand, for models with a classic train/test split, a range between 15 and 50 epochs was the preferred choice.

Following a similar approach as the one implemented when developing GANs, Adam optimizer and Xavier initialization were selected to train and initialize network weights for each model in a specific TPE trial. For this stage, hyper-parameters β_1 and β_2, included in equation (2.21), were fixed to specific values $\beta_1 = 0.9$ and $\beta_2 = 0.999$, which have worked well for other type of applications [54]. Thus, these hyper-parameters are not included in the search configuration space allowing a dimensionality reduction of such space.

The parameter configuration search space was defined differently based on each CNN architecture. In other words, different sampling ranges for distinct parameters were defined based on each network topology. For example, Table 3.2 illustrates all the parameters that were proposed for a classical CNN with a particular predefined topology (based on experiments), their search ranges, and their respective sampling distributions. In this case, the specific predefined topology is shown in Figure 3.8, which is basically a typical CNN (see Figure 2.5). Even though, there are slight differences in the proposed architecture:

1. The inclusion of batch normalization in each layer by using Equation 2.12.

2. The implementation of Tikhonov regularization per layer following a similar computation as

the one defined in Equation (2.9).

3. The number of filters F_i are reduced after half of the layers have been defined because of hardware limitations.

4. All max pooling layers have constant filter sizes of $(3\ x\ 3)$.

5. A final average pooling layer Avg is assigned before the fully-connected layer.

For details corresponding to the specific hyper-parameter configurations of the other predefined network architectures (ResNet, Inception, Vgg, LeNet, Inception-ResNet) refer to Appendix D.

Table 3.2. Hyper-parameters and sampling distributions to establish the sample search space for a specific CNN architecture

HYPER-PARAMETER	SAMPLING DISTRIBUTION	SEARCH RANGE
Learning rate α	Continuous uniform	[0.0005, 0.001]
Decay learning constant	Continuous uniform	[0.01, 0.05]
Number of filters F_i for the i^{th} block if $i \leq m/2$.	Discrete uniform	{0, 1, 2, 3, 4}
Number of filters for the i^{th} block if $i > m/2$.	Discrete uniform	{0, 1}
Filter size f_i for the i^{th} block.	Discrete uniform	{0, 1, 2, 3}
L2 regularization constants l_i for the i^{th} block.	Continuous uniform	[0.0001, 0.01]
Number of x dense layers	Discrete uniform	{0, 1, 2, 3}
Number of units n per dense layer	Discrete uniform	{0, 1,… , 15}
Number of average pooling layers	Discrete uniform	{0, 1}

Once the best model configuration was obtained after implementing the TPE algorithm, either a 10-cross fold validation or a classic training/test optimization (based on model training times) was implemented to keep improving the CNN's performance. Different number of training epochs were specified heuristically based on hardware limitations and model complexity. For the nodule

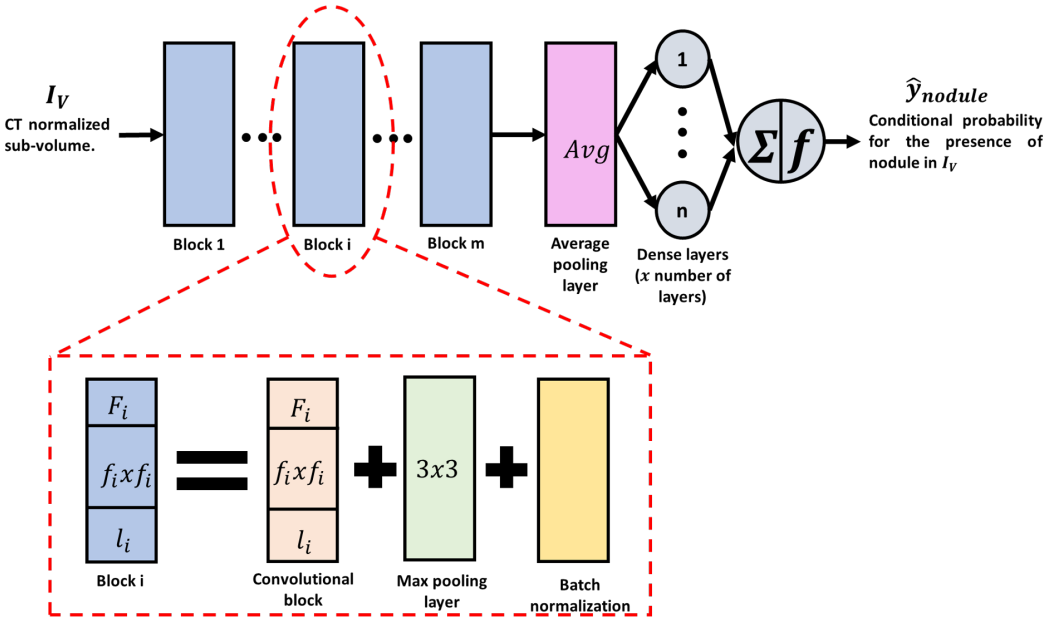

Fig. 3.8. Predefined network architecture and the hyper parameters that need to be tuned using TPE, for the nodule candidate generation step.

candidate detection CNN, the threshold was chosen following the same strategy as reported by Ignacio Sánchez: when the model sensitivity was equivalent to the specificity [105].

3.6.2 False positive reduction CNN

All training data used for this stage came from results obtained by running the CD algorithm. More precisely, once the best CNN architecture for the CD stage was obtained, the algorithm was run over a set of k normalized CT scans $\boldsymbol{I}_{norm}^{(i)}$ contained in database A. As expected, several FP per scan were generated during this process. As its name suggests, the main objective of the FPR CNN is to reduce the false positives generated by the CD CNN. Thereby, all negative data consisted of n_{FP} FP generated from the previous step. On the other hand, following the same approach as the one reported for the CD stage, positive data comprised different number of nodules $n_{nodules}$ depending on the three levels of agreement previously defined. In consequence, the total training data consisted of $m = n_{FP} + n_{nodules}$ instances. Again, due to hardware limitations, the number of training examples was reduced depending on the CNN architecture and training times.

A similar methodology reported for the CD stage was applied for the FPR task. Therefore, data augmentation, model selection, training and validation were essentially the same. A summary for every part of the FPR methodology is provided below (for a detailed explanation refer to section 3.6.1):

1. **Data preparation:** positive data consists of all $n_{nodules}$ nodules in database A. Moreover, negative data consists of all n_{FP} FP produced by the candidate generation CNN.

2. **Data augmentation:** five different data augmentation possibilities were tried following the same procedures of the CD stage: 1) 9 loss-less 3D rotations, 2) GANs, 3) nodule rescaling using the nearest neighbors algorithm, 4) seven random image translations with respect to each possible combination of axes (X, Y, Z, XY, XZ, XYZ) with a maximum translation value of 10 pixels, and 5) different combinations of the aforementioned techniques. No data augmentation was applied to negative instances.

3. **Hyper-parameter optimization:** the TPE algorithm was used for selecting the best set of hyper-parameters for each variation of the following network architectures: classic CNN, LeNet, Vgg, ResNet, Inception, and ResNet-Inception. Specifically, TPE was run between 10 to 30 trials depending on training times of each network; where each trial consisted on either a 10-cross fold validation process or a classical split of 70% training data and 30 % test samples.

4. **Training and validation:** once the best CNN was obtained using the TPE algorithm, either a 10-cross fold validation or a classical train/test split process was implemented to keep improving the model. Training and validation during both, hyper parameter searching and best model optimization, were performed using the Adam optimizer and Xavier initialization. For selecting the best threshold, the same technique used in Section 3.6.1 was implemented.

By combining and designing both stages, the nodule candidate detection CNN and the false positive reduction CNN, nodule detection can achieve reliable results. Thus, the algorithm could be implemented in real clinical contexts, where automated detection and standardized nodular

assessments are required. Furthermore, by providing a robust nodule detection system, the following stages in this dissertation can be implemented over a solid foundation.

3.7 Malignancy classifier

Several research studies have focused on classifying lung nodules as benign or malignant, regardless of whether nodular assessment was done from a radiologic or histologic perspective (Section 2.3.2 provide a thorough description of these studies). Therefore, the main objective was to develop a CNN that classified a pulmonary nodule as benign or malignant based on a subjective radiologic perspective. Specifically, this algorithm takes as input a portion of a normalized CT scan $\boldsymbol{I_V}$, previously defined as a nodule by the FPR CNN (refer to Section 3.6.2), and outputs a probability indicating malignancy, as defined in Equation (3.6).

All data for training, validation, and testing was obtained from Data Bases A and B (see Section 3.2.1), where only Data Base A was used for training and validating the architecture. Because malignancy classification is a crucial step in the algorithm, only nodules annotated with an agreement level of all four radiologists was considered. Moreover, to ensure model generalization, a 10 cross-fold validation was performed during model selection and optimization. Therefore, with these considerations, a total of 896 nodules were collected from Data Base A. Even so, there were models where other levels of agreements were considered, but they were discarded when evaluating their training and validation results.

To differentiate between malignant and benign nodules, all malignancy labels provided by the four expert radiologists were binarized, a strategy similar to the one defined by Shen, *et al.* [70]. Therefore, any nodule that was assigned with a malignancy score ≥ 3 was defined as malignant, otherwise it was established as benign. After applying this heuristically convention, 213 nodules were defined as benign and 722 anomalies were specified as malignant. It is important to mention that many experiments were performed without the binarization process. Without binarizing, scores $\hat{y}^{(i)}$ illustrated in Equation (2.4), were computed using the categorical softmax function instead of

its binary variant; the sigmoid unit represented in Equation (2.2).

Because of a lack of data for training and validation, the same two data augmentation techniques from the CD stage were developed to increase the training set size: 1) 9 lossless 3D transformations (90°, 180°, and 270° rotations with respect to each main axis of the CT) and 2) GANs for generating both benign and malignant artificial nodules [101, 102]. Also, there were models where only one of the two augmentation techniques were applied, but those CNNs were not included in the dissertation because of poor performance. Two GANs were implemented to generate benign and malignant artificial nodules. CNNs were the preferred architecture to develop the discriminator and generator for both GANs. As defined in Section 3.6.2, both the discriminator input and generator output sizes were defined with a receptive field size of $(32 \text{ x } 32 \text{ x } 32)$. Smooth labeling was implemented during the training process. To establish binary labels, the likelihood of malignancy was set based on a score ≥ 3, which was assigned be each of the four radiologists in Data Base A.

The Adam optimization algorithm with Xavier Initialization was chosen for training each CNN, where all hyper-parameter values specified in Equations (2.17), (2.18), (2.19), (2.20), and (2.21) were set to: $\beta_1 = 0.5$, $\beta_2 = 0.999$ and $\alpha = 0.0002$. Finally, due to hardware constraints, a batch size of 32 and 3000 training epochs were defined for optimizing both GANs. Figures 3.9 and 3.10 show the results after training both networks: the malignancy and benign nodule GANs, respectively.

By augmenting both, the malignant and benign samples, using the two augmentation techniques, a total of 6,019 instances were collected for the training set; where 3,000 and 3,020 cases comprised benign and malignant nodules, respectively. For the validation set, 64 benign and 217 malignant instances were included. Following the same procedure of the nodule detector stage (Section 3.6.1), hyper-parameter selection was performed using the TPE algorithm. Again, the same CNN architectures were chosen to solve the binary classification problem: classical CNN, ResNet, Inception Network, Vgg, LeNet, and Inception-ResNet. Each model was adapted to fulfill hardware requirements. To see a specific example refer to Figure 3.8.

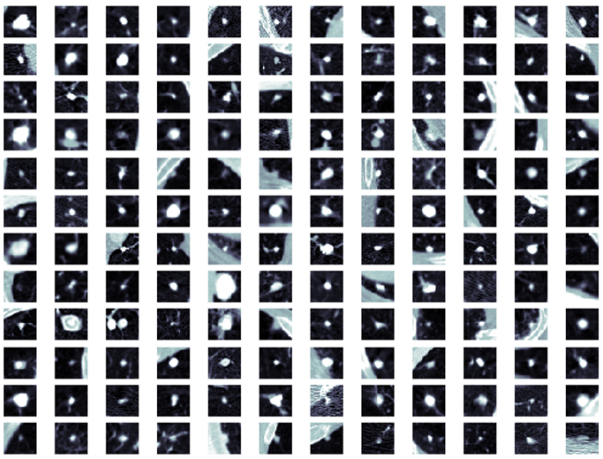

Fig. 3.9. Sample of 144 artificial benign nodule middle slices generated by one of the two trained GANs after 3000 epochs.

Fig. 3.10. Sample of 144 artificial nodule middle slices likely malignant generated by one of the two trained GANs after 3000 epochs.

Several trials were performed with variations of the aforementioned configurations using TPE, where the objective metric was defined as the cost function specified in equation (2.4). Specifically, between 10 and 30 trials (model variations) were evaluated for each of the proposed CNN architectures depending on the complexity and size of the network. For each trial of TPE, a 10-fold cross-validation procedure was chosen for training and validating the malignancy classifier CNN. A range of 5 to 20 epochs was chosen because of expensive computation times. Repeating the

same procedure of Section 3.6.1, training was performed using the Adam optimizer with Xavier initialization, and fixing the hyper-parameter values of β_1 and β_2 to 0.9 and 0.999, respectively [54].

Once the best model configuration was obtained, a 10-cross fold validation was implemented to keep improving the CNN's performance. The number of training epochs was specified based on hardware limitations and model complexity. Finally, the binary classifier threshold $thr_{malignant}$ was specified by using the same strategy reported in Section 3.6.1 [105]. It is important to illustrate that to obtain clinical performance for this stage, it is crucial to make accurate lung cancer predictions. Although, training these CNNs with subjective radiologic assessments of malignancy likelihood is not sufficient to provide an objective system that can be deployed in real clinical contexts. Therefore, there are other steps that need to be implemented to ensure the required clinical specifications.

3.8 Morphological estimator

One of the main problems of NN is related to model interpretability, where these models are considered as "black-boxes" [106]. Specifically, in the medical regime, interpretability becomes a major issue to address because of the implications derived from the model predictions; where justifying a medical decision becomes an important task to do when diagnosing and treating a patient. However, several efforts have focused on making interpretable models for lung nodule malignancy classification [70]. Therefore, the main objective is to provide model interpretability and relevant morphological information to justify the final lung cancer prediction. As specified by Lipton, this stage focused primarily on model decomposability, where *"each part of the model admits an intuitive explanation"*.

In this case, all intuitions reside on the morphological characteristics of each detected nodule. More concretely, these CNNs will receive as input a normalized sub-volume $\boldsymbol{I_V}$, which was previously detected as a nodule (refer to Section 3.6.2), and will output a normalized real value $\hat{y_{morph_j}}$, which indicates the j^{th} magnitude of a specific morphological characteristic present in that sub-volume. Thus, a value $\hat{y_{morph_j}} = 0$ indicates the absence of that particular morphological char-

acteristic, while $\hat{y_{morph_j}} = 1$ illustrates that such property is completely defined in that sub-volume.

In Data Base A, eight subjective radiologic characteristics were annotated (see Section 3.2.2). Two of them are not included in this analysis: the likelihood of malignancy because it has been included already (refer to Section 3.7) and calcification because there are inconsistencies present on the labels assigned by the radiologists. In consequence, there are six remaining characteristics that can be analyzed and incorporated to the workflow. Because of computational time and workflow complexity, a simple statistical analysis was performed to detect the two most positive correlated characteristics with the likelihood of malignancy. More precisely, a correlation matrix was computed for the seven morphological characteristics (including likelihood of malignancy) of all nodules assessed by the four expert radiologists in Data Base A. Figure 3.11 illustrates the correlation matrix and Table 3.3 shows how much each one of the six characteristics are correlated with the likelihood of malignancy. As a result, two main morphological characteristics were identified as the most positively correlated with respect to likelihood of malignancy: spiculation and lobulation. Therefore, two CNNs were developed to estimate the presence of each characteristic in the previously analyzed sub-volume $\boldsymbol{I_V}$.

Table 3.3. Correlation results of each morphological characteristic with respect to the likelihood of malignancy.

	malignancy
subtlety	0.21669795
sphericity	-0.1488796
margin	-0.22058295
lobulation	0.30918384
spiculation	0.34524731
texture	-0.10212231

	subtlety	sphericity	margin	lobulation	spiculation	texture	malignancy
subtlety	1	-0.00367608	0.30471617	0.16053637	0.17710993	0.42557599	0.21669795
sphericity		1	0.27903965	-0.20226738	-0.20278752	0.07977542	-0.1488796
margin			1	-0.25946339	-0.27119143	0.68275991	-0.22058295
lobulation				1	0.54423427	-0.05378396	0.30918384
spiculation					1	-0.05689417	0.34524731
texture						1	-0.10212231
malignancy							1

Fig. 3.11. Correlation matrix computed for the eight morphological characteristics of all nodules, colors tending to the green spectrum indicate a higher positive correlation.

3.8.1 Spiculation estimator CNN

Similar to the approach developed for the malignancy classifier, a binarization process was performed for all scores assigned by each of the four expert radiologists in Data Base A; where all training and validation data was collected from data base A. On the other hand, data base B was used for testing purposes.Specifically, all labels (including calcification and likelihood of malignancy) assigned by the four expert radiologists in data base A were collected, giving a total of 2,624 scores for each morphological characteristic (without considering calcification). Subsequently, several histograms were built to identify class imbalances. Figure 3.12 show all the histograms that were computed for each morphological characteristic, where the frequency sum for each histogram give a total of 2,624 scores. On the other hand, Table 3.4 illustrates the number of scores assigned to each class of the morphological characteristics.

To create the most balanced class ratio for spiculation, the binarization process was performed

by assigning ”1” to any label score ≥ 2 and ”0” otherwise. Also, an agreement level of at least four radiologists, for each detection, was considered. After applying these heuristics, a total of 245 nodules were defined as ”spiculated”. Moreover, 693 instances were specified as negative examples. Thus, it is important to note the class imbalance present in the two desired characteristics: lobulation and spiculation.

Data augmentation was applied following the same procedure for the Malignancy Classifier: 1) apply the same 9 lossless 3D transformations and 2) develop two different GANs that will generate both, artificial ”non-spiculated” and ”spiculated” nodules. Training specifications for optimizing both GANs were the same: use the Adam optimizer with Xavier initialization and define the hyper-parameter values to be $\beta_1 = 0.5$, $\beta_2 = 0.999$ and $\alpha = 0.0002$. Also, the same receptive field size was defined for both the discriminator input and the generator output: $(32 \text{ x } 32 \text{ x } 32)$. A batch size of 32 and 3000 training epochs were defined for optimizing both GANs. Figures 3.13 and 3.14 show the results of both GANs.

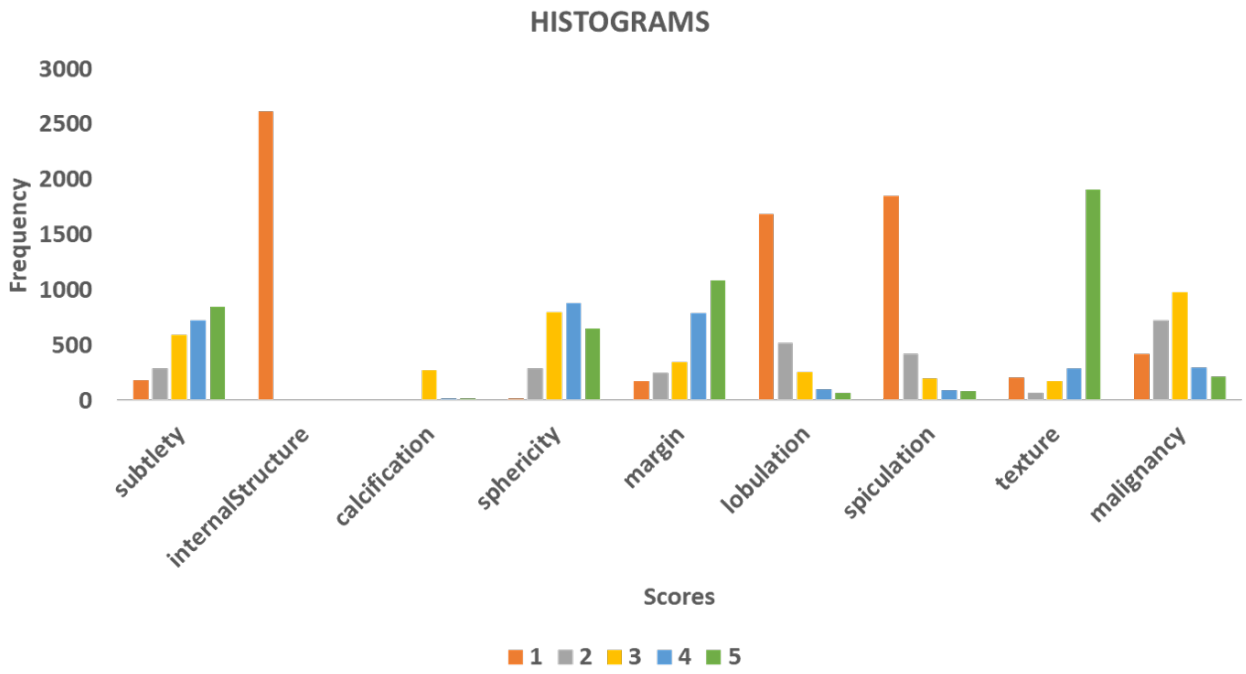

Fig. 3.12. Histograms computed for each morphological characteristic.

Table 3.4. Number of scores assigned to each class of the eight morphological characteristics.

Class	subtlety	Internal Structure	calcification	sphericity	margin	lobulation	spiculation	texture	malignancy
1	181	2609	2	15	169	1681	1847	202	415
2	283	4	5	290	242	521	417	62	720
3	594	0	269	798	344	258	193	170	980
4	722	10	17	875	786	101	87	284	295
5	844	1	16	646	1083	63	80	1906	214

Fig. 3.13. Sample of 169 artificial ”spiculated” nodule middle slices generated by one of the two trained GANs after 3000 epochs.

By augmenting both the ”spiculated” and ”non-spiculated” samples using the two augmentation techniques, a total of 6,000 instances were collected for the training set, where 3,000 and 3,000 cases comprised ”spiculated” and ”non-spiculated” nodules, respectively. For the validation set, 207 ”non-spiculated” and 103 ”spiculated” instances were included.

Using the same procedure of the previous stages, several trials of the TPE algorithm were performed with variations of different CNN configurations (refer to previous sections). Again, the

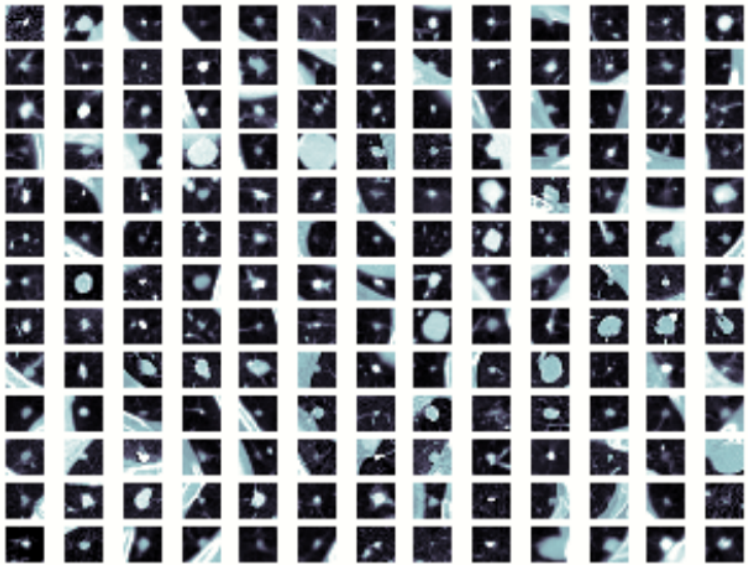

Fig. 3.14. Sample of 169 artificial ”non-spiculated” nodule middle slices generated by one of the two trained GANs.

objective metric was defined as the cost function specified in equation (2.4). Specifically, between 10 and 30 trials (model variations) were evaluated for each of the proposed CNN architectures. A 10-fold cross-validation procedure was chosen for training and validating each new CNN configuration at each TPE trial. Optimization was performed using the Adam optimizer with Xavier initialization, where hyper-parameter values of β_1 and β_2 were defined as 0.9 and 0.999, respectively [54].

After obtaining the best CNN configuration, a 10-cross fold validation was implemented to keep optimizing the best model. In this case, different number of training epochs were specified based on hardware limitations. Finally, the binary classifier threshold thr_{morph_1} was specified by using the same strategy reported in Sections 3.6.1 and 3.7; where $j = 1$ indicates that spiculation is the first morphological characteristic of the two analyzed nodular properties [105].

3.8.2 Lobulation estimator CNN

Because the procedure is extremely similar to the one developed for the spiculation estimator, a summary is provided with the main steps that were performed to implement the lobulation estimator

CNN:

1. **Binarization process:** to create the most balanced class ratio for lobulation, the binarization process was performed by assigning ”1” to any label score ≥ 2 and ”0” otherwise. Also, an agreement level of at least four radiologists for each detection was considered.

2. **Data preparation:** after applying the binarization process, a total of 348 nodules were assigned as ”lobulated” and 693 examples were defined as negative instances.

3. **Data augmentation:** two different data augmentation techniques were implemented: 1) the same 9 lossless 3D rotations and 2) GANs that generated both ”lobulated” and ”non-lobulated” nodules. Training specifications for GANs were the same as the ones implemented for the spiculation estimator. Results after training for both networks can be seen in Figures 3.15 and 3.16.

4. **Hyper-parameter optimization:** the TPE algorithm was implemented to select the best set of hyper-parameters for the same CNN configurations (refer to Section 3.7). Specifically, TPE was run between 10 to 30 trials, where each trial consisted on a 10-cross fold validation process.

5. **Training and validation:** once the best CNN was obtained using the TPE algorithm, a 10-cross fold validation was implemented to keep improving the CNN. Training and validation during both, hyper-parameter searching and best model optimization were performed using the Adam optimizer and Xavier initialization. For selecting the best threshold thr_{morph_2}, the same technique used in Section 3.6.1 was implemented.

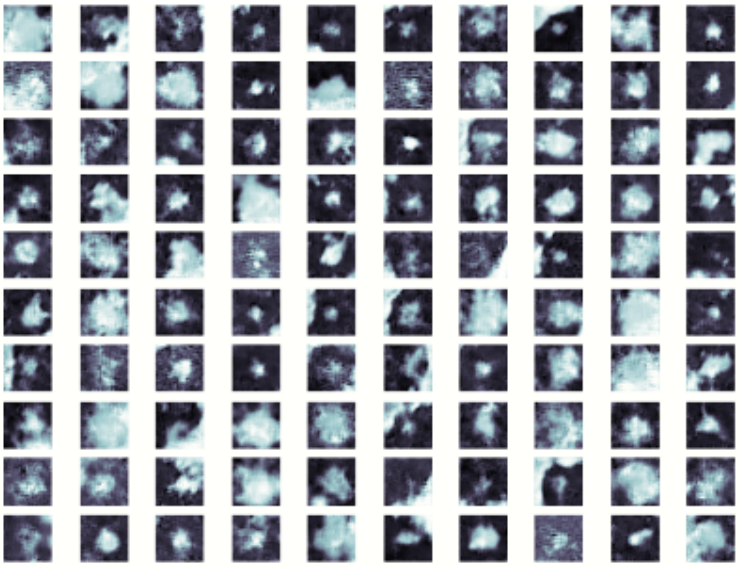

Fig. 3.15. Sample of 100 artificial ”lobulated” nodule middle slices generated by one of the two trained GANs after 3000 epochs.

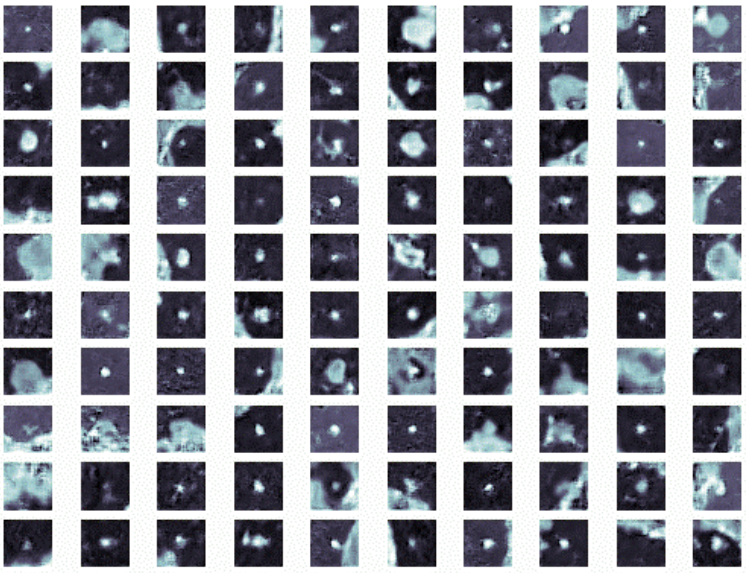

Fig. 3.16. Sample of 100 artificial ”non-lobulated” nodule middle slices generated by one of the two trained GANs.

3.9 Cancer estimator

This final step of the algorithm is the most important one, as it integrates all of the aforementioned steps to make a final prediction. Specifically, the predictions are performed given a feature vector $\vec{x_i}$ represented in Equation (3.9), and the output conditional probability is computed by using expression (3.10).

For this step, data base C1 was used for training and validation, while data base C2 was defined for testing purposes. Concretely, all previous algorithms: the nodule detector, malignancy classifier, and morphological estimators, were run over the 130 CT scans of database C1 to compute their respective predictions. Input feature vectors $\vec{x_i}$ were created for each patient CT scan. A variable input size DNN was proposed, where all inputs comprised a variable sized matrix of predictions made by previous estimators. This size variability was produced because each CNN outputs different number of predictions depending on each patient CT scan. Therefore, different sized output matrices $\boldsymbol{O_{malignancy}}$ and $\left\{\boldsymbol{O_{morph_1}}, \boldsymbol{O_{morph_2}}, ..., \boldsymbol{O_{morph_j}}\right\}$ resulted for each patient.

In order to solve this problem, instead of implementing a variable sized input, the max prediction over all results contained in each matrix was computed. With this approach, a fixed sized input $\vec{x_i} \in \mathbb{R}^{n_o+1}$ was defined for the DNN, where n_o represents the number of output matrices as defined in Section 3.5. On the other hand, no data augmentation techniques were implemented because of how data was represented. Therefore, there were only 36 corroborated benign cases and 95 malignant instances comprising the whole training and validation sets.

Again, the TPE algorithm was used for determining the best hyper-parameters comprising the DNN. A range between 10 and 30 different trials of the TPE algorithm were run. Also, a 10-fold cross-validation was specified for both, training and validation, of every model at each trial. All training procedures were implemented using the Adam optimizer and Xavier Initialization with fixed hyper-parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$. After finding the best model, further optimization was performed using the same 10-fold cross validation procedure, where different number of epochs

were specified accordingly.

By integrating all predictors as specified in Figure 3.4, an end-to-end diagnostic process is achieved. Furthermore, not only malignancy and cancer predictions are computed at both, the patient and nodular levels, but some degree of interpretability is defined by including the morphological estimators. With this algorithm workflow, the proposed solution adds an extra level of interpretability that can be help radiologists verify their assessments with a second evaluation tool.

3.10 Medical ethics guidelines

Due to the retrospective nature of the study, large amounts of required data, and no negative impact exerted directly or indirectly to the patients physically or emotionally, it is not mandatory to make informed consents. Nonetheless, in case a Medical Institution and ethics committee requires an informed consent, this was included in Appendix A.

All acquisition protocols were conducted based on the legal guidelines stipulated in *"apartado 5 de la NORMA Oficial Mexicana NOM-004-SSA3-2012, Del expediente clínico y en el apartado 5 de la NORMA Oficial Mexicana NOM-035-SSA3-2012, En materia de información en salud"*.

To be coherent with Mexican and International laws, all data will be stored in a hard drive and encrypted using a 256 bits Advanced Encryption Standard (AES). Only people directly related to the research study will have access to the encrypted data. All people working in the project include: Master's student, thesis advisors with a medical degree, and expert radiologists, oncologists, and pathologists from the corresponding medical institutions where data was collected.

Only relevant patient data was used in this study: CT scan, nodular annotations, and histologic confirmation for the presence or absence of cancer. Other data was removed to protect patient's identity and integrity. No procedure included in this research study compromised the patient's physical, mental, and emotional health. No surgical, clinical, or invasive procedure was required for

the research to be conducted.

All legal issues that arise with respect to this dissertation will be responsibility of the people working in the project as stated above, and will be submitted to revision based on the Mexican Law.

CHAPTER 4

Results and discussion

4.1 Nodule detector

4.1.1 Candidate nodule detection CNN

Approximately 100 experiments were run to find the best model for the nodule candidate generator. Several variations of the specifications enlisted below were tried in each experiment:

1. Different inspired CNN architectures: AlexNet, LeNet, Vgg, GoogleNet, ResNet, ResNet-Inception, and a classsical CNN.

2. Using either one, none or both augmentation techniques: GANs and the 9-lossless 3D rotations.

3. Hyper-parameter optimization: manual or TPE algorithm.

4. Training during model search: a classical 70%/30% train/test split (with or without Python generators) or a 10-fold cross-validation.

5. Training the best model: a classical 70%/30% train/test split (with or without Python generators) or a 10-fold cross-validation.

6. Different number of epochs during model search and best model training.

7. Negative instances were defined as either random sample crops not containing nodules, labels annotated by radiologists as "non-nodule $\geq$ 3 mm", or both.

During model search, the TPE algorithm tried 10 different model configurations. At each trial, both data augmentation techniques, GANs and nine lossless 3d rotations, were applied to the data set.

A random 10-fold cross validation strategy was chosen for optimizing each model configuration. All hyper-parameters that were tuned with their respective sample distributions and ranges are illustrated in Table 4.1. It can be observed that this hyper-parameter search space is similar to the one displayed in Table 3.2. The main difference is that the network is divided into four stages, where each stage has several blocks i concatenated in series. All CNNs in every stages used the same hyper-parameter space configuration with the exception of the final cancer estimator.

The best architecture was a classical CNN trained over 15 epochs using the Adam optimizer and Xavier initialization with fixed hyper-parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Table 4.2 enlists all hyper-parameter values of the best CNN, while Figure 4.1 illustrates its architecture configuration. Again, the topology is similar to the one presented in Figure 3.8, but the pattern of m convolutional blocks is repeated with different hyper-parameters at four distinct stages. For example, the best model has two, one, and none convolutional blocks for the first, second, third, and fourth stages, respectively. Each block comprises a convolutional layer, a max pooling layer and a batch normalization layer. This topology is also used for the other CNNs with the exception of the final cancer estimator.

A random 10-fold cross validation strategy was chosen to optimize the best model. Also, the same two data augmentation techniques were applied to the data set giving as a result 14,912 training and 1,657 validation examples at each random fold. Where negative instances included

Table 4.1. Hyper-parameter space configuration and sampling distributions to search the best set of values using the TPE algorithm.

HYPER-PARAMETER	SAMPLING DISTRIBUTION	SEARCH RANGE
Learning rate α	Continuous uniform	[0.0005, 0.001]
Decay learning constant	Continuous uniform	[0.01, 0.05]
m_1 = number of i blocks for stage 1.	Discrete uniform	{0, 1, 2, 3, 4}
m_2 = number of i blocks for stage 2.	Discrete uniform	{0, 1, 2, 3}
m_3 = number of i blocks for stage 3.	Discrete uniform	{0, 1}
m_4 = number of i blocks for stage 4.	Discrete uniform	{0, 1}
Number of filters F_i for the i^{th} block in stage ≤ 2.	Discrete uniform	{4, 8, 16}
Number of filters F_i for the i^{th} block in stage > 2.	Discrete uniform	{16, 32, 64}
Filter size f_i for the i^{th} block in any stage.	Discrete uniform	{(1x1), (3x3), (5x5)}
L2 regularization constants l_i for the i^{th} block in any stage	Continuous uniform	[0.00001, 0.0001]
Number of x dense layers	Discrete uniform	{0, 1, 2, 3}
Number of units n per dense layer	Discrete uniform	{0, 1,… , 15}
Number of average pooling layers with pool size = $(3x3)$ and valid padding.	Discrete uniform	{0, 1}

CT random crops not containing anomalies. Specifically, 1,273 nodules (with a radiologist-level of agreement of all four radiologists) represented the positive data set, without applying augmentation techniques.

Table 4.2. Final hyper-parameter values defining the best model configuration of the CD nodule detector.

HYPER-PARAMETER	VALUE
Learning rate α	0.000636
Decay learning constant	0.031588
m_1 = number of i blocks for stage 1.	2
m_2 = number of i blocks for stage 2.	1
m_3 = number of i blocks for stage 3.	0
m_4 = number of i blocks for stage 4.	0
Number of filters F_i for the i^{th} block in stage ≤ 2.	8 for stage 1 16 for stage 2
Number of filters F_i for the i^{th} block in stage > 2.	16 for stage 3 32 for stage 4
Filter size f_i for the i^{th} block in stage 1.	(1x1)
Filter size f_i for the i^{th} block in stage 2.	(3x3)
Filter size f_i for the i^{th} block in stage 3.	(3x3)
Filter size f_i for the i^{th} block in stage 4.	(1x1)
L2 regularization constants l_i for the i^{th} block in stages one and two.	0.00016 for stage 1 0.00019 for stage 2
L2 regularization constants l_i for the i^{th} block in stages three and four.	0.00006 for stage 3 0.00055 for stage 4
Number of x dense layers	0
Number of units n per dense layer	6
Number of average pooling layers with pool size = $(3x3)$ and valid padding.	1
Number of parameters	34,089

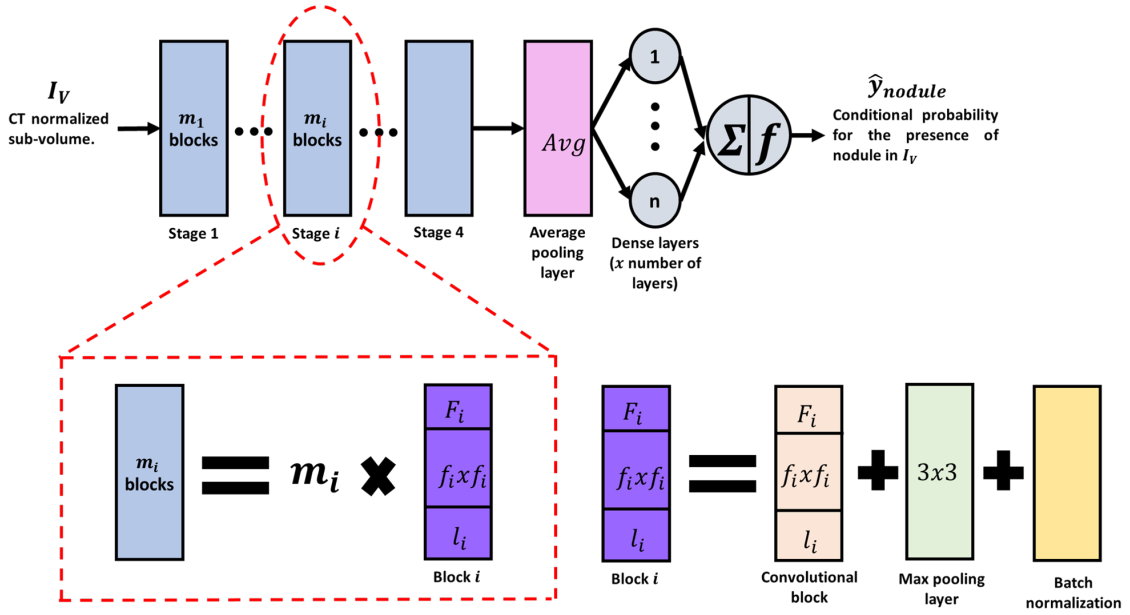

Fig. 4.1. Schematic of the best model architecture of the CD nodule detector, which is divided into four stages, each containing different convolutional i blocks.

Results for each one of the four validation metrics: log-loss, sensitivity, specificity, and F1 score, are illustrated in Figure 4.2. Specifically, each graph illustrate the average validation scores over the 10 folds during 15 epochs of training. Also, Table 4.3 shows the final validation scores for each metric including the area under the ROC curve.

After optimizing the best model, it was run over 30 CT scans from database A. Specifically, a sliding window with strides(5 x 10 x 10) was developed to extract different CT portions that were classified as ”nodule” or ”non-nodule”. Different binary decision thresholds were defined to obtain a wide range of detection results. To compute the sensitivity metric, all true positive labels were specified with a radiologist-level of agreement of all four radiologists. If the algorithm detected a nodule that had a lower radiologist-level of agreement, it was considered as a false positive. Also, any new detection made by the algorithm, which exceeded a 30% intersection (see Equation (4.1)) with previous detections was not considered as a detection; thus, it could not count as neither a false positive nor a true positive. Figure 4.3 illustrates the resultant FROC curve with eight different

thresholds ranging from 0.1 to 0.9 with 0.1 intervals.

$$Idetection = \frac{V_{new\ detection} \bigcap V_{previous\ detection}}{V_{previous\ detection}} \tag{4.1}$$

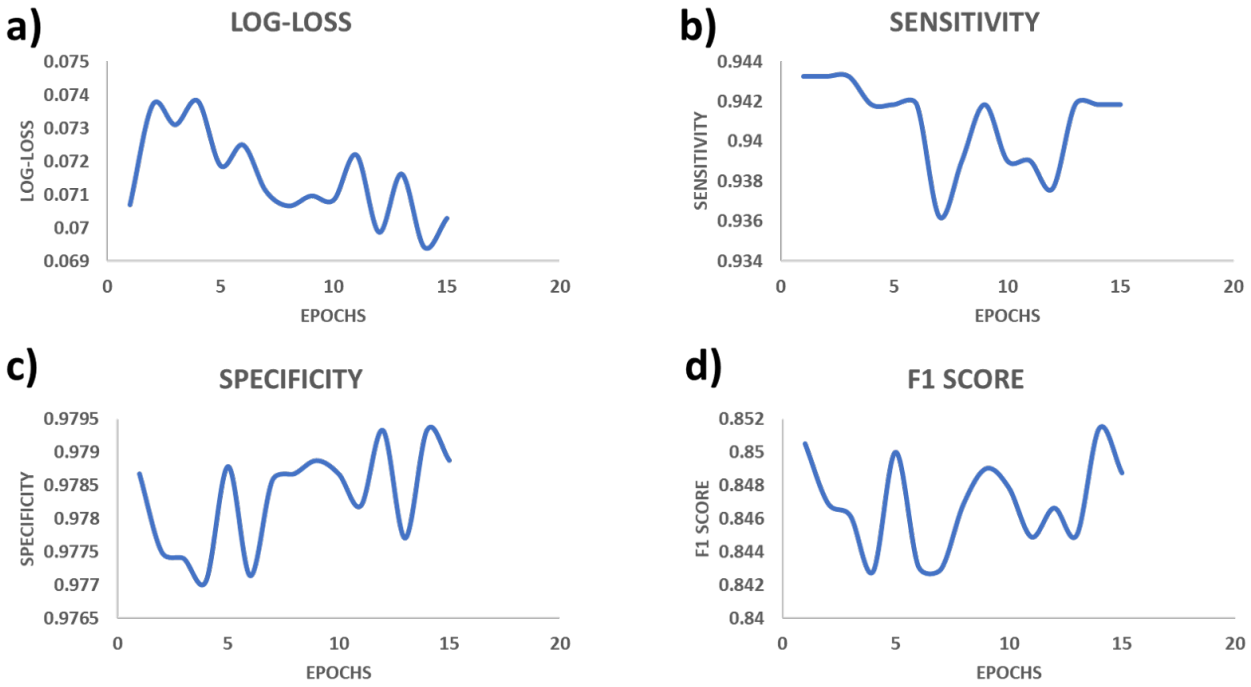

Fig. 4.2. Summary of the four specific target metrics of the candidate nodule detector.

Table 4.3. Final average validation scores of the CD CNN.

METRIC	SCORE
Sensitivity	0.94184
Specificity	0.9788
F1 Score	0.8487
Area under the ROC curve	0.9995
Log-loss	0.0702

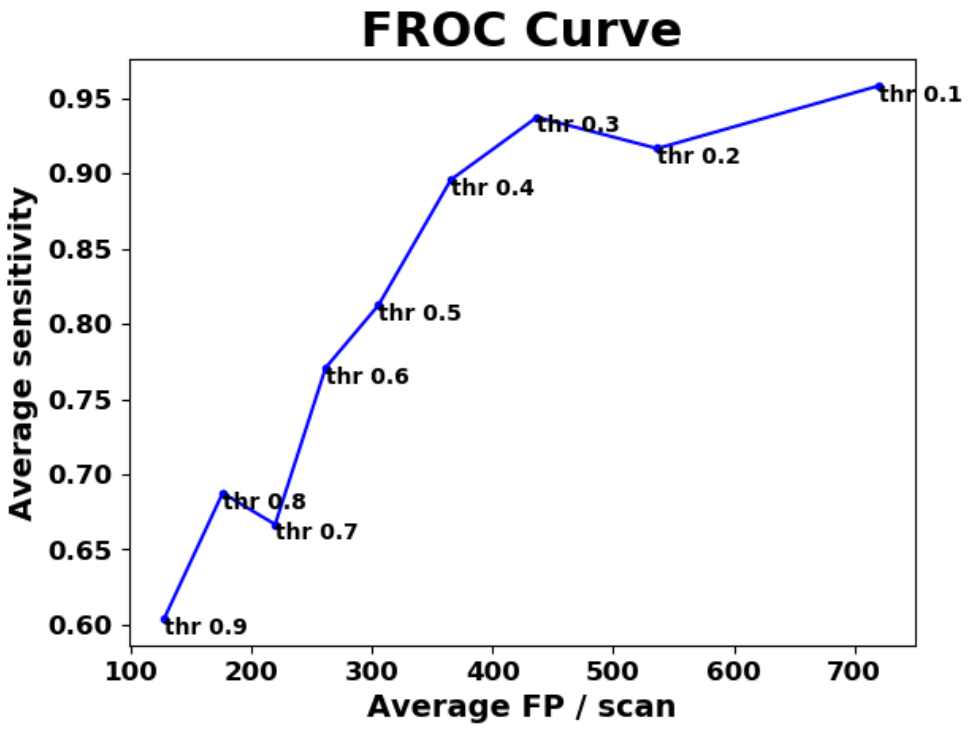

Fig. 4.3. Free-Response ROC Curve of the CD CNN.

The candidate generation CNN was applied to a set of 23 CTs contained in database B2 using a threshold of 0.5. True positives were defined as nodules $\geq$ 3 mm following the criteria posed in the Guidelines for Management of Incidentally Detected Pulmonary Nodules in Adults [18]. Results are shown in 4.4. Due to time constraints, further experiments will be conducted using data bases B1 and B3.

Sensitivity	Average FP / CT
0.8	238.56

Fig. 4.4. Results of the candidate generator CNN applied to database B2.

4.1.2 False positive reduction CNN

To reduce false positives, the FPR CNN was developed using a similar approach. Training data consisted on 13,103 false positives (negative instances), generated by the candidate generator using a threshold of 0.3, and the same 1,273 nodules (positive examples). Also, hyper-parameter optimization was implemented using the TPE algorithm, which was run for 7 trials. Each model configuration was optimized using a 10-fold cross validation technique over one epoch.

Data augmentation was performed over the positive examples using two lossless 3D rotations (90° and 180°) with respect to the Z axis, one nodule rescaling using the nearest neighbors algorithm, and seven random image translations with respect to each possible combination of axes (X, Y, Z, XY, XZ, XYZ) with a maximum translation value of 10 pixels. Thus, a total of 13,637 augmented nodules were used for training and validation. Table 4.5 enlists all hyper-parameter values of the best FPR CNN, while the topology of the network is the same as the CD Nodule CNN represented in Figure 4.1.

Afterwards, the best CNN was trained and validated using a classic 70/30% technique over 30 epochs. The same data was used for training and validating the best model. Figure 4.5 illustrates the four statistical metric results. On the other hand, Table 4.6 shows the final values. The same technique, used for the candidate generation CNN, was implemented to compute the optimal binarization threshold [105]. Afterwards, the FPR model was run over 20 CT patient scans by fixing the CD CNN threshold to 0.1 and setting the optimal threshold to 0.65. Final results are illustrated in Table 4.4 indicating a final sensitivity of 0.25 and an average of 41.8 false positives per scan.

Table 4.4. Validation metrics of the FPR CNN after being run over 20 CT scans.

Sensitivity	Average FP / CT
0.25	**41.8**

Table 4.5. Final hyper-parameter values defining the best model configuration of the FPR CNN after running the TPE algorithm for 10 different models.

HYPER-PARAMETER	VALUE
Learning rate α	0.00095
Decay learning constant	0.01083
m_1 = number of i blocks for stage 1.	1
m_2 = number of i blocks for stage 2.	2
m_3 = number of i blocks for stage 3.	0
m_4 = number of i blocks for stage 4.	0
Number of filters F_i for the i^{th} block in stage ≤ 2.	8 for stage 1 16 for stage 2
Number of filters F_i for the i^{th} block in stage > 2.	16 for stage 3 64 for stage 4
Filter size f_i for the i^{th} block in stage 1.	(1x1)
Filter size f_i for the i^{th} block in stage 2.	(3x3)
Filter size f_i for the i^{th} block in stage 3.	(3x3)
Filter size f_i for the i^{th} block in stage 4.	(1x1)
L2 regularization constants l_i for the i^{th} block in stages one and two.	0.00031 for stage 1 0.00033 for stage 2
L2 regularization constants l_i for the i^{th} block in stages three and four.	0.00008 for stage 3 0.00030 for stage 4
Number of x dense layers	1
Number of units n per dense layer	10
Number of average pooling layers with pool size = $(3x3)$ and valid padding.	1
Number of parameters	742,839

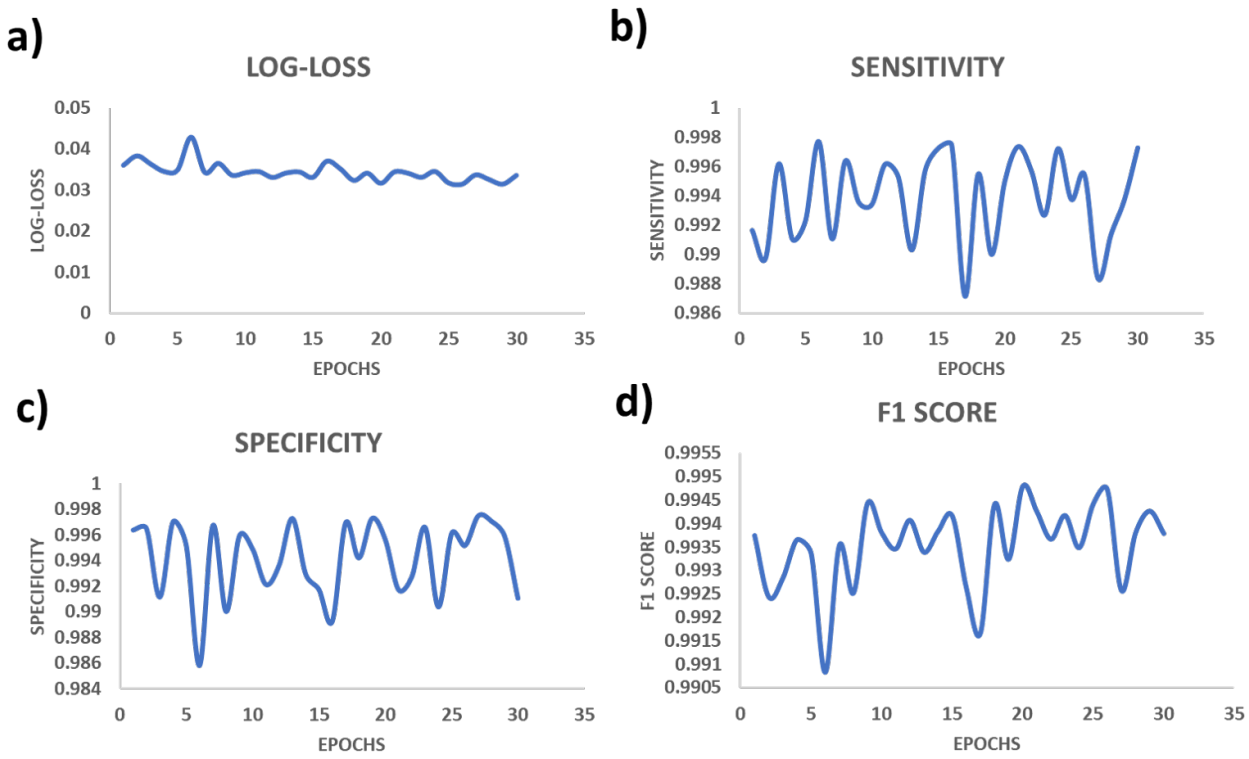

Fig. 4.5. Summary of the four specific target metrics of the FPR CNN.

Table 4.6. Final average validation scores of the FPR CNN.

METRIC	SCORE
Sensitivity	0.9972
Specificity	0.9910
F1 Score	0.9937
Area under the ROC curve	1
Log-loss	0.0336

4.2 Malignancy classifier

Similar to the nodule detector stage, approximately 60 different experiments were performed, where each one of them included variations of the aforementioned specifications that were enlisted in Section 4.1 (excluding the last specification). As stated in Section 3.7, all training and validation data comprised 722 and 213 nodules that were considered as malignant and benign respectively (for details see Section 3.7). After applying both, nine lossless 3D rotations and GANs, the training set was augmented to 6,019 instances: 3,000 and 3,020 benign and malignant examples respectively.

Hyper-parameter optimization was performed using the TPE algorithm, where 15 different model configurations were tried. Each trial implemented a 10-fold cross-validation training, with a total of 5 epochs.Repeating the same procedure, the Adam optimizer and Xavier initialization were implemented during training, with fixed hyper-parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$. Also, the hyper-parameter space is the same as the one illustrated in Figure 4.1. Therefore, the topology schematic is defined in the same way, as the one presented in Figure 4.1. Specifically, Table 4.8 enlists all hyper-parameter values of the best malignancy classifier CNN.

Once the best model was identified, further optimization was performed to improve its classifying capabilities. Specifically, training was done over 30 epochs using the same 10-fold cross-validation scheme, and the two aforementioned data augmentation techniques: GANs and the nine lossless 3D rotations. Validation results for this final optimization are illustrated in Figure 4.6 and Table 4.7.

Table 4.7. Final average validation scores of the malignancy classifier CNN.

METRIC	SCORE
Sensitivity	1
Specificity	1
F1 Score	1
Area under the ROC curve	1
Log-loss	0.008171

Table 4.8. Final hyper-parameter values defining the best model configuration of the malignancy classifier.

HYPER-PARAMETER	VALUE
Learning rate α	0.0003434
Decay learning constant	0.001285
m_1 = number of i blocks for stage 1.	1
m_2 = number of i blocks for stage 2.	1
m_3 = number of i blocks for stage 3.	1
m_4 = number of i blocks for stage 4.	1
Number of filters F_i for the i^{th} block in stage ≤ 2.	16 for stage 1 16 for stage 2
Number of filters F_i for the i^{th} block in stage > 2.	8 for stage 3 16 for stage 4
Filter size f_i for the i^{th} block in stage 1.	(3x3)
Filter size f_i for the i^{th} block in stage 2.	(5x5)
Filter size f_i for the i^{th} block in stage 3.	(3x3)
Filter size f_i for the i^{th} block in stage 4.	(5x5)
L2 regularization constants l_i for the i^{th} block in stages one and two.	0.00058 for stage 1 .00007 for stage 2
L2 regularization constants l_i for the i^{th} block in stages three and four.	0.00012 for stage 3 0.0008 for stage 4
Number of x dense layers	2
Number of units n per dense layer	8
Number of average pooling layers with pool size = $(3x3)$ and valid padding.	1
Number of parameters	328,613

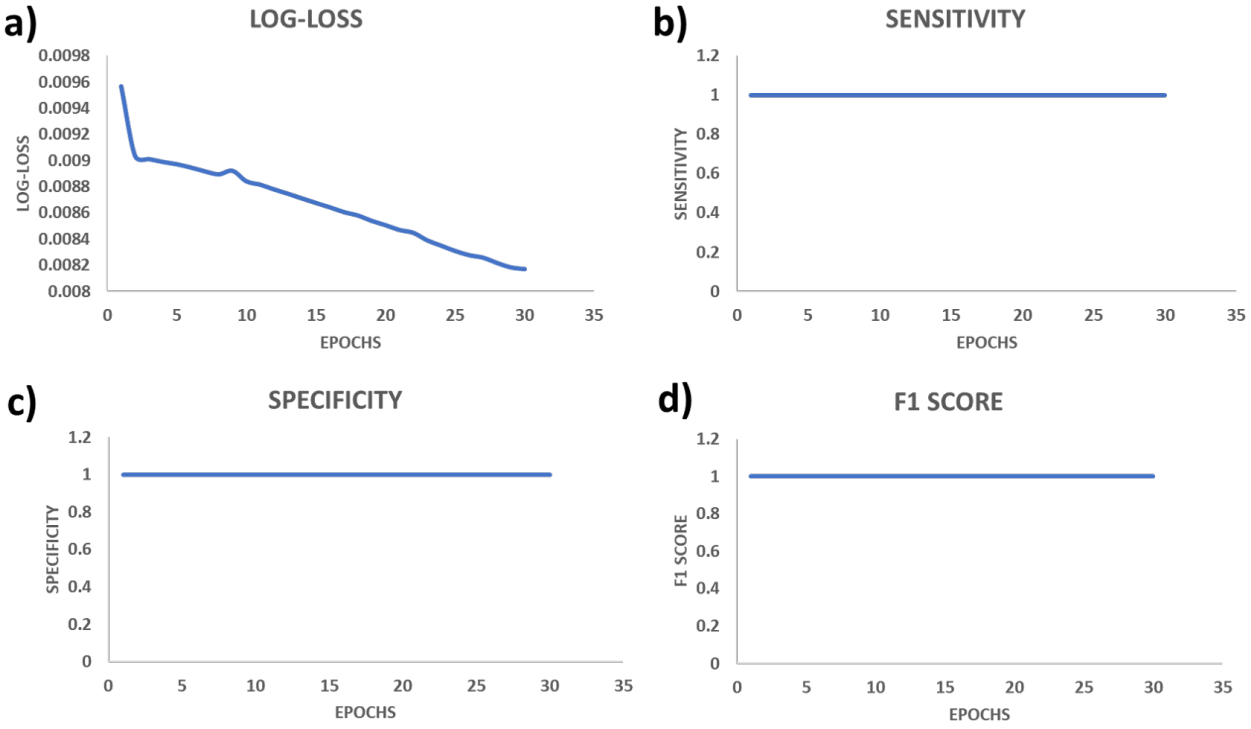

Fig. 4.6. Summary of the four specific target metrics of the malignancy classifier CNN.

4.3 Morphological estimator

4.3.1 Spiculation estimator CNN

All data was comprised by 245 nodules that were considered ”spiculated” and 693 anomalies defined as ”non-spiculated”. By applying both data augmentation techniques, 6,000 instances were set for the whole training data.

For this stage, hyper-parameter optimization was run over 15 models by using the TPE algorithm with a 10-fold cross-validation and a training interval of five epochs. Also, the same network configuration of the previous stages was defined for the spiculation estimator. During each model evaluation of TPE, the same optimization specifications of the malignancy classifier, the CD and FPR CNNs were set (see Sections 4.1.1, 4.1.2, and 4.2). Results of the best hyper-parameter configuration are shown in Table 4.10. Further optimization of the best model was performed using the same settings of the malignancy classifier (see Section 4.2). All training results for the four averaged validation metrics are illustrated in Figure 4.7, where all final scores are depicted in Table 4.9.

Table 4.9. Final average validation scores of the spiculation estimator CNN.

METRIC	SCORE
Sensitivity	0.9965
Specificity	0.9984
F1 Score	0.9966
Area under the ROC curve	0.9999
Log-loss	0.0133

Table 4.10. Final hyper-parameter values defining the best model configuration of the spiculation estimator.

HYPER-PARAMETER	VALUE
Learning rate α	0.0005987
Decay learning constant	0.00374
m_1 = number of i blocks for stage 1.	0
m_2 = number of i blocks for stage 2.	1
m_3 = number of i blocks for stage 3.	1
m_4 = number of i blocks for stage 4.	0
Number of filters F_i for the i^{th} block in stage ≤ 2.	16 for stage 1 16 for stage 2
Number of filters F_i for the i^{th} block in stage > 2.	8 for stage 3 32 for stage 4
Filter size f_i for the i^{th} block in stage 1.	(3x3)
Filter size f_i for the i^{th} block in stage 2.	(5x5)
Filter size f_i for the i^{th} block in stage 3.	(5x5)
Filter size f_i for the i^{th} block in stage 4.	(5x5)
L2 regularization constants l_i for the i^{th} block in stages one and two.	0.00049 for stage 1 0.00009 for stage 2
L2 regularization constants l_i for the i^{th} block in stages three and four.	0.00043 for stage 3 0.00064 for stage 4
Number of x dense layers	0
Number of units n per dense layer	0
Number of average pooling layers with pool size = $(3x3)$ and valid padding.	1
Number of parameters	40,265

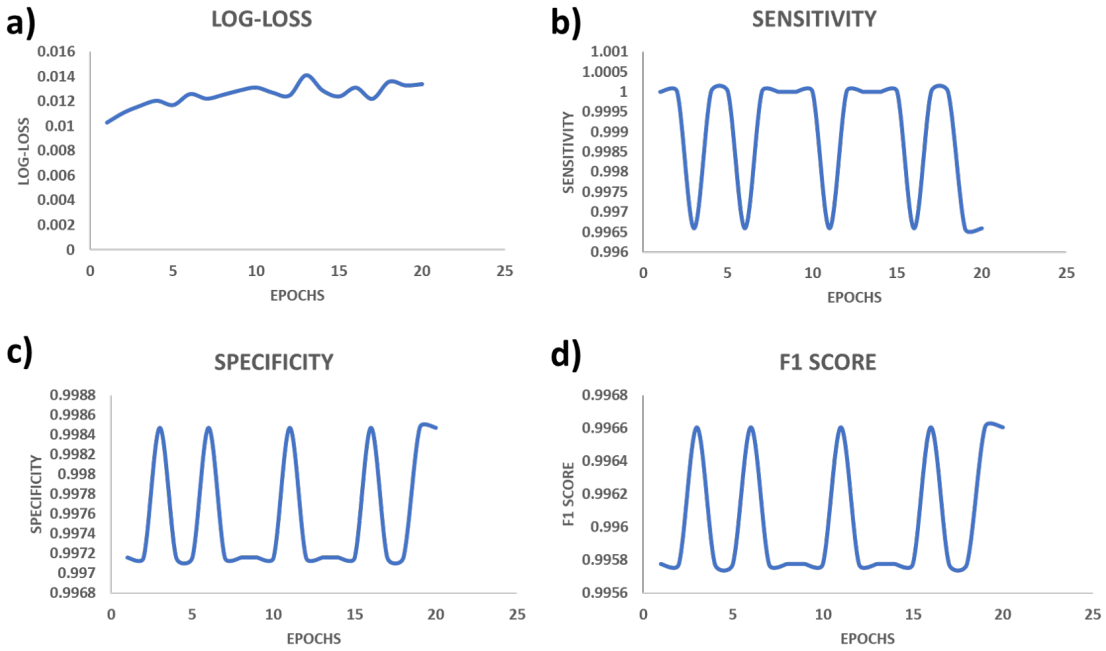

Fig. 4.7. Summary of the four specific target metrics of the spiculation estimator CNN.

4.3.2 Lobulation estimator CNN

All data was comprised by 245 nodules that were considered ”lobulated” and 693 anomalies defined as ”non-lobulated”, giving a total of 6,000 instances for the whole training set, after applying both augmentation techniques, GANs and 9 lossless 3D rotations.

Hyper-parameter optimization was run using the TPE algorithm to evaluate 15 different CNNs. Training of each model was performed using a 10-fold cross-validation over 5 epochs. Network search was performed over the same configuration space and CNN topology specified for the previous stages. During each model evaluation of TPE, the same optimizer and hyper-parameters of the Spiculation Estimator were defined (see Section 4.3.1). Results of the best hyper-parameter configuration are shown in Table 4.12. Training of the best model was performed using the same settings of the spiculation estimator (see Section 4.3.1), but with a training interval of 20 epochs. All training results for the four averaged validation metrics are illustrated in Figure 4.8, where all final scores are depicted in Table 4.11.

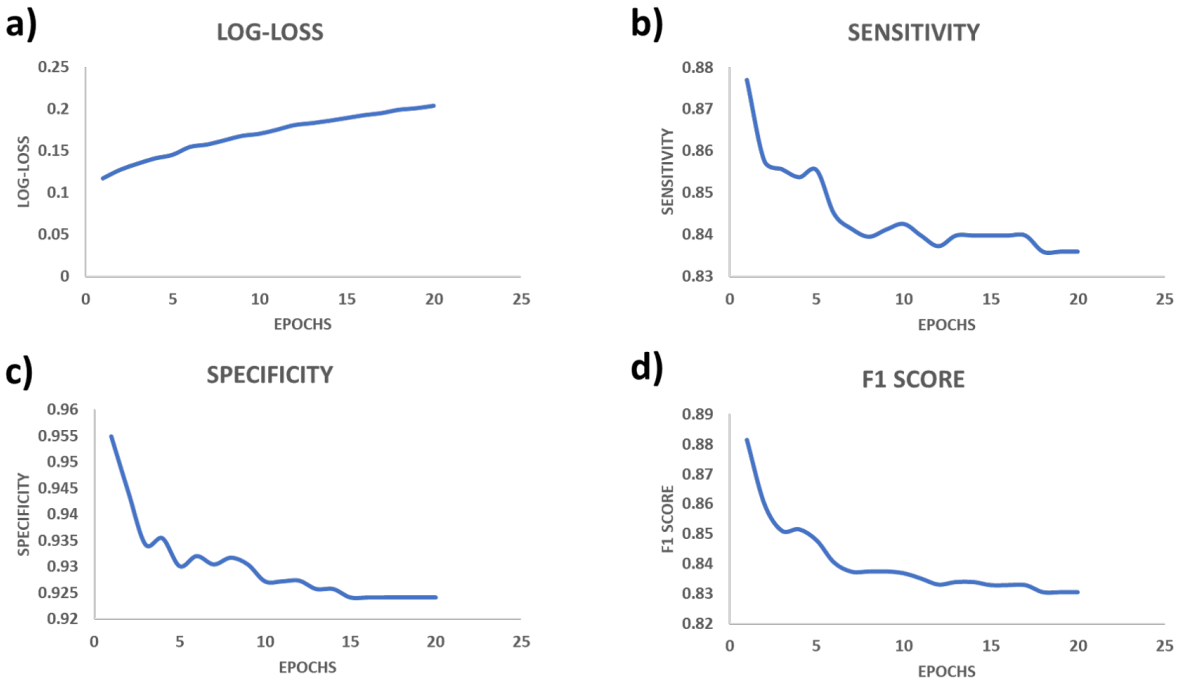

Fig. 4.8. Summary of the four specific target metrics of the lobulation estimator CNN.

Table 4.11. Final average validation scores of the lobulation estimator CNN.

METRIC	SCORE
Sensitivity	0.8358
Specificity	0.9240
F1 Score	0.8306
Area under the ROC curve	0.9831
Log-loss	0.2039

Table 4.12. Final hyper-parameter values defining the best model configuration of the lobulation estimator.

HYPER-PARAMETER	VALUE
Learning rate α	0.00055
Decay learning constant	0.03048
m_1 = number of i blocks for stage 1.	0
m_2 = number of i blocks for stage 2.	1
m_3 = number of i blocks for stage 3.	1
m_4 = number of i blocks for stage 4.	1
Number of filters F_i for the i^{th} block in stage ≤ 2.	32 for stage 1 16 for stage 2
Number of filters F_i for the i^{th} block in stage > 2.	16 for stage 3 16 for stage 4
Filter size f_i for the i^{th} block in stage 1.	(5x5)
Filter size f_i for the i^{th} block in stage 2.	(3x3)
Filter size f_i for the i^{th} block in stage 3.	(3x3)
Filter size f_i for the i^{th} block in stage 4.	(5x5)
L2 regularization constants l_i for the i^{th} block in stages one and two.	0.00096 for stage 1 0.00024 for stage 2
L2 regularization constants l_i for the i^{th} block in stages three and four.	0.00034 for stage 3 0.00034 for stage 4
Number of x dense layers	3
Number of units n per dense layer	14
Number of average pooling layers with pool size = $(3x3)$ and valid padding.	1
Number of parameters	4,227,535

4.4 Cancer estimator

For the final stage, data was built using the output matrices $\{O_{morph_1}, O_{morph_2}, ..., O_{morph_j}\}$ specified in Section 3.9. Specifically, 20 feature vectors $\vec{x_i}$ (described in Equation (3.9)) were obtained after running all previous stages over 20 pathological-confirmed CT scans: 10 benign and 10 malignant cases.

A classical DNN scheme, represented in Figure 4.9 was proposed for performing hyper-parameter optimization using the TPE algorithm; thus, reducing the search space considerably. Similar to the schematic provided in Figure 3.8, the general DNN configuration was set to contain five different blocks, where each block i has a distinct number of m_i layers containing n_i units. To avoid overfitting, L2 regularization constants were tuned at each block, and a dropout layer was set at the end of the DNN. A detailed list of all the hyper-parameters that were tuned are enlisted in Table 4.13.

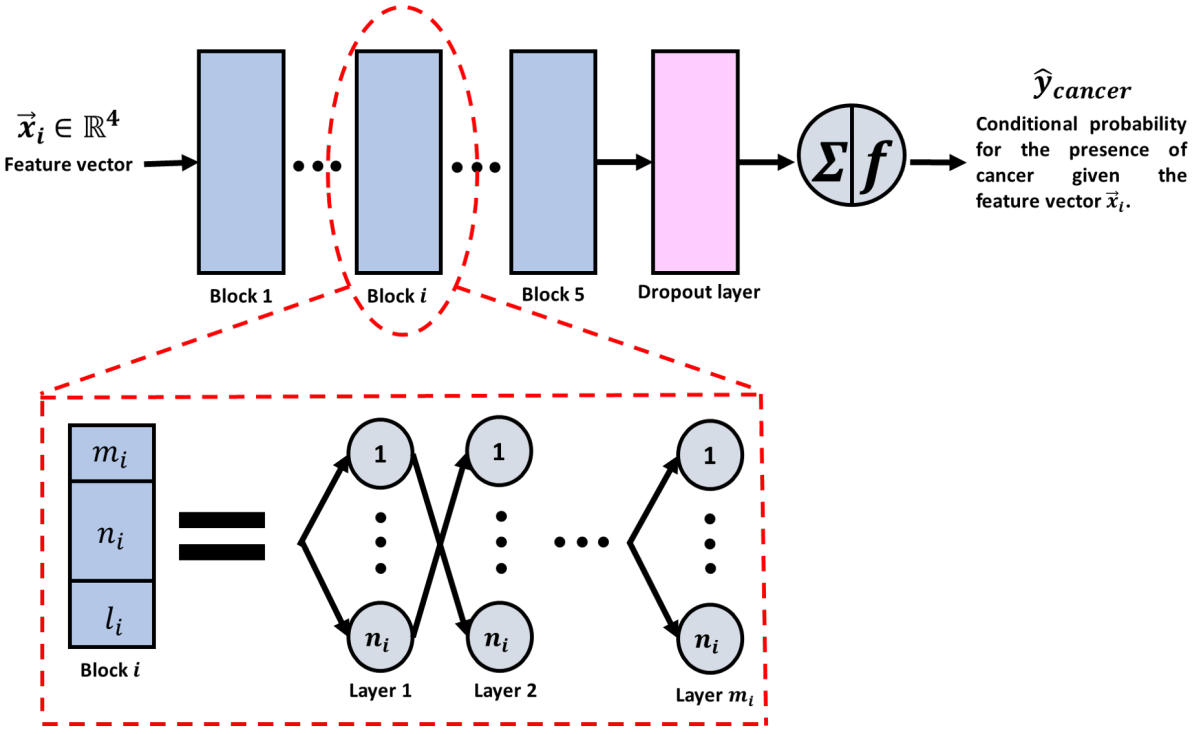

Fig. 4.9. Schematic of the DNN representing the cancer estimator.

Table 4.13. Hyper-parameter space configuration and sampling distributions to search the best set of values using the TPE algorithm for the cancer estimator.

HYPER-PARAMETER	SAMPLING DISTRIBUTION	SEARCH RANGE
Learning rate α	Continuous uniform	[0.0005, 0.001]
Decay learning constant	Continuous uniform	[0.01, 0.05]
Number of m_i dense layers in block i	Discrete uniform	{1, 2, 3, 4, 5}
Number of n_i units in each layer of block i	Discrete uniform	{1, ..., 1000}
L2 regularization constants l_i for all layers in block i	Continuous uniform	[0.0001, 0.01]
Choose if dropout is implemented or not	Discrete uniform	{0, 1}
Dropout rate	Continuous uniform	{0, 0.01}

With these specifications, the TPE algorithm was implemented to evaluate 10 models. Again, the Adam optimizer with hyper-parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$, along with Xavier initialization were implemented for training each model. A 10-fold cross-validation scheme was used at each trial. Results indicating the best set of hyper-parameters of the DNN are defined in Table 4.14.

Once the best model was obtained, it was further trained over 300 epochs using the same 10-fold cross-validation strategy. The same approach was chosen: using the Adam optimizer hyper-parameters and Xavier Initialization. Validation results, over the 300 epochs, are depicted in Figure 4.10. Table 4.15 enlists the final scores obtained for each metric: sensitivity, specificity, log-loss and F1 score.

Table 4.14. Final hyper-parameter values defining the best model configuration of the cancer estimator.

HYPER-PARAMETER	VALUE
Learning rate α	0.00089
Decay learning constant	0.01756
m_0 = number of dense layers for block 0.	4
m_1 = number of dense layers for block 1.	3
m_2 = number of dense layers for block 2.	4
m_3 = number of dense layers for block 3.	2
m_4 = number of dense layers for block 4.	3
n_0 = number of units in each layer of block 0.	571
n_1 = number of units in each layer of block 1.	370
n_2 = number of units in each layer of block 2.	418
n_3 = number of units in each layer of block 3.	849
n_4 = number of units in each layer of block 4.	180
l_0 = L2 regularization constant of all layers in block 0.	0.000679
l_1 = L2 regularization constant of all layers in block 1.	0.002327
l_2 = L2 regularization constant of all layers in block 2.	0.000201
l_3 = L2 regularization constant of all layers in block 3.	0.004646
l_4 = L2 regularization constant of all layers in block 4.	0.003652
Use dropout at the end of each block.	No
Dropout rate.	-
Number of parameters	4,857,030

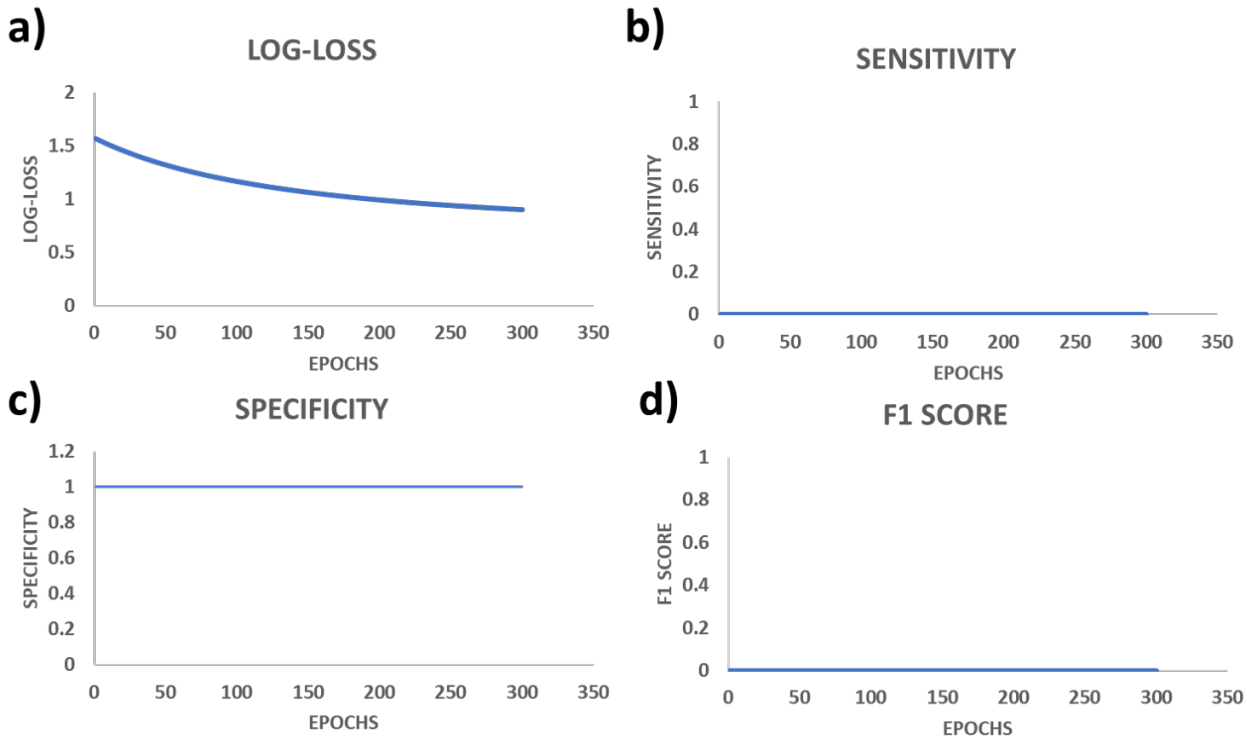

Fig. 4.10. Summary of the four specific target metrics of the cancer estimator DNN.

Table 4.15. Final average validation scores of the cancer estimator CNN.

METRIC	SCORE
Sensitivity	0
Specificity	1
F1 Score	0
Area under the ROC curve	0.5
Log-loss	3.968

4.5 Discussion

Reviewing the results of this dissertation, it was determined that the detection and characterization of nodular findings was a partially successful. Even though it seems that both, sensitivity and specificity metrics were attained, they do not provide an adequate validation scheme for the nodule detection stage. As it has been stated in several studies, the most relevant validation metric for medical imaging detection applications is the FROC curve [69, 72, 79, 81, 83]. A high sensitivity of 94.1% was obtained for the candidate generation CNN with a binarization threshold of 0.3, along with a high number of false positives per scan.

As illustrated in the results, the main problem that was encountered during the research was the development of the false positive reduction stage. Even though more than 30 models were tested using the TPE algorithm, no successful results were obtained to reduce false positive findings without compromising sensitivity. Final experiments showed that the best FPR models were obtained when heavy data augmentation was applied to the data, specifically when random nodular translations were included. A potential hypothesis to explain this phenomenom resides on the positive to negative ratio encountered in a CT. More concretely, there is a huge amount of CT portions not containing nodular findings compared to CT volumes with nodules. Furthermore, depending on the number of strides specified for the sliding detection window, portions of a single nodule may not be detected by the CNN. Thus, augmenting the positive examples with translational variations may help the algorithm to detect the same nodule even if it is not centered exactly within the detection window.

An important remark resides in the probability estimations computed by almost all of the developed FPR models. Most of the highest detection rates per CT slice were obtained outside of the CT scan. The addition of preprocessing algorithms, could reduce undesired CT regions and decrease the number of false positive findings. Although it was not mentioned in the dissertation, a lung segmentation algorithm was built, but due to inconsistencies during the segmentation it was not included in the algorithm workflow. Poor results may also be attributed to the exclusion of repeated annotations of the same nodule provided by all four radiologists from database A. However, it

is possible that the restrictions imposed to define a true positive were too flexible. For example, a study defined as true positives only nodules with sizes greater than 8 mm^3 [83]. Other study removed candidates whose shape was under 33 mm^3 [72]. Thus, it seems that comparing results between frameworks is a major challenge due to different criteria stablished to define true positives and false positives. As a result, many studies may have achieved reasonable results due to these considerations. Other possible cause associated to this undesired behavior, could be related with the sepecified receptive field, an issue that was addressed by Dou *et al.* [79], who specified different receptive fields, and Dai *et al.* who defined a bigger recepetive field of (64 x 64 x 64) pixels compared to the receptive field of (32 x 32 x 32) pixels established in this thesis [69].

Because the best results for the FPR model were attained using test time augmentation (augmenting samples during validation), further analysis and research will be conducted using this methodology. Also, the inclusion of statistical metrics, such as curvedness and Shape Index, for generating nodule candidates will be tested; a strategy that was developed in one of the CD models mentioned by Setio *et al.* [72]. More importantly, best results were reported by studies that used ensemble methods to boost model performance [72, 79]. Although one of the main contributions of this thesis consisted on achieving a similar performance by training fewer models, none of the 100 and 30 architectures trained for both, the CD and FPR CNNs, indicated that this can be achieved. Thus, further research will focus in this machine learning technique, along with decision trees and random forests.

Even though the FPR model did not achieved the desired performance, the morphological and malignancy estimators showed remarkable results. Specifically, the malignancy classifier outperformed all related studies that focused on the same research objective [29, 70, 73]. Two possible reasons to explain this boost in performance are: 1) using GANs to generate more artifcial nodules [62] and 2) working with training data comprised only by non-confirmed histo-pathological malignant nodules. Although these results sound promising, further testing must be done to verify them.

Independent testing of the candidate detection CNN was performed using database B2. Results were better than the ones obtained during validation and training. Different strides used in both, training and independent testing, may explain the differences in performance. Bigger strides were applied when running the candidate generation CNN using database B2. Therefore, with bigger strides the number of false positives was reduced. Moreover, one of the main contributions of this dissertation is based on the morphological estimators, as they provide some degree of interpretability to the radiologists. A justification that is also stated by the research conducted by Shen *et al.*.

Due to the sequential design of the algorithm workflow, many of the poor results obtained in previous steps are reflected in further stages of the algorithm. Specifically, the poor results obtained for the cancer estimator are explained by this architecture design. Because the CD and FPR CNNs fail to detect nodules with success, all the generated false positives induce noise to the training data of the cancer estimator. Although one of the key design choices was to select the maximum values of all the computed joint probabilities, there are some false positives that have greater joint probability compared to the true positives. Thus, some modifications will be made to the algorithm workflow to avoid these issues.

One of the main limitations of this research study was the hardware. Although many of the trained models were inspired on more complex architectures, such as ResNet, Inecption Net, Vgg and Inception-Resnet, their abstraction capabilities cannot be compared to the original ones. This limitation was seen as another contribution of this thesis, to achieve similar results by training models with fewer parameters. It is important to state that similar studies have worked with a lot more resources, allowing them to train more complex models, such as the Inception-Resnet architecture [69, 81].

CHAPTER 5

Conclusions

This dissertation started by describing a major health problem worldwide, the high mortality rates associated to lung cancer. Moreover, the literature review showed the major challenges associated to the disease. Specifically, the NLST study showed a high number of false positives when comparing two technologies, low-dose CT and chest X-ray. Furthermore, different research studies showed several complications related to false positives, such as higher economic costs and a risk increase of morbidity/mortality. Inter-grader variability and subjective assessment of CTs have been associated to inconsistencies in the prognosis procedure, which may result in false positives and negatives.

To help in the CT assessment and reduce the effects of the aforementioned problems, several research studies have focused on developing computer aided-diagnostic tools. Some researchers have worked on different stages of the prognostic procedure, such as lung segmentation, candidate nodule detection, false positive reduction, malignancy estimation of nodular findings, and end-to-end cancer diagnosis models. Because very few researchers have centered their efforts in model interpretability, one of the main research questions addressed in this thesis consisted on determining the best set of morphological nodular characteristics to aid in the detection and characterization

of lung nodules. Also, a second question was defined to determine if the automation of nodular detection would help reduce the number of false positives in lung cancer diagnosis. A third and last question was presented to evaluate nodular detection and characterization performance using a divide and conquer strategy.

Based on the results of the morphological estimators, spiculation and lobulation have showed the highest linear correlation with the likelihood of nodular malignancy, a result that will help with model interpretability. The performance of the FPR detector showed that the reduction of false positives require a higher number of models to train with higher complexity. Moreover, errors encountered in the first stages of the algorithm propagate throughout the pipeline. Thus, a sequential design is not adequate for tackling this problem. Different model configurations need to be explored to avoid these sequential bottlenecks.

We can conclude that the main problem resided on the FPR stage. It is ideal to continue the research to optimize the FPR architecture. As exposed in the discussion, several efforts will focus on test-time data augmentation techniques, ensemble methods and different receptive fields. As stated in the beginning, two main contributions of the dissertation consisted on obtaining reliable results with few computational resources, smaller and fewer CNN models. Although none of the two main contributions were attained, further efforts will focus on obtaining results that align with these two objectives. Also, other strategies will be developed to train large ensmebles of models with bigger architectures. Research efforts will be centered in solving the pipeline bottleneck associated to the sequential nature of the algorithm workflow.

Developing a lung cancer end-to-end detection approach is a very complex and challenging task. Even more, trying to develop a limited amount of models with few trainable parameters poses additional constraints that make the problem even harder to solve. It seems that the best solution consists in developing large ensemble of models, with heavy data augmentation techniques, and smart feature engineering for preprocessing CTs. Moreover, a parallel divide and conquer strategy semi-dependent on previous stages, may alleviate some of the problems that can arise within the

pipeline; thus, computing final predictions that are not heavily affected if one of the stages resulted in poor performance.

Lung cancer has posed many social and economic problems to society. However, this works has proved the value of CAD models to aid in the detection of lung cancer with some degree of accuracy. It is expected that better technologies will be developed in the future, helping radiologists detect lung cancer in early stages to provide immediate healthcare to patients.

Appendices

APPENDIX A

Informed consent forms

ANEXO A: DOCUMENTO DE CONSENTIMIENTO INFORMADO

En caso de tratarse de un menor de edad, o un paciente que se encuentre en estado de incapacidad transitoria o permanente, o que por su situación legal no pueda expedir el consentimiento libremente, la autorización será otorgada por el familiar más cercano en vínculo que le acompañe, o en su caso, por tutor o representante legal.

DATOS PERSONALES

Nombre del paciente:

_____

Matrícula: _____ Edad: ____ Sexo: _____ Teléfono: _____

Domicilio:

_____

Nombre de la persona que recibió la información y el consentimiento:

_____

Paciente ( ) Representante legal ( ) Familiar ( )

Parentesco: _____

REALIZARÁ LOS PROCEDIMIENTOS: Ing. Jonathan Domínguez Aldana

PROCEDIMIENTO: La comprobación histológica por biopsia, la tomografía computarizada del paciente y las anotaciones hechas por el radiólogo experto sobre las características encontradas en la tomografía, serán usados para ser integrados a una base de datos de fácil acceso. Esta base de datos será usada en un protocolo de investigación para alimentar a un algoritmo de inteligencia artificial que ratificará la presencia de cáncer de pulmón.

PROPÓSITO DEL ESTUDIO: Obtener la información clínica pertinente para que sea usada por un algoritmo de inteligencia artificial, que le permita mejorar el diagnóstico temprano de cáncer de pulmón a través de tomografía computarizada.

RIESGOS Y COMPLICACIONES: Ninguno

POR LO TANTO, CON LA ANOTACIÓN VERBAL Y ESCRITA

1. Declaro de forma libre y voluntaria sin existir ninguna presión física o moral sobre mi persona, que he comprendido por las explicaciones que se me han proporcionado, el propósito y los riesgos del procedimiento, aclarando las dudas que he planteado. Así mismo declaro que he leído y comprendo totalmente el consentimiento y los espacios en blanco que han sido llenados antes de firmar.
2. Estoy enterado(a) que en cualquier momento y sin necesidad de dar explicación, puedo revocar el consentimiento que otorgo.

AUTORIZO

QUE SE ME (LE) REALICEN LOS PROCEDIMIENTOS ANTERIORMENTE PLANTEADOS Y AUTORIZO EL MANEJO DE LAS CONTINGENCIAS DERIVADAS DEL ACTO ARRIBA AUTORIZADO

En _____

Lugar y fecha Hora

_____

Nombre y firma de la persona que dio el consentimiento

APPENDIX B

Scripts for each hospital data base

classes_angeles.py

```
import os
import numpy as np
import matplotlib.pyplot as plt
import pydicom


class Patient(object):

    def __init__(self, patient_id, patient_dir):
        self.patient_id = patient_id
        self.patient_dir = patient_dir

        for s in os.listdir(patient_dir):
            if not s.endswith('.dcm'):
                os.rename(patient_dir + '/' + s, patient_dir + '/' + s + '.dcm')

        self.saving_path_lungs = patient_dir + '/pixel_data_lungs'
        self.scan = [pydicom.dcmread(patient_dir + '/' + s) for s in os.listdir(patient_dir) if s.endswith('.dcm')]

        self.scan.sort(key=lambda x: float(abs(x.ImagePositionPatient[2])))
        self.scan_pixels = np.stack([s.pixel_array for s in self.scan])
        self.pixel_lungs = []

        # Create saving path for lungs
        if not os.path.isdir(self.saving_path_lungs):
            os.makedirs(self.saving_path_lungs)

    def anonymize_patient(self):
        print('Anonymizing patient {} data...'.format(self.patient_id))
        # Iterate over all slices and change name for patient id
        for s_name in os.listdir(self.patient_dir):
            # Check all dicom data
            if s_name.endswith('.dcm'):
                s = pydicom.dcmread(self.patient_dir + '/' + s_name)
                # Change name
                s.PatientName = self.patient_id
                # Save anonymized data
                s.save_as(self.patient_dir + '/' + s_name)
        print('New patient name: {}'.format(self.patient_id))

    def show_ct(self, num_fig=1, title='COMPLETE SCAN'):
        fig = plt.figure(num_fig)
        fig.suptitle(title)
        im = plt.imshow(self.scan_pixels[0])
        for s in self.scan_pixels:
            im.set_data(s)
            plt.draw()
            plt.pause(0.001)

    def extract_lungs(self, slice_range):
        self.pixel_lungs = self.scan_pixels[slice_range[0]:slice_range[1], :, :]
        return self.pixel_lungs

    def show_lungs(self, num_fig=1, title='LUNG SCAN'):

        if self.pixel_lungs is []:
            print('No extracted lungs!')
            return None
        else:
            fig = plt.figure(num_fig)
            fig.suptitle(title)
            im = plt.imshow(self.pixel_lungs[0])
            for s in self.pixel_lungs:
                im.set_data(s)
                plt.draw()
                plt.pause(0.001)

    def save_extracted_lungs(self, slice_range, graphics=False):

        # Extract lungs
        print('Extracting lungs')
        self.pixel_lungs = self.extract_lungs(slice_range=slice_range)

        # Show complete scan and extracted lungs
        if graphics:
            self.show_ct(num_fig=1)
            self.show_lungs(num_fig=2)

        # Create temporary scan
        temp_scan = self.scan

        # Save results
        print('Saving scan...')
        for i in range(slice_range[0], slice_range[1]):
            # Overwrite pixel data
```

```
            temp_scan[i].PixelData = self.pixel_lungs[i - slice_range[0]].tostring()
            temp_scan[i].save_as(self.saving_path_lungs + '/' + str(i) + '.dcm')
        print('Scan was saved!')

    def load_extracted_lungs(self):

        if self.pixel_lungs is []:

            if len(os.listdir(self.saving_path_lungs)) is 0:
                print('There are no extracted lungs saved!')
                return None

            # Load results
            print('Loading scan...')
            loaded_scan = [pydicom.dcmread(self.saving_path_lungs + '/' + s) for s in
                           os.listdir(self.saving_path_lungs)]
            loaded_scan.sort(key=lambda x: float(abs(x.SliceLocation)))
            self.pixel_lungs = np.stack([s.pixel_array for s in loaded_scan])
            print('Scan was loaded!')

        return self.pixel_lungs
```

classes_h.py

```
import os
import pydicom
import pandas as pd
import numpy as np
from matplotlib import rc
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import scipy.misc
import tensorflow as tf
from classes_lidc import Extractor, PreProcessor
import math

patient_range = (8, 8)


class Patient(object):

    def __init__(self, patient_id, patient_dir):

        self.patient_id = patient_id
        self.patient_dir = patient_dir

        p = PreProcessor()

        if not os.path.isdir(self.patient_dir):
            print('Creating patient folder')
            os.makedirs(self.patient_dir)

        try:
            self.torax_scan = pydicom.dcmread(self.patient_dir + '/' + self.patient_id + '.dcm')

            self.torax_pixels = self.torax_scan.pixel_array
            self.torax_pixels = np.flip(self.torax_pixels, 0)
            self.torax_pixels = np.flip(self.torax_pixels, 2)
            self.torax_pixels, self.resize_factor = p.resampleData(self.torax_pixels, self.torax_scan)
            self.torax_pixels = p.preProcess(self.torax_pixels, self.torax_scan)

            try:
                self.nodule_ids = self.get_nodule_ids()
                self.true_labels = self.get_true_labels()
            except Exception:
                print('No labels have been defined')

        except RuntimeError:
            raise RuntimeError(
                'No Mango file has been saved for patient or CSV file is corrupted{}.'.format(self.patient_id))

    def anonymize_patient(self):

        print('Anonymizing patient {} data...'.format(self.patient_id))
        self.torax_scan.PatientName = self.patient_id
        self.torax_scan.save_as(self.patient_dir + '/' + self.patient_id + '.dcm')

    def get_scan_pixels(self):
        return self.torax_pixels

    def get_scan(self):
        return self.torax_scan

    def show_ct(self, num_fig=1, title='SCAN', speed=0.001):

        fig = plt.figure(num_fig)
        im = plt.imshow(self.torax_pixels[0], cmap='bone')
        for i, s in enumerate(self.torax_pixels):
            fig.suptitle(self.patient_id + ' ' + title)
            im.set_data(s)
            plt.draw()
            plt.pause(speed)
        fig.clear()

    def get_nodule_ids(self):

        df = pd.read_csv(self.patient_dir + '/stats.csv')
        nodule_ids = df['Type'].tolist()
        nodule_ids = [int(nodule_id.split('_')[1]) for nodule_id in nodule_ids]
        nodule_ids = list(dict.fromkeys(nodule_ids))

        return nodule_ids

    def get_true_labels(self, target_size=(32, 32, 32)):

        """
            params:
                - None

            :return:
                - Returns a list of arrays. Where each array represents coordinates
                of a unique nodule in the scan (no repetitions of the same nodule).

                Each array has the following structure:
                array = [minX, maxX, minY, maxY, minZ, maxZ]
        """

        true_labels = []
```

```
        df = pd.read_csv(self.patient_dir + '/stats.csv')

        # Get lists of coordinates
        x = df['X'].to_list()
        y = df['Y'].to_list()
        z = df['Z'].to_list()

        # Get list of nodule IDs without repetitions
        nodule_ids = self.nodule_ids

        # Create interval indexing for saving coordinates (goes in multiples of 3)
        start = 0
        end = 3

        # Iterate over all nodules and save their coordinates in a dictionary.
        for i in range(len(nodule_ids)):
            x_coords = x[start:end]
            minX = min(x_coords)
            maxX = max(x_coords) + (max(x_coords) - minX)

            y_coords = y[start:end]
            minY = min(y_coords)
            maxY = max(y_coords) + (max(y_coords) - minY)

            z_coords = z[start:end]
            minZ = min(z_coords)
            maxZ = max(z_coords) + (max(z_coords) - minZ)

            # Get dimensions
            dim_x = maxX - minX
            dim_y = maxY - minY
            dim_z = maxZ - minZ

            # Get deltas w.r.t target shape
            dx = target_size[0] - dim_x
            dy = target_size[1] - dim_y
            dz = target_size[2] - dim_z

            # If the scan was resampled, then transform cube coords to
            # resampled scan
            if self.resize_factor is not None:
                minX = math.floor(minX * self.resize_factor[1])
                maxX = math.ceil(maxX * self.resize_factor[1])
                minY = math.floor(minY * self.resize_factor[2])
                maxY = math.ceil(maxY * self.resize_factor[2])
                minZ = math.floor(minZ * self.resize_factor[0])
                maxZ = math.ceil(maxZ * self.resize_factor[0])

            # Assign maximums and minimums
            if target_size is not None:
                # Adjusting each coordinate to match target size
                minX = minX - math.floor(dx / 2)
                maxX = maxX + math.ceil(dx / 2)
                minY = minY - math.floor(dy / 2)
                maxY = maxY + math.ceil(dy / 2)
                minZ = minZ - math.floor(dz / 2)
                maxZ = maxZ + math.ceil(dz / 2)

            nodule_coords = [minX, maxX, minY, maxY, minZ, maxZ]
            true_labels.append(nodule_coords)

            # Update indices in multiples of 3
            start += 3
            end += 3

        return true_labels

    def extract_nodule(self, num_nodule, target_shape=(32, 32, 32)):

        if num_nodule not in self.nodule_ids:
            print('That nodule does not exist')
            return None
        else:
            minX = self.true_labels[num_nodule - 1][0]
            maxX = self.true_labels[num_nodule - 1][1]

            minY = self.true_labels[num_nodule - 1][2]
            maxY = self.true_labels[num_nodule - 1][3]

            minZ = self.true_labels[num_nodule - 1][4]
            maxZ = self.true_labels[num_nodule - 1][5]

            # Adjust to target shape
            if target_shape is not None:
                delta = int(abs(target_shape[0] - (maxX - minX)) / 2)
                minX = minX - delta
                maxX = maxX + delta

                delta = int(abs(target_shape[1] - (maxY - minY)) / 2)
                minY = minY - delta
                maxY = maxY + delta

                delta = int(abs(target_shape[2] - (maxZ - minZ)) / 2)
                minZ = minZ - delta
                maxZ = maxZ + delta
```

```
            return self.torax_pixels[minZ:maxZ, minY:maxY, minX:maxX]


class Detector(object):

    def __init__(self):
        self.no_attribute = None

    @staticmethod
    def augment(data):
        # Create data transformer to augment data
        data_transformer = DataTransformer()
        augmented = np.empty((4,) + data.shape)
        augmented[0] = data

        # Rotate in X axis
        rot90_X = data_transformer.rot_tf(data, angle=90, axis=0)
        augmented[1] = rot90_X
        # Rotate in Y axis
        rot90_Y = data_transformer.rot_tf(data, angle=90, axis=1)
        augmented[2] = rot90_Y
        # Rotate in Z axis
        rot90_Z = data_transformer.rot_tf(data, angle=90, axis=2)
        augmented[3] = rot90_Z

        return augmented

    @staticmethod
    def is_false_positive(detection_cube, true_labels,
                          threshold=0.95):
        """
            Returns if the 3D detection coincides with any of the true
            labels in the scan
        """
        # Create the extractor
        e = Extractor()

        # Compute IoU for each possible ground truth label
        for true_label in true_labels:
            iou = e.IoU_Volume(true_label, detection_cube)
            if iou > threshold:
                print('RESULT IoU = {0} \n'.format(iou))
                # If there is a big intersection then it is a true pos
                return False

        # If no intersection was detected with the ground truth labels
        # then it is a false positive
        return True

    @staticmethod
    def update_true_labels(detection_cube, true_labels, z_coords,
                           threshold=0.95):

        """
            Returns the updated true labels
        """
        # Create the extractor
        e = Extractor()

        # Compute IoU for each possible ground truth label
        for i, true_label in enumerate(true_labels):
            iou = e.IoU_Volume(true_label, detection_cube)
            if iou > threshold:
                # If there is a big intersection then it is a true pos, remove true label
                true_labels.pop(i)
                z_coords.pop(i)
                return true_labels

    @staticmethod
    def get_z_coords_labels(true_labels):
        """
            Extracts all the Z coordinates from all nodules present
            in the scan, this is done to plot the contours of the nodule
            during the sliding window process
        """
        z_coords = []
        for label in true_labels:
            minZ = label[4]
            maxZ = label[5]
            z_Range = [minZ, maxZ]
            z_coords.append(z_Range)

        return z_coords

    @staticmethod
    def detect_nodules_in_slice(slice_coord, z_coords, true_labels):
        """
            Returns arrays of X and Y coords of nodules that are in current
            slice. This is used to plot contours of those nodules.
        """
        coordsXY = []
        for i, z_coord in enumerate(z_coords):
            minZ = z_coord[0]
            maxZ = z_coord[1]
            # If slice coord is between the Z range, nodule is there, therefore
            # extract X and Y coords
            if minZ <= slice_coord <= maxZ:
```

```
            minX = true_labels[i][0]
            maxX = true_labels[i][1]
            minY = true_labels[i][2]
            maxY = true_labels[i][3]
            coordXY = [minX, maxX, minY, maxY]
            coordsXY.append(coordXY)

        return coordsXY

    @staticmethod
    def is_new_detection(detection_cube, detected_nodules, threshold=0.5):
        """
            Returns a bool that specifies if the detected nodule is a new
            detection or it was already detected
        """
        e = Extractor()

        # Iterate over the coordinates of the detected nodules
        for detected in detected_nodules:
            # Get the iou
            iou = e.IoU_Volume(detected, detection_cube)

            # If IoU is greater than threshold
            if iou > threshold:
                # No new detection
                return False
        # New detection
        return True

    def slide_window_3d(self, model, patient, graph,
                        saving_path_slides,
                        saving_path_fp,
                        data_dims=None,
                        filter_size=(60, 60, 60),
                        stride=(10, 10, 10),
                        detection_threshold=0.5,
                        save_slides=False,
                        save_fp=False,
                        graphics_show=False,
                        augment_data=False):
        """

        :rtype: object
        """
        true_pos_detections = []
        false_pos_detections = []

        # Get resampled scan pixels
        scan_pixels = patient.get_scan_pixels()

        # Get true labels
        true_labels = patient.get_true_labels()
        true_labels_temp = patient.get_true_labels()
        num_nodules = len(true_labels)

        # Get all z coordinates of all nodules in scan. Used for plotting
        z_coords = self.get_z_coords_labels(true_labels)
        z_coords_temp = self.get_z_coords_labels(true_labels)

        # Initialize number of true positives and false positives
        true_pos = 0
        false_pos = 0

        # Initialize a list to keep track of the detected nodules
        # this list contains the coordinates of the detected nodules
        detected_nodules = []

        # Get the number of steps to iterate in each dimension of the scan
        stepsX = (scan_pixels.shape[0] - filter_size[0])
        stepsY = (scan_pixels.shape[1] - filter_size[1])
        stepsZ = (scan_pixels.shape[2] - filter_size[2])
        stepsX = int(stepsX)
        stepsY = int(stepsY)
        stepsZ = int(stepsZ)

        # Initialize the matrix of results
        results = np.empty((stepsX, stepsY, stepsZ))

        # Iterate over the 3 dimensions
        print('Sliding the window over the patient scan...')

        # Initialize number of detections
        num_detections = 0

        # Change font size
        rc('font', size=4)

        # Check saving folders
        if save_slides:
            if not os.path.isdir(saving_path_slides):
                os.mkdir(saving_path_slides)

        if save_fp:
            if not os.path.isdir(saving_path_fp):
                os.makedirs(saving_path_fp)

        # Iterate over the scan
```

```
        for i in range(0, stepsX, stride[0]):

            # Initialize max prediction
            max_detection_value = 0
            max_detection_loc = [0, 0]

            # Plot the middle slice of the 3D window
            first_slice = i
            middle_slice = i + int((filter_size[0]) / 2)
            last_slice = i + filter_size[0]

            # Create all graphics on image of 3D scan
            if save_slides or graphics_show:
                fig, axarr = plt.subplots(3, 2)
                plt.subplots_adjust(top=0.92, bottom=0.08,
                                    left=0.10, right=0.95,
                                    hspace=0.35, wspace=0.35)
                fig.suptitle((' Num of detections: {}'.format(num_detections))
                             + (' Max prediction: {}'.format(max_detection_value))
                             + (' Num FP: {}'.format(false_pos))
                             + (' Num TP: {}'.format(true_pos))
                             + (' Num of nodules: {}'.format(num_nodules)))

                first_img = scan_pixels[first_slice, :, :]
                axarr[0][0].title.set_text('First Slice {}'.format(first_slice))
                axarr[0][0].set_xlabel('X')
                axarr[0][0].set_ylabel('Y')
                axarr[0][0].imshow(first_img)
                first_ax_img = axarr[0][1].imshow(first_img)

                middle_img = scan_pixels[middle_slice, :, :]
                axarr[1][0].title.set_text('Middle Slice {}'.format(middle_slice))
                axarr[1][0].set_xlabel('X')
                axarr[1][0].set_ylabel('Y')
                axarr[1][0].imshow(middle_img)
                middle_ax_img = axarr[1][1].imshow(middle_img)

                last_img = scan_pixels[last_slice, :, :]
                axarr[2][0].title.set_text('Last Slice {}'.format(last_slice))
                axarr[2][0].set_xlabel('X')
                axarr[2][0].set_ylabel('Y')
                axarr[2][0].imshow(last_img)
                last_ax_img = axarr[2][1].imshow(last_img)

                # Initialize rectangles for max predition
                rectangle_max_1 = patches.Rectangle((0, 0),
                                                    filter_size[1],
                                                    filter_size[2],
                                                    linewidth=1,
                                                    edgecolor='y',
                                                    facecolor='none')
                rectangle_max_2 = patches.Rectangle((0, 0),
                                                    filter_size[1],
                                                    filter_size[2],
                                                    linewidth=1,
                                                    edgecolor='y',
                                                    facecolor='none')
                rectangle_max_3 = patches.Rectangle((0, 0),
                                                    filter_size[1],
                                                    filter_size[2],
                                                    linewidth=1,
                                                    edgecolor='y',
                                                    facecolor='none')
                axarr[0][0].add_patch(rectangle_max_1)
                axarr[1][0].add_patch(rectangle_max_2)
                axarr[2][0].add_patch(rectangle_max_3)

                # If a nodule is in either the first, middle or last slice
                # add rectangle
                first_coords = self.detect_nodules_in_slice(first_slice,
                                                            z_coords,
                                                            true_labels)
                middle_coords = self.detect_nodules_in_slice(middle_slice,
                                                             z_coords,
                                                             true_labels)
                last_coords = self.detect_nodules_in_slice(last_slice,
                                                           z_coords,
                                                           true_labels)
                # Add rectangles of nodule in first slice
                for coords in first_coords:
                    minX = coords[0]
                    minY = coords[2]
                    width = coords[1] - minX
                    height = coords[3] - minY
                    rectangle_first = patches.Rectangle((minX, minY),
                                                        width,
                                                        height,
                                                        linewidth=1,
                                                        edgecolor='c',
                                                        facecolor='none')
                    axarr[0][0].add_patch(rectangle_first)

                # Add rectangles of nodule in middle slice
                for coords in middle_coords:
                    minX = coords[0]
                    minY = coords[2]
                    width = coords[1] - minX
```

```
                height = coords[3] - minY
                rectangle_middle = patches.Rectangle((minX, minY),
                                                     width,
                                                     height,
                                                     linewidth=1,
                                                     edgecolor='c',
                                                     facecolor='none')
                axarr[1][0].add_patch(rectangle_middle)

            # Add rectangles of nodule in last slice
            for coords in last_coords:
                minX = coords[0]
                minY = coords[2]
                width = coords[1] - minX
                height = coords[3] - minY
                rectangle_last = patches.Rectangle((minX, minY),
                                                   width,
                                                   height,
                                                   linewidth=1,
                                                   edgecolor='c',
                                                   facecolor='none')
                axarr[2][0].add_patch(rectangle_last)

        # Iterate over current slice
        for j in range(0, stepsY, stride[1]):
            for k in range(0, stepsZ, stride[2]):

                # Extract current piece of scan to be analyzed by the CNN
                extracted = scan_pixels[i:i + filter_size[0], j:j + filter_size[1], k:k + filter_size[2]]

                # Ensure that the extracted piece of the scan corresponds
                # to the input dimensions of the CNN
                if data_dims != filter_size[0]:
                    resizeFactor = data_dims / filter_size
                    extracted = scipy.ndimage.interpolation.zoom(extracted,
                                                                 resizeFactor,
                                                                 mode='nearest')
                # Augment data if wanted
                if augment_data:
                    augmented = self.augment(extracted)
                    extracted_reshaped = augmented.reshape(augmented.shape + (1,))
                else:
                    extracted_reshaped = extracted.reshape((1,) + extracted.shape + (1,))

                # Predict  and initialize TensorFlow graph
                with graph.as_default():
                    results[i, j, k] = np.average(model.predict(extracted_reshaped))

                # Save max value
                if results[i, j, k] > max_detection_value:
                    # Save the max prediction
                    max_detection_value = results[i, j, k]
                    max_detection_loc[0] = i
                    max_detection_loc[1] = j

                    # Remove the rectangle with previous max detection
                    rectangle_max_1.remove()
                    rectangle_max_2.remove()
                    rectangle_max_3.remove()

                    # Add the rectangle in position of max prediction
                    rectangle_max_1 = patches.Rectangle((k, j),
                                                        filter_size[1],
                                                        filter_size[2],
                                                        linewidth=1,
                                                        edgecolor='y',
                                                        facecolor='none')
                    rectangle_max_2 = patches.Rectangle((k, j),
                                                        filter_size[1],
                                                        filter_size[2],
                                                        linewidth=1,
                                                        edgecolor='y',
                                                        facecolor='none')
                    rectangle_max_3 = patches.Rectangle((k, j),
                                                        filter_size[1],
                                                        filter_size[2],
                                                        linewidth=1,
                                                        edgecolor='y',
                                                        facecolor='none')
                    axarr[0][0].add_patch(rectangle_max_1)
                    axarr[1][0].add_patch(rectangle_max_2)
                    axarr[2][0].add_patch(rectangle_max_3)

                # Create 3 rectangles for correct detection
                rectangle1 = patches.Rectangle((k, j),
                                               filter_size[1],
                                               filter_size[2],
                                               linewidth=1,
                                               edgecolor='r',
                                               facecolor='none')
                rectangle2 = patches.Rectangle((k, j),
                                               filter_size[1],
                                               filter_size[2],
                                               linewidth=1,
                                               edgecolor='r',
                                               facecolor='none')
                rectangle3 = patches.Rectangle((k, j),
```

```
                                filter_size[1],
                                filter_size[2],
                                linewidth=1,
                                edgecolor='r',
                                facecolor='none')

            # Check if prediction is considered nodule or not
            if results[i, j, k] > detection_threshold:
                # Get bounding cube of detection
                minX = k
                maxX = k + filter_size[2]
                minY = j
                maxY = j + filter_size[1]
                minZ = i
                maxZ = i + filter_size[0]
                detection_cube = np.array([minX, maxX,
                                minY, maxY,
                                minZ, maxZ])

                # Create rectangles of detection and sliding window
                rectangle_detected_1 = patches.Rectangle((k, j),
                                                filter_size[1],
                                                filter_size[2],
                                                linewidth=1,
                                                edgecolor='violet',
                                                facecolor='none')
                rectangle_detected_2 = patches.Rectangle((k, j),
                                                filter_size[1],
                                                filter_size[2],
                                                linewidth=1,
                                                edgecolor='violet',
                                                facecolor='none')
                rectangle_detected_3 = patches.Rectangle((k, j),
                                                filter_size[1],
                                                filter_size[2],
                                                linewidth=1,
                                                edgecolor='violet',
                                                facecolor='none')

                # If it is a new nodule, add it to the number of detections
                # but only if it is a new detection
                if self.is_new_detection(detection_cube, detected_nodules, threshold=0.2):
                    # Append new detection
                    detected_nodules.append(detection_cube)

                    # Add new detection
                    num_detections += 1

                    # Add rectangles of detection
                    axarr[0][0].add_patch(rectangle_detected_1)
                    axarr[0][1].title.set_text('Detected - Prediction : {}'.format(results[i, j, k]))
                    axarr[1][0].add_patch(rectangle_detected_2)
                    axarr[1][1].title.set_text('Detected - Prediction : {}'.format(results[i, j, k]))
                    axarr[2][0].add_patch(rectangle_detected_3)
                    axarr[2][1].title.set_text('Detected - Prediction : {}'.format(results[i, j, k]))

                    # Compute if the new detection is a false positive
                    if self.is_false_positive(detection_cube, true_labels_temp, threshold=0.5):

                        # Check if it is not a true positive that was already detected
                        if self.is_new_tp(detection_cube, true_pos_detections, threshold=0.1):
                            false_pos += 1
                            false_pos_detections.append(detection_cube)

                            if save_fp:
                                print('Saving FP {0}'.format(false_pos))
                                file_name = saving_path_fp + '/FP {0}'.format(false_pos)
                                np.save(file_name, extracted_reshaped)

                    # It is a true positive
                    else:
                        # Check if it was not detected before
                        if self.is_new_detection(detection_cube, true_pos_detections):
                            true_labels_temp = self.update_true_labels(detection_cube, true_labels_temp, z_coords_temp)
                            true_pos += 1
                            true_pos_detections.append(detection_cube)

                axarr[0][0].add_patch(rectangle1)
                axarr[1][0].add_patch(rectangle2)
                axarr[2][0].add_patch(rectangle3)

            else:
                # If not nodule only add rectangle of sliding window
                axarr[0][0].add_patch(rectangle1)
                axarr[0][1].title.set_text('Not Detected - Prediction : {}'.format(results[i, j, k]))
                axarr[1][0].add_patch(rectangle2)
                axarr[1][1].title.set_text('Not Detected - Prediction : {}'.format(results[i, j, k]))
                axarr[2][0].add_patch(rectangle3)
                axarr[2][1].title.set_text('Not Detected - Prediction : {}'.format(results[i, j, k]))

            # Update figures
            fig.suptitle((' Num of detections: {}'.format(num_detections))
                         + ('  -  Max prediction: {}'.format(round(max_detection_value, 3)))
                         + ('  -  FP: {}'.format(false_pos))
                         + ('  -  FN: {}'.format(num_nodules - true_pos))
                         + ('  -  TP: {}'.format(true_pos))
                         + ('  -  Num of nodules: {}'.format(num_nodules))
```

```
                                )
                    axarr[0][0].title.set_text(
                        'First Slice {} - Location of window: x={} y={}'.format(first_slice, j, k))
                    first_ax_img.set_data(extracted[0, :, :])
                    axarr[1][0].title.set_text(
                        'Middle Slice {} - Location of window: x={} y={}'.format(middle_slice, j, k))
                    middle_ax_img.set_data(extracted[int(extracted.shape[0] / 2), :, :])
                    axarr[2][0].title.set_text(
                        'Last Slice {} - Location of window: x={} y={}'.format(last_slice, j, k))
                    last_ax_img.set_data(extracted[extracted.shape[0] - 1, :, :])

                    # Show result
                    if graphics_show:
                        plt.draw()
                        plt.pause(0.00001)

                    # Remove sliding rectangles
                    rectangle1.remove()
                    rectangle2.remove()
                    rectangle3.remove()
                    print('\rLocation of window: {} {} {}'.format(i, j, k), end='\r', flush=True)

                print('\nSaving figure...')
                save_name = 'Result {}.png'.format(str(i))
                fig.savefig(saving_path_slides + '/' + save_name, dpi=300)
                print('Figure saved...')
                plt.close()

        results_file = saving_path_slides + '/general_results.txt'
        f = open(results_file, 'w')
        f.write('METRICS\n')
        f.write('   FP: {}\n'.format(str(false_pos)))
        f.write('   FN: {}\n'.format(str(num_nodules - true_pos)))
        f.write('   TP: {}\n'.format(str(true_pos)))
        f.write('   Total nodules: {}\n'.format(str(num_nodules)))
        f.close()

        return results, true_pos_detections, false_pos_detections, true_labels

class DataTransformer(object):

    def __init__(self):
        self = self

    @staticmethod
    def central_scale(data, scale=0.75):
        """
            Input: numpy array of dims = (*dims, num_channels)
            Output: if 2D rescaled numpy array of dims = (1, *dims, num_channels)
                    if 3D rescaled numpy array of dims = (1, *dims)
        """
        if len(data.shape) is 2:
            data = np.reshape(data, data.shape + (1,))

            # Various settings needed for Tensorflow operation
        boxes = np.zeros((len(scale), 4), dtype=np.float32)
        for index, scale in enumerate(scale):
            x1 = y1 = 0.5 - 0.5 * scale  # To scale centrally
            x2 = y2 = 0.5 + 0.5 * scale
            boxes[index] = np.array([y1, x1, y2, x2], dtype=np.float32)
        box_ind = np.zeros(1, dtype=np.int32)
        crop_size = np.array([data.shape[0], data.shape[1]], dtype=np.int32)

        X_scale_data = []
        tf.reset_default_graph()

        # Define Tensorflow operation for all scales but only one base image at a time
        with tf.Session() as sess:
            batch_img = np.expand_dims(data, axis=0)
            batch_img_tf = tf.Variable(batch_img)
            tf_img = tf.image.crop_and_resize(batch_img_tf, boxes, box_ind, crop_size)
            sess.run(tf.global_variables_initializer())
            scaled_imgs = sess.run(tf_img)
            X_scale_data.extend(scaled_imgs)

        X_scale_data = np.array(X_scale_data, dtype=np.float32)
        X_scale_data = np.reshape(X_scale_data, data.shape)
        return X_scale_data

    def rot90_2D(self, array, num_rot=1):

        def rot_coord_90(coord):
            rot_mat = np.array([[0, -1],
                                [1, 0]
                                ])
            rotated = np.matmul(rot_mat, coord)
            return rotated[0], rotated[1]

        def rot_array_90(array):
            rotated = np.empty(array.shape)
            for coordX in range(array.shape[0]):
                for coordY in range(array.shape[1]):
                    coord = np.array([[coordX, coordY]])
                    rotX, rotY = rot_coord_90(coord.T)
                    if rotX <= 0:
                        rotX = array.shape[0] + rotX - 1
                    rotated[rotX, rotY] = array[coordX, coordY]
```

```
            return rotated

        for i in range(num_rot):
            array = rot_array_90(array)

        return array

    def rot90_3D(self, array, num_rot=1, axis=0):
        if axis == 0:
            for i in range(array.shape[0]):
                array[i, :, :] = self.rot90_2D(array[i, :, :], num_rot=num_rot)
        if axis == 1:
            for j in range(array.shape[1]):
                array[:, j, :] = self.rot90_2D(array[:, j, :], num_rot=num_rot)
        if axis == 2:
            for k in range(array.shape[2]):
                array[:, :, k] = self.rot90_2D(array[:, :, k], num_rot=num_rot)
        return array

    def rot90(self, array, num_rot=1, axis=0):
        dims = len(array.shape)

        if dims is 2:
            array = self.rot90_2D(array, num_rot=num_rot)

        if dims is 3:
            array = self.rot90_3D(array, num_rot=num_rot, axis=axis)

        return array

    def rot_tf(self, orig_array, angle=90, num_rot=1, axis=0):

        num_dims = len(orig_array.shape)
        angle = (angle / 180) * np.pi * num_rot

        config = tf.ConfigProto(log_device_placement=True)
        config.gpu_options.allow_growth = True

        g = tf.Graph()

        with g.as_default():
            sess = tf.Session(graph=g, config=config)

            if num_dims == 2:
                array = np.reshape(orig_array, (1,) + orig_array.shape + (1,))
                array_tf = tf.Variable(array)
                sess.run(tf.global_variables_initializer())
                array = sess.run(array_tf)
                rotated = tf.contrib.image.rotate(array, angle)
                array = sess.run(rotated)

            if num_dims == 3:
                array = np.reshape(orig_array, orig_array.shape + (1,))

                if axis == 0:
                    # No need to reshape (x, y, z, 1), rotate all slices in x
                    array_tf = tf.Variable(array)
                    sess.run(tf.global_variables_initializer())
                    array = sess.run(array_tf)
                    rotated = tf.contrib.image.rotate(array, angle)
                    array = sess.run(rotated)

                if axis == 1:
                    # Reshape to (y, x, z, 1)
                    reshaped_array = np.moveaxis(array, 1, 0)
                    array_tf = tf.Variable(reshaped_array)
                    # Rotate all slices in y
                    sess.run(tf.global_variables_initializer())
                    array = sess.run(array_tf)
                    rotated = tf.contrib.image.rotate(array, angle)
                    rotated_array = sess.run(rotated)
                    # Reshape back to (x, y, z, 1)
                    array = np.moveaxis(rotated_array, 0, 1)

                if axis == 2:
                    # Reshape to (z, x, y, 1)
                    reshaped_array = np.moveaxis(array, 2, 0)  # (z y x 1)
                    reshaped_array = np.moveaxis(reshaped_array, 2, 1)  # (z x y 1)
                    array_tf = tf.Variable(reshaped_array)
                    # Rotate all slices in z
                    sess.run(tf.global_variables_initializer())
                    array = sess.run(array_tf)
                    rotated = tf.contrib.image.rotate(array, angle)
                    rotated_array = sess.run(rotated)
                    # Reshape back to (x, y, z, 1)
                    array = np.moveaxis(rotated_array, 1, 2)  # (z y x 1)
                    array = np.moveaxis(array, 0, 2)  # (x y z 1)

        # tf.reset_default_graph()

        array = np.reshape(array, orig_array.shape)
        return array

    def rotate_data(self, data, rotations, angles):

        """
            :param data: numpy array with dimensions [num_data, data_shape] that will be rotated
```

```
            :param rotations: list with the axes of rotation as strings e.g: ['x', 'y', 'z']
            :param angles: list with the angles of rotation for each axes as ints e.g: [90°, 180°, 270°]
            :return: rotated: numpy array with dimensions [num_data_rotated, data_shape]
        """

        # Initialize rotated
        rotated = list(data)

        # Make transformations (rotations)
        for d in data:
            for r in rotations:
                for angle in angles:
                    rotated.append(self.rot_tf(d, angle=angle, axis=r))

        return np.array(rotated)
```

classes_star.py

```
import os
import matplotlib.pyplot as plt
import pydicom
import pandas as pd

patients_dir = 'F:/Base de datos maestria/StarMedica/patients'
patient_range = (1, 16)


class Patient(object):

    def __init__(self, patient_id, patient_dir):

        self.patient_id = patient_id
        self.patient_dir = patient_dir

        if not os.path.isdir(self.patient_dir):
            print('Creating patient folder')
            os.makedirs(self.patient_dir)

        try:
            self.torax_scan = pydicom.dcmread(self.patient_dir + '/' + self.patient_id + '.dcm')
            self.torax_pixels = self.torax_scan.pixel_array
        except:
            raise RuntimeError('No Mango file has been saved for patient {}.'.format(self.patient_id))

    def anonymize_patient(self):
        print('Anonymizing patient {} data...'.format(self.patient_id))
        self.torax_scan.PatientName = self.patient_id
        self.torax_scan.save_as(self.patient_dir + '/' + self.patient_id + '.dcm')

    def get_scan_pixels(self):
        return self.torax_pixels

    def get_scan(self):
        return self.torax_scan

    def show_ct(self, num_fig=1, title='SCAN', speed=0.001):

        fig = plt.figure(num_fig)
        im = plt.imshow(self.torax_pixels[0], cmap='bone')
        for i, s in enumerate(self.torax_pixels):
            fig.suptitle(self.patient_id + ' ' + title)
            im.set_data(s)
            plt.draw()
            plt.pause(speed)
        fig.clear()

    def get_nodule_labels(self):
        xls_file = pd.read_excel(self.patient_dir + 'stats.xlsx')
```

APPENDIX C

Scripts for data normalization

preprocessing.py

```
import numpy as np
from skimage import measure
import scipy.misc

# Dimension calculated for pre-processing
GLOBAL_MEAN = 0.25
HU_THRESHOLD = -100
VOXEL_SHAPE = [1, 1, 1]  # New resolution between pixels
MIN_BOUND = -1000.0  # Low threshold = -1000 --> air
MAX_BOUND = 400.0  # High threshold = 400 --> we omit bones
EXTRACT_LUNG = False

# DIRECTORY PATHS
patients_dir = 'E:/Base de datos maestría/LIDC-IDRI/DOI'
saving_path = 'E:/Base de datos maestría/LIDC-IDRI'
patients_data_path = 'E:/Base de datos maestría/LIDC-IDRI/patient_data'
luna_data_path = 'E:/Base de datos maestría/LUNA 16/data'


class PreProcessor(object):

    def __init__(self):
        pass

    @staticmethod
    def normalize_data(self, data):
        maxHU = 400.
        minHU = -1000.
        data = (data - minHU) / (maxHU - minHU)
        data[data > 1] = 1.
        data[data < 0] = 0.
        return data

    @staticmethod
    def getMean(self, nparray):
        mean = np.mean(nparray)
        return mean

    @staticmethod
    def substractMeanNormalized(self, nparray):
        nparray = nparray - GLOBAL_MEAN
        return nparray

    @staticmethod
    def rescaleHU(nparray, scan):
        nparray = nparray.astype(np.int16)

        # Set outside-of-scan pixels to 0
        nparray[nparray == -2000] = 0

        # Convert to Hounsfield units (HU)
        for slice_number in range(len(scan)):
            intercept = scan[slice_number].RescaleIntercept
            slope = scan[slice_number].RescaleSlope

            if slope != 1:
                nparray[slice_number] = slope * nparray[slice_number].astype(np.float64)
                nparray[slice_number] = nparray[slice_number].astype(np.int16)

            nparray[slice_number] += np.int16(intercept)

        nparray = np.array(nparray, dtype=np.int16)
        return nparray

    @staticmethod
    def resampleData(self, pixel_data, scan, new_spacing=VOXEL_SHAPE):
        # Resample all scans to have isotropic resolution
        # this allows CONVnets to avoid learning slice thickness variances
        # and also zooming and contraction

        # Determine current pixel spacing in 3 directions
        if len(pixel_data.shape) == 3:
            a = [float(scan[0].SliceThickness), float(scan[0].PixelSpacing[0]), float(scan[0].PixelSpacing[1])]
            spacing = np.array(a, dtype=np.float32)

        # Determine current pixel spacing in 2 directions
        if len(pixel_data.shape) == 2:
            spacing = np.array(scan[0].PixelSpacing, dtype=np.float32)

        # Get the scaling factor to get the new shape of scan
        resize_factor = spacing / new_spacing
        new_real_shape = pixel_data.shape * resize_factor

        if 0 in pixel_data.shape:
```

```
            return None
        new_shape = np.around(new_real_shape, 20)

        real_resize_factor = new_shape / pixel_data.shape
        new_spacing = spacing / real_resize_factor

        resampled_data = scipy.ndimage.interpolation.zoom(pixel_data, real_resize_factor, mode='nearest')

        return resampled_data, real_resize_factor


    @staticmethod
    def largest_label_volume(self, im, bg=-1):

        # We get the all the labels without repeating, that is only the unique number
        # of labels: 1, 2, 3, etc. Not a lot of 2's, 3's and so on.
        # We also want how many pixels are associated with that label with the
        # variable counts.
        vals, counts = np.unique(im, return_counts=True)

        # We only consider the pixels that are not air (bg = background = air)
        # We only consider the greatest pieces of lung or air, it depends
        # As well as the labels
        counts = counts[vals != bg]
        vals = vals[vals != bg]

        # Argmax = Returns the indices of the maximum values along an axis.
        # with the index with the max value we can know which label (group of
        # components) hast the largest volume
        if len(counts) > 0:
            return vals[np.argmax(counts)]
        else:
            return None

    def segment_lung_mask(self, image, hu_value, fill_lung_structures=True):
        # We get all the pixels that have an intenisty greater that HU -320
        # Because we know that HU for lungs is -500
        # We don't want 0 or 1 (not actually binary, but 1 and 2) that is why
        # we add 1..
        # 0 is treated as background, which we do not want
        binary_image = np.array(image > hu_value, dtype=np.int8) + 1

        # Label connected regions of an integer array.
        # Two pixels are connected when they are neighbors and have the same value.
        # In 2D, they can be neighbors either in a 1- or 2-connected sense. The
        # value refers to the maximum number of orthogonal hops to consider a
        # pixel/voxel a neighbor:
        # By default we have a connectivity of 4, face sharing.
        labels = measure.label(binary_image)

        # Pick the pixel in the very corner to determine which label is air.
        #   Improvement: Pick multiple background labels from around the patient
        #   More resistant to "trays" on which the patient lays cutting the air
        #   around the person in half
        background_label = labels[0, 0, 0]

        # Fill the air around the person
        # We know the label related with air which is the background label
        # We search all the connected components assigned to that label, that
        # is we search for air (background_label == labels) and we fill the image
        # with actual "tissue" different from lung

        binary_image[background_label == labels] = 2

        # Method of filling the lung structures (that is superior to something like
        # morphological closing)
        if fill_lung_structures:
            # Enumerate allows us to loop over something and have an automatic
            # counter.
            # For every slice we determine the largest solid structure
            for i, axial_slice in enumerate(binary_image):

                # We substract 1 beacuse enumerate returns the list starting
                # in 1, but arrays only work straing in 0
                axial_slice = axial_slice - 1

                # We get the labels of each connected component
                labeling = measure.label(axial_slice)

                # Find the label (group of connected components) with the
                # largest volume, specifying that we do not want to take
                # into account the pixels that are air = bg = background
                l_max = self.largest_label_volume(labeling, bg=0)

                if l_max is not None:  # This slice contains some lung ( 1 )
                    # We search for all the connected components in the image
                    # if the label of the connected component (piece of lung)
```

```
                # doesnt correspond with the label of the maximum volume of lung
                # we assign it as lung because is "tissue" that actually
                # needs to be lung
                binary_image[i][labeling != l_max] = 1

        binary_image -= 1  # Make the image actual binary, lungs are 0
        binary_image = 1 - binary_image  # Invert it, lungs are now 1

        # Remove other air pockets insided body
        # Background = 0 --> Consider all pixels with this value as background
        # pixels, and label them as 0. By default, 0-valued pixels are considered
        # as background pixels.
        # Get the labels of connected components, but now we have labels
        # as 0 for all coponents that have a value of 0
        labels = measure.label(binary_image, background=0)
        l_max = self.largest_label_volume(labels, bg=0)  #
        if l_max is not None:  # There are air pockets
            binary_image[labels != l_max] = 0

        # We apply a dilation with conectivity 1 in 2 dimensions
        dilated_binary_image = scipy.ndimage.morphology.binary_dilation(binary_image, iterations=1)

        return dilated_binary_image

    def preProcess(self, patient_pixels, scan):
        # Normalize the images
        rescaled_pixels = self.rescaleHU(patient_pixels, scan)
        norm_pix = self.normalize_data(rescaled_pixels)

        # Segment lungs
        if EXTRACT_LUNG:
            # Normalize and subtract global mean to the HU threshold
            hu_value_normalized = ((HU_THRESHOLD - MIN_BOUND) / (MAX_BOUND - MIN_BOUND)) - GLOBAL_MEAN
            segmented_lungs_fill = self.segment_lung_mask(norm_pix, hu_value_normalized, True)
            # Fill all the spaces between lungs
            segmented_lungs_fill = np.multiply(segmented_lungs_fill, norm_pix)
            return segmented_lungs_fill
        else:
            return norm_pix
```

APPENDIX D

Script for all detectors

CNN_Models.py

```
from keras.layers import Add, Input, AveragePooling3D, ZeroPadding3D, Dense, Activation, Lambda, Concatenate, \
    concatenate
from keras.layers import BatchNormalization, Flatten, Conv3D, MaxPooling3D, Dropout, GlobalAveragePooling3D, \
    GlobalMaxPooling3D
from keras.models import Model
from keras import regularizers
from keras import backend

class Bayesian:

    def __init__(self):
        pass

    # ---------------------------------- VGG 16 ---------------------------------#

    # Instead of starting with 64 filters, you start with 8 filters
    # the reason is because of GPU limitations

    # CNN MODEL - Similiar to Vgg - 16
    def model_vgg_16(input_shape,
                     n_classes,
                     kernel_initializer,
                     bias_initializer,
                     use_dropout,
                     prob_dropout):

        # Define the input placeholder as a tensor with shape input_shape.
        # Think of this as your input image!
        x_input = Input(shape=input_shape)

        # CONV LAYER 1
        # CONV -> BN -> RELU Block applied to x
        x = Conv3D(6, (3, 3, 3), strides=(1, 1, 1), padding='same',
                   name='conv_0', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(x_input)
        x = BatchNormalization(axis=4, name='bn_0')(x)
        x = Activation('relu')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # CONV LAYER 2
        # CONV -> BN -> RELU Block applied to x
        x = Conv3D(8, (3, 3, 3), padding='same',
                   strides=(1, 1, 1),
                   name='conv_1',
                   kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(x)
        x = BatchNormalization(axis=4, name='bn_1')(x)
        x = Activation('relu')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # MAXPOOL 1
        x = MaxPooling3D(pool_size=(2, 2, 2),
                         strides=(2, 2, 2),
                         padding='valid',
                         name='max_pool_1')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # CONV LAYER 3
        # CONV -> BN -> RELU Block applied to x
        x = Conv3D(16, (3, 3, 3), strides=(1, 1, 1), padding='same',
                   name='conv_2', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(x)
        x = BatchNormalization(axis=4, name='bn_2')(x)
        x = Activation('relu')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # CONV LAYER 4
        # CONV -> BN -> RELU Block applied to x
        x = Conv3D(16, (3, 3, 3), strides=(1, 1, 1), padding='same',
                   name='conv_3', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(x)
        x = BatchNormalization(axis=4, name='bn_3')(x)
        x = Activation('relu')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # MAXPOOL 2
        x = MaxPooling3D(pool_size=(2, 2, 2),
                         strides=(2, 2, 2),
                         padding='valid',
                         name='max_pool_2')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # CONV LAYER 5
        # CONV -> BN -> RELU Block applied to x
        x = Conv3D(32, (3, 3, 3), strides=(1, 1, 1), padding='same',
```

```
                name='conv_4', kernel_initializer=kernel_initializer,
                bias_initializer=bias_initializer)(x)
    x = BatchNormalization(axis=4, name='bn_4')(x)
    x = Activation('relu')(x)
    # DROPOUT
    if use_dropout:
        x = Dropout(prob_dropout)(x)

    # CONV LAYER 6
    # CONV -> BN -> RELU Block applied to x
    x = Conv3D(32, (3, 3, 3), strides=(1, 1, 1), padding='same',
                name='conv_5', kernel_initializer=kernel_initializer,
                bias_initializer=bias_initializer)(x)
    x = BatchNormalization(axis=4, name='bn_5')(x)
    x = Activation('relu')(x)
    # DROPOUT
    if use_dropout:
        x = Dropout(prob_dropout)(x)

    # CONV LAYER 7
    # CONV -> BN -> RELU Block applied to x
    x = Conv3D(32, (3, 3, 3), strides=(1, 1, 1), padding='same',
                name='conv_6', kernel_initializer=kernel_initializer,
                bias_initializer=bias_initializer)(x)
    x = BatchNormalization(axis=4, name='bn_6')(x)
    x = Activation('relu')(x)
    # DROPOUT
    if use_dropout:
        x = Dropout(prob_dropout)(x)

    # MAXPOOL 3
    x = MaxPooling3D(pool_size=(2, 2, 2),
                     strides=(2, 2, 2),
                     padding='valid',
                     name='max_pool_3')(x)
    # DROPOUT
    if use_dropout:
        x = Dropout(prob_dropout)(x)

    # FC LAYER 1
    # FLATTEN x (means convert it to a vector) + FULLYCONNECTED
    x = Flatten()(x)
    x = Dense(256, activation='relu', name='fc_0',
              kernel_initializer=kernel_initializer,
              bias_initializer=bias_initializer)(x)
    # DROPOUT
    if use_dropout:
        x = Dropout(prob_dropout)(x)

    # FC LAYER 2
    # FLATTEN x (means convert it to a vector) + FULLYCONNECTED
    x = Dense(256, activation='relu', name='fc_1',
              kernel_initializer=kernel_initializer,
              bias_initializer=bias_initializer)(x)
    # DROPOUT
    if use_dropout:
        x = Dropout(prob_dropout)(x)

    # FC LAYER 3
    # FLATTEN x (means convert it to a vector) + FULLYCONNECTED
    x = Dense(256, activation='relu', name='fc_2',
              kernel_initializer=kernel_initializer,
              bias_initializer=bias_initializer)(x)
    # DROPOUT
    if use_dropout:
        x = Dropout(prob_dropout)(x)

    # FC LAYER 4
    # FLATTEN x (means convert it to a vector) + FULLYCONNECTED
    if n_classes <= 2:
        x = Dense(1, activation='sigmoid', name='fc_3',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(x)
    else:
        x = Dense(n_classes, activation='softmax', name='fc_3',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(x)

    # COMPLETE CONV NETWORK
    # Create model. This creates your Keras model instance,
    # you'll use this instance to train/test the model.
    model = Model(inputs=x_input, outputs=x, name='Vgg16')

    return model

# ---------------------------------- AlexNet ---------------------------------#

# CNN MODEL - Similiar to AlexNet, less strides, with stride = 1
# Instead of 6 x 6 x 256 we have 12 x 12 x 256 at the last conv layer.
# That is why we modify the fully connected layer
# to have 9216 / 144 = 64 neurons

def model_alexnet(input_shape,
                  n_classes,
                  kernel_initializer,
                  bias_initializer,
                  use_dropout,
                  prob_dropout):
```

```
        # Define the input placeholder as a tensor with shape input_shape.
        # Think of this as your input image!
        x_input = Input(shape=input_shape)

        # CONV LAYER 1
        # CONV -> BN -> RELU Block applied to x
        x = Conv3D(96, (11, 11, 11), strides=(1, 1, 1), padding='valid',
                   name='conv_0', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(x_input)
        x = BatchNormalization(axis=4, name='bn_0')(x)
        x = Activation('relu')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # MAXPOOL 1
        x = MaxPooling3D(pool_size=(3, 3, 3),
                         strides=(2, 2, 2),
                         padding='valid',
                         name='max_pool_1')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # CONV LAYER 2
        # CONV -> BN -> RELU Block applied to x
        x = Conv3D(256, (5, 5, 5), strides=(1, 1, 1), padding='same',
                   name='conv_1', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(x)
        x = BatchNormalization(axis=4, name='bn_1')(x)
        x = Activation('relu')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # MAXPOOL 2
        x = MaxPooling3D(pool_size=(3, 3, 3),
                         strides=(2, 2, 2),
                         padding='valid',
                         name='max_pool_2')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # CONV LAYER 3
        # CONV -> BN -> RELU Block applied to x
        x = Conv3D(384, (3, 3, 3), strides=(1, 1, 1), padding='same',
                   name='conv_2', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(x)
        x = BatchNormalization(axis=4, name='bn_2')(x)
        x = Activation('relu')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # CONV LAYER 4
        # CONV -> BN -> RELU Block applied to x
        x = Conv3D(384, (3, 3, 3), strides=(1, 1, 1), padding='same',
                   name='conv_3', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(x)
        x = BatchNormalization(axis=4, name='bn_3')(x)
        x = Activation('relu')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # CONV LAYER 5
        # CONV -> BN -> RELU Block applied to x
        x = Conv3D(256, (3, 3, 3), strides=(1, 1, 1), padding='same',
                   name='conv_4', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(x)
        x = BatchNormalization(axis=4, name='bn_4')(x)
        x = Activation('relu')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # MAXPOOL 3
        x = MaxPooling3D(pool_size=(3, 3, 3),
                         strides=(1, 1, 1),
                         padding='valid',
                         name='max_pool_3')(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # FC LAYER 1
        # FLATTEN x (means convert it to a vector) + FULLYCONNECTED
        x = Flatten()(x)
        x = Dense(64, activation='relu', name='fc_0',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # FC LAYER 2
        # FLATTEN x (means convert it to a vector) + FULLYCONNECTED
        x = Dense(32, activation='relu', name='fc_1',
```

```
                kernel_initializer=kernel_initializer,
                bias_initializer=bias_initializer)(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # FC LAYER 3
        # FLATTEN x (means convert it to a vector) + FULLYCONNECTED
        x = Dense(32, activation='relu', name='fc_2',
                kernel_initializer=kernel_initializer,
                bias_initializer=bias_initializer)(x)
        # DROPOUT
        if use_dropout:
            x = Dropout(prob_dropout)(x)

        # FC LAYER 4
        # FLATTEN x (means convert it to a vector) + FULLYCONNECTED
        if n_classes <= 2:
            x = Dense(1, activation='sigmoid', name='fc_3',
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(x)
        else:
            x = Dense(n_classes, activation='softmax', name='fc_3',
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(x)

        # COMPLETE CONV NETWORK
        # Create model. This creates your Keras model instance,
        # you'll use this instance to train/test the model.
        model = Model(inputs=x_input, outputs=x, name='Alexnet')

        return model

    # ---------------------------------- ResNet ----------------------------------#

    # CNN MODEL - Similiar to ResNet
    def model_resnet_50(input_shape,
                        n_classes,
                        kernel_initializer,
                        bias_initializer,
                        use_dropout,
                        prob_dropout,
                        use_l2_regularizer,
                        l2_regularizer):

        def convolutional_block(x, f, filters, stage, block, kernel_initializer,
                                bias_initializer,
                                use_l2_regularizer,
                                l2_regularizer,
                                s=2):
            """
            Implementation of the convolutional block, when dimensions mismatch
            between the input activation and the last activation

            Arguments:
            x -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
            f -- integer, specifying the shape of the middle CONV's window for the main path
            filters -- python list of integers, defining the number of filters in the CONV layers of the main path
            stage -- integer, used to name the layers, depending on their position in the network
            block -- string/character, used to name the layers, depending on their position in the network
            s -- Integer, specifying the stride to be used

            Returns:
            x -- output of the convolutional block, tensor of shape (n_H, n_W, n_C)
            """

            # defining name basis
            conv_name_base = 'res' + str(stage) + block + '_branch'
            bn_name_base = 'bn' + str(stage) + block + '_branch'

            # Retrieve Filters
            F1, F2, F3 = filters

            # Save the input value
            X_shortcut = x

            # First component of main path
            if use_l2_regularizer:
                x = Conv3D(F1, (1, 1, 1), strides=(s, s, s),
                           name=conv_name_base + '2a',
                           kernel_initializer=kernel_initializer,
                           bias_initializer=bias_initializer,
                           kernel_regularizer=regularizers.l2(l2_regularizer))(x)
            else:
                x = Conv3D(F1, (1, 1, 1), strides=(s, s, s),
                           name=conv_name_base + '2a',
                           kernel_initializer=kernel_initializer,
                           bias_initializer=bias_initializer)(x)
            x = BatchNormalization(axis=4, name=bn_name_base + '2a')(x)
            x = Activation('relu')(x)

            # Second component of main path (≈3 lines)
            if use_l2_regularizer:
                x = Conv3D(filters=F2, kernel_size=(f, f, f), strides=(1, 1, 1),
                           padding='same', name=conv_name_base + '2b',
                           kernel_initializer=kernel_initializer,
                           bias_initializer=bias_initializer,
                           kernel_regularizer=regularizers.l2(l2_regularizer))(x)
            else:
```

```
                x = Conv3D(filters=F2, kernel_size=(f, f, f), strides=(1, 1, 1),
                           padding='same', name=conv_name_base + '2b',
                           kernel_initializer=kernel_initializer,
                           bias_initializer=bias_initializer)(x)
            x = BatchNormalization(axis=4, name=bn_name_base + '2b')(x)
            x = Activation('relu')(x)

            # Third component of main path (≈2 lines)
            if use_l2_regularizer:
                x = Conv3D(filters=F3, kernel_size=(1, 1, 1), strides=(1, 1, 1),
                           padding='valid', name=conv_name_base + '2c',
                           kernel_initializer=kernel_initializer,
                           bias_initializer=bias_initializer,
                           kernel_regularizer=regularizers.l2(l2_regularizer))(x)
            else:
                x = Conv3D(filters=F3, kernel_size=(1, 1, 1), strides=(1, 1, 1),
                           padding='valid', name=conv_name_base + '2c',
                           kernel_initializer=kernel_initializer,
                           bias_initializer=bias_initializer)(x)
            x = BatchNormalization(axis=4, name=bn_name_base + '2c')(x)

            X_shortcut = Conv3D(filters=F3, kernel_size=(1, 1, 1),
                                strides=(s, s, s), padding='valid',
                                name=conv_name_base + '1',
                                kernel_initializer=kernel_initializer,
                                bias_initializer=bias_initializer)(X_shortcut)
            X_shortcut = BatchNormalization(axis=4, name=bn_name_base + '1')(X_shortcut)

            # Final step: Add shortcut value to main path, and pass it through a
            # RELU activation (≈2 lines)
            x = Add()([X_shortcut, x])
            x = Activation('relu')(x)

            return x

        def identity_block(X, f, filters, stage, block,
                           kernel_initializer,
                           bias_initializer,
                           use_l2_regularizer,
                           l2_regularizer):
            """
            Implementation of the identity block composed of three mini blocks:
                [ CONV3D --> Batch Norm --> RELU ]

            Arguments:
            X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
            f -- integer, specifying the shape of the middle CONV's window for the main path
            filters -- python list of integers, defining the number of filters in the CONV layers of the main path
            stage -- integer, used to name the layers, depending on their position in the network
            block -- string/character, used to name the layers, depending on their position in the network

            Returns:
            X -- output of the identity block, tensor of shape (n_H, n_W, n_C)
            """

            # defining name basis
            conv_name_base = 'res' + str(stage) + block + '_branch'
            bn_name_base = 'bn' + str(stage) + block + '_branch'

            # Retrieve Filters
            F1, F2, F3 = filters

            # Save the input value. You'll need this later to add back to the main path.
            X_shortcut = X

            # First component of main path
            if use_l2_regularizer:
                X = Conv3D(filters=F1, kernel_size=(1, 1, 1), strides=(1, 1, 1),
                           padding='valid', name=conv_name_base + '2a',
                           kernel_initializer=kernel_initializer,
                           bias_initializer=bias_initializer,
                           kernel_regularizer=regularizers.l2(l2_regularizer))(X)
            else:
                X = Conv3D(filters=F1, kernel_size=(1, 1, 1), strides=(1, 1, 1),
                           padding='valid', name=conv_name_base + '2a',
                           kernel_initializer=kernel_initializer,
                           bias_initializer=bias_initializer)(X)
            X = BatchNormalization(axis=4, name=bn_name_base + '2a')(X)
            X = Activation('relu')(X)

            # Second component of main path (≈3 lines)
            if use_l2_regularizer:
                X = Conv3D(filters=F2, kernel_size=(f, f, f), strides=(1, 1, 1),
                           padding='same', name=conv_name_base + '2b',
                           kernel_initializer=kernel_initializer,
                           bias_initializer=bias_initializer,
                           kernel_regularizer=regularizers.l2(l2_regularizer))(X)
            else:
                X = Conv3D(filters=F2, kernel_size=(f, f, f), strides=(1, 1, 1),
                           padding='same', name=conv_name_base + '2b',
                           kernel_initializer=kernel_initializer,
                           bias_initializer=bias_initializer)(X)
            X = BatchNormalization(axis=4, name=bn_name_base + '2b')(X)
            X = Activation('relu')(X)

            # Third component of main path (≈2 lines)
            if use_l2_regularizer:
                X = Conv3D(filters=F3, kernel_size=(1, 1, 1), strides=(1, 1, 1),
                           padding='valid', name=conv_name_base + '2c',
```

```
                    kernel_initializer=kernel_initializer,
                    bias_initializer=bias_initializer,
                    kernel_regularizer=regularizers.l2(l2_regularizer))(X)
        else:
            X = Conv3D(filters=F3, kernel_size=(1, 1, 1), strides=(1, 1, 1),
                    padding='valid', name=conv_name_base + '2c',
                    kernel_initializer=kernel_initializer,
                    bias_initializer=bias_initializer)(X)
        X = BatchNormalization(axis=4, name=bn_name_base + '2c')(X)

        # Final step: Add shortcut value to main path, and pass it through a
        # RELU activation (≈2 lines)
        X = Add()([X_shortcut, X])
        X = Activation('relu')(X)

        return X

    # Define the input placeholder as a tensor with shape input_shape.

    # Think of this as your input image!
    X_input = Input(shape=input_shape)

    # Zero-Padding
    X = ZeroPadding3D((3, 3, 3))(X_input)

    # Stage 1
    if use_l2_regularizer:
        X = Conv3D(32, (3, 3, 3), strides=(1, 1, 1), name='conv1',
                kernel_initializer=kernel_initializer,
                bias_initializer=bias_initializer,
                kernel_regularizer=regularizers.l2(l2_regularizer))(X)
    else:
        X = Conv3D(32, (3, 3, 3), strides=(1, 1, 1), name='conv1',
                kernel_initializer=kernel_initializer,
                bias_initializer=bias_initializer)(X)
    X = BatchNormalization(axis=4, name='bn_conv1')(X)
    X = Activation('relu')(X)
    X = MaxPooling3D((3, 3, 3), strides=(2, 2, 2))(X)

    # Stage 2
    X = convolutional_block(X, f=3, filters=[32, 32, 64], stage=2,
                            block='a', s=1,
                            kernel_initializer=kernel_initializer,
                            bias_initializer=bias_initializer,
                            use_l2_regularizer=use_l2_regularizer,
                            l2_regularizer=l2_regularizer)
    X = identity_block(X, 3, [32, 32, 64], stage=2, block='b',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
                       l2_regularizer=l2_regularizer)
    # DROPOUT
    if use_dropout:
        X = Dropout(prob_dropout)(X)

    X = identity_block(X, 3, [32, 32, 64], stage=2, block='c',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
                       l2_regularizer=l2_regularizer)

    # Stage 3
    X = convolutional_block(X, f=3, filters=[64, 64, 128], stage=3,
                            block='a', s=2,
                            kernel_initializer=kernel_initializer,
                            bias_initializer=bias_initializer,
                            use_l2_regularizer=use_l2_regularizer,
                            l2_regularizer=l2_regularizer)
    X = identity_block(X, 3, [64, 64, 128], stage=3, block='b',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
                       l2_regularizer=l2_regularizer)
    X = identity_block(X, 3, [64, 64, 128], stage=3, block='c',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
                       l2_regularizer=l2_regularizer)
    # DROPOUT
    if use_dropout:
        X = Dropout(prob_dropout)(X)

    X = identity_block(X, 3, [64, 64, 128], stage=3, block='d',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
                       l2_regularizer=l2_regularizer)

    # Stage 4
    X = convolutional_block(X, f=3, filters=[128, 128, 256], stage=4,
                            block='a', s=2,
                            kernel_initializer=kernel_initializer,
                            bias_initializer=bias_initializer,
                            use_l2_regularizer=use_l2_regularizer,
                            l2_regularizer=l2_regularizer)
    X = identity_block(X, 3, [128, 128, 256], stage=4, block='b',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
```

```
                        l2_regularizer=l2_regularizer)
    X = identity_block(X, 3, [128, 128, 256], stage=4, block='c',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
                       l2_regularizer=l2_regularizer)
    # DROPOUT
    if use_dropout:
        X = Dropout(prob_dropout)(X)

    X = identity_block(X, 3, [128, 128, 256], stage=4, block='d',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
                       l2_regularizer=l2_regularizer)
    X = identity_block(X, 3, [128, 128, 256], stage=4, block='e',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
                       l2_regularizer=l2_regularizer)

    # DROPOUT
    if use_dropout:
        X = Dropout(prob_dropout)(X)

    X = identity_block(X, 3, [128, 128, 256], stage=4, block='f',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
                       l2_regularizer=l2_regularizer)

    # Stage 5
    X = convolutional_block(X, f=3, filters=[256, 256, 512], stage=5,
                            block='a', s=2,
                            kernel_initializer=kernel_initializer,
                            bias_initializer=bias_initializer,
                            use_l2_regularizer=use_l2_regularizer,
                            l2_regularizer=l2_regularizer)
    X = identity_block(X, 3, [256, 256, 512], stage=5, block='b',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
                       l2_regularizer=l2_regularizer)

    # DROPOUT
    if use_dropout:
        X = Dropout(prob_dropout)(X)

    X = identity_block(X, 3, [256, 256, 512], stage=5, block='c',
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       use_l2_regularizer=use_l2_regularizer,
                       l2_regularizer=l2_regularizer)

    # Fully Connected Layer
    X = AveragePooling3D((2, 2, 2), name='avg_pool')(X)
    X = Flatten()(X)

    if n_classes <= 2:
        X = Dense(1, activation='sigmoid', name='fc_2',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(X)
    else:
        X = Dense(n_classes, activation='softmax', name='fc_2',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(X)

    # COMPLETE CONV NETWORK
    # Create model. This creates your Keras model instance,
    # you'll use this instance to train/test the model.
    model = Model(inputs=X_input, outputs=X, name='ResNet50')

    return model

# CNN MODEL - Similiar to ResNet

def model_resnet_adaptative(input_shape,
                            n_classes,
                            space,
                            kernel_initializer,
                            bias_initializer):

    def convolutional_block(X, f, filters, stage, block, kernel_initializer,
                            num_conv,
                            bias_initializer,
                            l2_regularizer,
                            s=2):
        """
        Implementation of the convolutional block, when dimensions mismatch
        between the input activation and the last activation

        Arguments:
        X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
        f -- integer, specifying the shape of the middle CONV's window for the main path
        filters -- python list of integers, defining the number of filters in the CONV layers of the main path
        stage -- integer, used to name the layers, depending on their position in the network
        block -- string/character, used to name the layers, depending on their position in the network
        s -- Integer, specifying the stride to be used
```

```
        Returns:
        X -- output of the convolutional block, tensor of shape (n_H, n_W, n_C)
        """

        # defining name basis
        conv_name_base = 'conv_res' + str(stage) + block + str(num_conv) + '_branch'
        bn_name_base = 'conv_bn' + str(stage) + block + str(num_conv) + '_branch'

        # Retrieve Filters
        F1, F2, F3 = filters

        # Save the input value
        X_shortcut = X

        # First component of main path
        X = Conv3D(F1, (1, 1, 1), strides=(s, s, s),
                   name=conv_name_base + '2a',
                   kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer,
                   kernel_regularizer=regularizers.l2(l2_regularizer))(X)
        X = BatchNormalization(axis=4, name=bn_name_base + '2a')(X)
        X = Activation('relu')(X)

        # Second component of main path (≈3 lines)
        X = Conv3D(filters=F2, kernel_size=(f, f, f), strides=(1, 1, 1),
                   padding='same', name=conv_name_base + '2b',
                   kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer,
                   kernel_regularizer=regularizers.l2(l2_regularizer))(X)
        X = BatchNormalization(axis=4, name=bn_name_base + '2b')(X)
        X = Activation('relu')(X)

        # Third component of main path (≈2 lines)
        X = Conv3D(filters=F3, kernel_size=(1, 1, 1), strides=(1, 1, 1),
                   padding='valid', name=conv_name_base + '2c',
                   kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer,
                   kernel_regularizer=regularizers.l2(l2_regularizer))(X)
        X = BatchNormalization(axis=4, name=bn_name_base + '2c')(X)

        # SHORTCUT PATH
        X_shortcut = Conv3D(filters=F3, kernel_size=(1, 1, 1),
                            strides=(s, s, s), padding='valid',
                            name=conv_name_base + '1',
                            kernel_initializer=kernel_initializer,
                            bias_initializer=bias_initializer)(X_shortcut)
        X_shortcut = BatchNormalization(axis=4, name=bn_name_base + '1')(X_shortcut)

        # Final step: Add shortcut value to main path, and pass it through a
        # RELU activation (≈2 lines)
        X = Add()([X_shortcut, X])
        X = Activation('relu')(X)

        return X

    def identity_block(X, f, filters, stage, block,
                       num_identity,
                       kernel_initializer,
                       bias_initializer,
                       l2_regularizer):
        """
        Implementation of the identity block composed of three mini blocks:
            [ CONV3D --> Batch Norm --> RELU ]

        Arguments:
        X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
        f -- integer, specifying the shape of the middle CONV's window for the main path
        filters -- python list of integers, defining the number of filters in the CONV layers of the main path
        stage -- integer, used to name the layers, depending on their position in the network
        block -- string/character, used to name the layers, depending on their position in the network

        Returns:
        X -- output of the identity block, tensor of shape (n_H, n_W, n_C)
        """

        # Defining name basis
        conv_name_base = 'ident_res' + str(stage) + block + str(num_identity) + '_branch'
        bn_name_base = 'ident_bn' + str(stage) + block + str(num_identity) + '_branch'

        # Retrieve Filters
        F1, F2, F3 = filters

        # Save the input value.
        X_shortcut = X

        # First component of main path
        X = Conv3D(filters=F1, kernel_size=(1, 1, 1), strides=(1, 1, 1),
                   padding='valid', name=conv_name_base + '2a',
                   kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer,
                   kernel_regularizer=regularizers.l2(l2_regularizer))(X)
        X = BatchNormalization(axis=4, name=bn_name_base + '2a')(X)
        X = Activation('relu')(X)

        # Second component of main path
        X = Conv3D(filters=F2, kernel_size=(f, f, f), strides=(1, 1, 1),
                   padding='same', name=conv_name_base + '2b',
                   kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer,
```

```
                    kernel_regularizer=regularizers.l2(l2_regularizer))(X)
        X = BatchNormalization(axis=4, name=bn_name_base + '2b')(X)
        X = Activation('relu')(X)

        # Third component of main path
        X = Conv3D(filters=F3, kernel_size=(1, 1, 1), strides=(1, 1, 1),
                   padding='valid', name=conv_name_base + '2c',
                   kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer,
                   kernel_regularizer=regularizers.l2(l2_regularizer))(X)
        X = BatchNormalization(axis=4, name=bn_name_base + '2c')(X)

        # Final step: Add shortcut value to main path, and pass it through a
        X = Add()([X_shortcut, X])
        X = Activation('relu')(X)

        return X

    # ----------------------------- START -----------------------------

    # Define input placeholder as a tensor with shape input_shape.
    X_input = Input(shape=input_shape)

    # Zero-Padding
    X = ZeroPadding3D((3, 3, 3))(X_input)

    # ----------------------------- Sample Space -------------------------

    # Number of filters for each dimension
    F1 = space['F1']
    F2 = space['F2']
    F3 = space['F3']

    # Size of kernel
    f = space['filter_size']

    # Number of conv blocks and identity
    num_conv = space['num_conv']
    num_identity = space['num_identity']
    num_stages = space['num_stages']

    # Regularizers
    L2_conv = space['L2_conv']
    L2_identity = space['L2_identity']

    # ----------------------------- START -----------------------------
    X = Conv3D(32, (3, 3, 3), strides=(1, 1, 1), name='conv1',
               kernel_initializer=kernel_initializer,
               bias_initializer=bias_initializer,
               kernel_regularizer=regularizers.l2(L2_conv))(X)

    X = BatchNormalization(axis=4, name='bn_conv1')(X)
    X = Activation('relu')(X)
    X = MaxPooling3D((3, 3, 3), strides=(2, 2, 2))(X)

    # ----------------------------- Stages -----------------------------
    for stage in range(0, num_stages):

        # ----------------- Conv Blocks ---------------------------------
        for n_conv in range(0, num_conv):
            X = convolutional_block(X, f=f, filters=[F1, F2, F3], stage=stage,
                                    num_conv=n_conv, block='a', s=1,
                                    kernel_initializer=kernel_initializer,
                                    bias_initializer=bias_initializer,
                                    l2_regularizer=L2_conv)

        # ----------------- Idenity Blocks ------------------------------
        for n_identity in range(0, num_identity):
            X = identity_block(X, 3, [F1, F2, F3], stage=stage,
                               num_identity=n_identity,
                               block='b',
                               kernel_initializer=kernel_initializer,
                               bias_initializer=bias_initializer,
                               l2_regularizer=L2_identity)

        # Double number of filters for each stage
        F1 = 2 * F1
        F2 = 2 * F2
        F3 = 2 * F3

    # ----------------------------- FINAL ---------------------------------
    X = AveragePooling3D((2, 2, 2), name='avg_pool')(X)
    X = Flatten()(X)

    if n_classes <= 2:
        X = Dense(1, activation='sigmoid', name='fc_2',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(X)
    else:
        X = Dense(n_classes, activation='softmax', name='fc_2',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(X)

    model = Model(inputs=X_input, outputs=X, name='ResNet50')

    return model

# --------------------------------- GoogLeNet ---------------------------------#
```

```
    # CNN MODEL - Similiar to ResNet

    # Going deeper with convolutions
    # Szegedy, Christian; Liu, Wei; Jia, Yangqing; Sermanet, Pierre; Reed, Scott; Anguelov, Dragomir;
    # Erhan, Dumitru; Vanhoucke, Vincent; Rabinovich, Andrew
    # arXiv:1409.4842

    def model_googlenet(input_shape,
                        n_classes,
                        space,
                        kernel_initializer,
                        bias_initializer):

        def inception_block(X, bottlenecks, filters, block,
                            kernel_initializer, bias_initializer, l2_regularizer):
            """
            Implementation of the inception block

            Arguments:
            X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
            bottlenecks -- python list of integers, defining the number of filters for each bottleneck in inception block
            filters -- python list of integers, defining the number of filters for each of the 4 branches applying Convolutions
            block -- string/character, used to name the layers, depending on their position in the network

            Returns:
            X -- output of the convolutional block, tensor of shape (n_H, n_W, n_C)
            """

            # defining name basis
            conv_name_base = 'block' + block + '_branch_'
            bn_name_base = 'block' + block + '_branch_'

            # Retrieve Filters
            F1, F2, F3, F4 = filters

            # Retrieve Bottlenecks
            B2, B3 = bottlenecks

            # First branch - 1 x 1 conv
            b1 = Conv3D(F1, (1, 1, 1), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'conv_1',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer)(X)
            b1 = BatchNormalization(axis=4, name=bn_name_base + 'bn1')(b1)
            b1 = Activation('relu')(b1)

            # Second branch - bottleneck
            b2 = Conv3D(B2, (1, 1, 1), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'bottleneck_2',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer)(X)
            # Second branch - Convolution 3 x 3 x 3
            b2 = Conv3D(F2, (3, 3, 3), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'conv_2',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer))(b2)
            b2 = BatchNormalization(axis=4, name=bn_name_base + 'bn2')(b2)
            b2 = Activation('relu')(b2)

            # Third branch - bottleneck
            b3 = Conv3D(B3, (1, 1, 1), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'bottleneck_3',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer)(X)
            b3 = BatchNormalization(axis=4, name=bn_name_base + 'bn3')(b3)
            b3 = Activation('relu')(b3)
            # Third branch - Convolution 5 x 5 x 5
            b3 = Conv3D(F3, (5, 5, 5), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'conv_3',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer))(b3)
            b3 = BatchNormalization(axis=4, name=bn_name_base + 'bn4')(b3)
            b3 = Activation('relu')(b3)

            # Concatenate results of 3 branches
            X = concatenate([b1, b2, b3], axis=4)

            return X

        # Define the input placeholder as a tensor with shape input_shape.

        # Think of this as your input image!
        X_input = Input(shape=input_shape)

        # First Inception Block
        X = inception_block(X_input, filters=[8, 16, 4, 4],
                            bottlenecks=[16, 8], block='1',
                            kernel_initializer=kernel_initializer,
                            bias_initializer=bias_initializer,
                            l2_regularizer=0)

        # Second Inception Block
        X = inception_block(X, filters=[8, 16, 4, 4],
                            bottlenecks=[16, 8], block='2',
                            kernel_initializer=kernel_initializer,
                            bias_initializer=bias_initializer,
                            l2_regularizer=space['L2_s1'])
```

```
        # Third Inception Block
        X = inception_block(X, filters=[4, 8, 2, 2],
                            bottlenecks=[8, 4], block='3',
                            kernel_initializer=kernel_initializer,
                            bias_initializer=bias_initializer,
                            l2_regularizer=space['L2_s2'])

        # Fully Connected Layer
        X = Flatten()(X)

        # Add extra dense layers
        num_layer = 1
        for num_layer in range(0, space['extra_dense']):
            extra_units = space['extra_units'] + 1
            X = Dense(extra_units, activation='relu', name='fc_' + str(num_layer),
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)

        if n_classes <= 2:
            X = Dense(1, activation='sigmoid', name='fc_' + str(num_layer + 1),
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)
        else:
            X = Dense(n_classes, activation='softmax', name='fc_' + str(num_layer + 1),
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)
        # Complete Model
        model = Model(inputs=X_input, outputs=X, name='GoogLeNet')

        return model

    def model_googlenet_light(input_shape,
                              n_classes,
                              space,
                              kernel_initializer,
                              bias_initializer):

        def inception_block(X, bottlenecks, filters, block,
                            kernel_initializer, bias_initializer,
                            l2_regularizer_1, l2_regularizer_2):
            """
            Implementation of the inception block

            Arguments:
            X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
            bottlenecks -- python list of integers, defining the number of filters for each bottleneck in inception block
            filters -- python list of integers, defining the number of filters for each of the 4 branches applying Convolutions
            block -- string/character, used to name the layers, depending on their position in the network

            Returns:
            X -- output of the convolutional block, tensor of shape (n_H, n_W, n_C)
            """

            # defining name basis
            conv_name_base = 'block' + block + '_branch_'
            bn_name_base = 'block' + block + '_branch_'

            # Retrieve Filters
            F1, F2, F3, F4 = filters

            # Retrieve Bottlenecks
            B2, B3 = bottlenecks

            # First branch - 1 x 1 conv
            b1 = Conv3D(F1, (1, 1, 1), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'conv_1',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer_2))(X)
            b1 = BatchNormalization(axis=4, name=bn_name_base + 'bn1')(b1)
            b1 = Activation('relu')(b1)

            # Second branch - bottleneck
            b2 = Conv3D(B2, (1, 1, 1), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'bottleneck_2',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer_1))(X)
            b2 = BatchNormalization(axis=4, name=bn_name_base + 'bn2')(b2)
            b2 = Activation('relu')(b2)
            # Second branch - Convolution 3 x 3 x 3
            b2 = Conv3D(F2, (3, 3, 3), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'conv_2',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer_2))(b2)
            b2 = BatchNormalization(axis=4, name=bn_name_base + 'bn3')(b2)
            b2 = Activation('relu')(b2)

            # Third branch - bottleneck
            b3 = Conv3D(B3, (1, 1, 1), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'bottleneck_3',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer_1))(X)
            b3 = BatchNormalization(axis=4, name=bn_name_base + 'bn4')(b3)
            b3 = Activation('relu')(b3)
            # Third branch - Convolution 5 x 5 x 5
```

```
            b3 = Conv3D(F3, (5, 5, 5), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'conv_3',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer_2))(b3)
            b3 = BatchNormalization(axis=4, name=bn_name_base + 'bn5')(b3)
            b3 = Activation('relu')(b3)

            # Concatenate results of 3 branches
            X = concatenate([b1, b2, b3], axis=4)

            return X

        # Define the input placeholder as a tensor with shape input_shape.

        # Think of this as your input image!
        X_input = Input(shape=input_shape)

        # First Inception Block
        X = inception_block(X_input, filters=[8, 4, 4, 4],
                            bottlenecks=[8, 8], block='1',
                            kernel_initializer=kernel_initializer,
                            bias_initializer=bias_initializer,
                            l2_regularizer_1=space['L2_s1'],
                            l2_regularizer_2=space['L2_s2'])

        # Fully Connected Layer
        X = Flatten()(X)

        # Add extra dense layers
        num_layer = 1
        for num_layer in range(0, space['extra_dense']):
            extra_units = space['extra_units'] + 1
            X = Dense(extra_units, activation='relu', name='fc_' + str(num_layer),
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)

        if n_classes <= 2:
            X = Dense(1, activation='sigmoid', name='fc_' + str(num_layer + 1),
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)
        else:
            X = Dense(n_classes, activation='softmax', name='fc_' + str(num_layer + 1),
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)
        # Complete Model
        model = Model(inputs=X_input, outputs=X, name='GoogLeNet_Light')

        return model

    def model_googlenet_adaptative(input_shape,
                                   n_classes,
                                   space,
                                   kernel_initializer,
                                   bias_initializer):

        def inception_block(X, bottlenecks, filters, block,
                            kernel_initializer, bias_initializer,
                            l2_regularizer_1, l2_regularizer_2):
            """
            Implementation of the inception block

            Arguments:
            X -- input tensor of shape (m, n_H_prev, n_W_prev, n_C_prev)
            bottlenecks -- python list of integers, defining the number of filters for each bottleneck in inception block
            filters -- python list of integers, defining the number of filters for each of the 4 branches applying Convolutions
            block -- string/character, used to name the layers, depending on their position in the network

            Returns:
            X -- output of the convolutional block, tensor of shape (n_H, n_W, n_C)
            """

            # defining name basis
            conv_name_base = 'block' + block + '_branch_'
            bn_name_base = 'block' + block + '_branch_'

            # Retrieve Filters
            F1, F2, F3 = filters

            # Retrieve Bottlenecks
            B1, B2, B3 = bottlenecks

            # First branch - 1 x 1 conv
            b1 = Conv3D(F1, (1, 1, 1), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'conv_1',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer_2))(X)
            b1 = BatchNormalization(axis=4, name=bn_name_base + 'bn1')(b1)
            b1 = Activation('relu')(b1)

            # Second branch - bottleneck
            b2 = Conv3D(B2, (1, 1, 1), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'bottleneck_2',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer_1))(X)
            b2 = BatchNormalization(axis=4, name=bn_name_base + 'bn2')(b2)
            b2 = Activation('relu')(b2)
```

```
            # Second branch - Convolution 3 x 3 x 3
            b2 = Conv3D(F2, (3, 3, 3), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'conv_2',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer_2))(b2)
            b2 = BatchNormalization(axis=4, name=bn_name_base + 'bn3')(b2)
            b2 = Activation('relu')(b2)

            # Third branch - bottleneck
            b3 = Conv3D(B3, (1, 1, 1), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'bottleneck_3',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer_1))(X)
            b3 = BatchNormalization(axis=4, name=bn_name_base + 'bn4')(b3)
            b3 = Activation('relu')(b3)
            # Third branch - Convolution 5 x 5 x 5
            b3 = Conv3D(F3, (5, 5, 5), strides=(1, 1, 1), padding='same',
                        name=conv_name_base + 'conv_3',
                        kernel_initializer=kernel_initializer,
                        bias_initializer=bias_initializer,
                        kernel_regularizer=regularizers.l2(l2_regularizer_2))(b3)
            b3 = BatchNormalization(axis=4, name=bn_name_base + 'bn5')(b3)
            b3 = Activation('relu')(b3)

            # Concatenate results of 3 branches
            X = concatenate([b1, b2, b3], axis=4)

            return X

        # Define the input placeholder as a tensor with shape input_shape.

        # Think of this as your input image!
        X_input = Input(shape=input_shape)

        # Vary the size of filters: F1, F2, F3, F4
        # size of filters: B1, B2, B3, B4
        filters = [space['F1'], space['F2'], space['F3']]
        bottlenecks = [space['B1'], space['B2'], space['B3']]

        # Vary regularizers
        L2_s1 = space['L2_s1']
        L2_s2 = space['L2_s2']

        # Vary the number of inception blocks
        X = X_input
        for i in range(0, space['num_inception']):
            X = inception_block(X, filters=filters,
                                bottlenecks=bottlenecks, block=str(i),
                                kernel_initializer=kernel_initializer,
                                bias_initializer=bias_initializer,
                                l2_regularizer_1=L2_s1,
                                l2_regularizer_2=L2_s2)

        # Fully Connected Layer
        X = Flatten()(X)

        # Add extra dense layers
        num_layer = 1
        for num_layer in range(0, space['extra_dense']):
            extra_units = space['extra_units'] + 1
            X = Dense(extra_units, activation='relu', name='fc_' + str(num_layer),
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)

        if n_classes <= 2:
            X = Dense(1, activation='sigmoid', name='fc_' + str(num_layer + 1),
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)
        else:
            X = Dense(n_classes, activation='softmax', name='fc_' + str(num_layer + 1),
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)
        # Complete Model
        model = Model(inputs=X_input, outputs=X, name='GoogLeNet_Light')

        return model

    # ---------------------------------- LeNet 5 ----------------------------------#

    # CNN MODEL - Similiar to LeNet5
    def model_lenet_5(input_shape,
                      n_classes,
                      kernel_initializer,
                      bias_initializer,
                      use_dropout,
                      prob_dropout):

        # Define the input placeholder as a tensor with shape input_shape.
        # Think of this as your input image!
        X_input = Input(shape=input_shape)

        # CONV LAYER 1
        X = Conv3D(6, (5, 5, 5), strides=(1, 1, 1), padding='valid',
                   name='conv_0', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(X_input)
        # MAXPOOL
        X = MaxPooling3D(pool_size=(2, 2, 2),
```

```
                        strides=(2, 2, 2),
                        padding='valid',
                        name='max_pool_0')(X)
        # DROPOUT
        if use_dropout:
            X = Dropout(prob_dropout)(X)

        # CONV LAYER 2
        X = Conv3D(16, (5, 5, 5), strides=(1, 1, 1), padding='valid',
                   name='conv_1', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(X)
        # MAXPOOL
        X = MaxPooling3D(pool_size=(2, 2, 2),
                        strides=(2, 2, 2),
                        padding='valid',
                        name='max_pool_1')(X)
        # DROPOUT
        if use_dropout:
            X = Dropout(prob_dropout)(X)

        # FC LAYER 1
        # FLATTEN X (means convert it to a vector) + FULLYCONNECTED
        X = Flatten()(X)
        X = Dense(120, activation='relu', name='fc_0',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(X)

        # FC LAYER 2
        # FLATTEN X (means convert it to a vector) + FULLYCONNECTED
        X = Dense(84, activation='relu', name='fc_1',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(X)

        # FC LAYER 3
        # FLATTEN X (means convert it to a vector) + FULLYCONNECTED
        if n_classes <= 2:
            X = Dense(1, activation='sigmoid', name='fc_2',
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)
        else:
            X = Dense(n_classes, activation='softmax', name='fc_2',
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)

        # COMPLETE CONV NETWORK
        # Create model. This creates your Keras model instance,
        # you'll use this instance to train/test the model.
        model = Model(inputs=X_input, outputs=X, name='LeNet5')

        return model

    # ---------------------------------- Custom 1 ---------------------------------#

    # CNN MODEL 1
    def model_1(input_shape,
                n_classes,
                kernel_initializer,
                bias_initializer,
                use_dropout,
                prob_dropout):
        # Define the input placeholder as a tensor with shape input_shape.
        # Think of this as your input image!
        X_input = Input(shape=input_shape)

        # CONV LAYER 1
        # CONV -> BN -> RELU Block applied to X
        X = Conv3D(32, (3, 3, 3), strides=(1, 1, 1),
                   name='conv_0', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(X_input)
        X = BatchNormalization(axis=4, name='bn_0')(X)
        X = Activation('relu')(X)
        # MAXPOOL
        X = MaxPooling3D(pool_size=(2, 2, 2),
                        strides=(2, 2, 2),
                        padding='valid',
                        name='max_pool_0')(X)
        # DROPOUT
        if use_dropout:
            X = Dropout(prob_dropout)(X)

        # CONV LAYER 2
        # CONV -> BN -> RELU Block applied to X
        X = Conv3D(32, (3, 3, 3),
                   strides=(1, 1, 1),
                   name='conv_1',
                   kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(X)
        X = BatchNormalization(axis=4, name='bn_1')(X)
        X = Activation('relu')(X)
        # MAXPOOL
        X = MaxPooling3D(pool_size=(2, 2, 2),
                        strides=(2, 2, 2),
                        padding='valid',
                        name='max_pool_1')(X)
        # DROPOUT
        if use_dropout:
            X = Dropout(prob_dropout)(X)

        # CONV LAYER 3
```

```
        # CONV -> BN -> RELU Block applied to X
        X = Conv3D(32, (3, 3, 3), strides=(1, 1, 1),
                   name='conv_2', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(X)
        X = BatchNormalization(axis=4, name='bn_2')(X)
        X = Activation('relu')(X)
        # MAXPOOL
        X = MaxPooling3D(pool_size=(2, 2, 2),
                         strides=(2, 2, 2),
                         padding='valid',
                         name='max_pool_2')(X)
        # DROPOUT
        if use_dropout:
            X = Dropout(prob_dropout)(X)

        # FC LAYER 1
        # FLATTEN X (means convert it to a vector) + FULLYCONNECTED
        X = Flatten()(X)

        if n_classes <= 2:
            X = Dense(1, activation='sigmoid', name='fc',
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)
        else:
            X = Dense(n_classes, activation='softmax', name='fc',
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)

        # COMPLETE CONV NETWORK
        # Create model. This creates your Keras model instance,
        # you'll use this instance to train/test the model.
        model = Model(inputs=X_input, outputs=X, name='Model_1')

        return model

    # ---------------------------------- Custom 2 -------------------------------#

    # CNN MODEL 2
    def model_2(input_shape,
                n_classes,
                kernel_initializer,
                bias_initializer,
                use_dropout):

        # Define the input placeholder as a tensor with shape input_shape.
        # Think of this as your input image!
        X_input = Input(shape=input_shape)

        # CONV LAYER 1
        X = Conv3D(32, (3, 3, 3),
                   strides=(1, 1, 1),
                   name='conv_0', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(X_input)
        X = BatchNormalization(axis=4, name='bn_0')(X)
        X = Activation('relu')(X)
        # MAXPOOL
        X = MaxPooling3D(pool_size=(2, 2, 2),
                         strides=(1, 1, 1),
                         padding='valid',
                         name='max_pool_0')(X)
        # DROPOUT
        if use_dropout:
            X = Dropout(0.2)(X)

        # CONV LAYER 2
        # CONV -> BN -> RELU Block applied to X
        X = Conv3D(32, (3, 3, 3),
                   strides=(1, 1, 1),
                   name='conv_1',
                   kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(X)
        X = BatchNormalization(axis=4, name='bn_1')(X)
        X = Activation('relu')(X)
        # MAXPOOL
        X = MaxPooling3D(pool_size=(2, 2, 2),
                         strides=(1, 1, 1),
                         padding='valid',
                         name='max_pool_1')(X)
        # DROPOUT
        if use_dropout:
            X = Dropout(0.3)(X)

        # CONV LAYER 3
        # CONV -> BN -> RELU Block applied to X
        X = Conv3D(16, (2, 2, 2),
                   strides=(1, 1, 1),
                   name='conv_2',
                   kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(X)
        X = BatchNormalization(axis=4, name='bn_2')(X)
        X = Activation('relu')(X)
        # MAXPOOL
        X = MaxPooling3D(pool_size=(2, 2, 2),
                         strides=(1, 1, 1),
                         padding='valid',
                         name='max_pool_2')(X)
        # DROPOUT
        if use_dropout:
            X = Dropout(0.4)(X)
```

```
    # CONV LAYER 4
    # CONV -> BN -> RELU Block applied to X
    X = Conv3D(8, (2, 2, 2),
               strides=(1, 1, 1),
               name='conv_3',
               kernel_initializer=kernel_initializer,
               bias_initializer=bias_initializer)(X)
    X = BatchNormalization(axis=4, name='bn_3')(X)
    X = Activation('relu')(X)
    # MAXPOOL
    X = MaxPooling3D(pool_size=(1, 1, 1),
                     strides=(1, 1, 1),
                     padding='valid',
                     name='max_pool_3')(X)
    # DROPOUT
    if use_dropout:
        X = Dropout(0.5)(X)

    # CONV LAYER 5
    X = Conv3D(4, (2, 2, 2),
               strides=(1, 1, 1),
               activation="relu",
               name="last_conv")(X)
    X = BatchNormalization(axis=4, name='last_bn')(X)
    X = Activation('relu')(X)
    # MAXPOOL
    X = MaxPooling3D(pool_size=(1, 1, 1),
                     strides=(1, 1, 1),
                     padding='valid',
                     name='last_max_pool')(X)
    # FC LAYER 1
    # FLATTEN X (means convert it to a vector) + FULLYCONNECTED
    X = Flatten()(X)

    if n_classes <= 2:
        X = Dense(1, activation='sigmoid',
                  name='fc',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(X)
    else:
        X = Dense(n_classes, activation='softmax',
                  name='fc',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(X)

    # COMPLETE CONV NETWORK
    # Create model. This creates your Keras model instance,
    # you'll use this instance to train/test the model.
    model = Model(inputs=X_input, outputs=X, name='Model_2')

    return model

# ---------------------------------- Custom 3 ----------------------------------#

# CNN MODEL 3
def model_3(input_shape,
            n_classes,
            kernel_initializer,
            bias_initializer,
            use_dropout,
            prob_dropout,
            space=None):
    # Define the input placeholder as a tensor with shape input_shape.
    # Think of this as your input image!
    X_input = Input(shape=input_shape)

    # CONV LAYER 1
    # CONV -> BN -> RELU Block applied to X
    X = Conv3D(32, (3, 3, 3), strides=(1, 1, 1),
               name='conv_0', kernel_initializer=kernel_initializer,
               bias_initializer=bias_initializer)(X_input)
    X = BatchNormalization(axis=4, name='bn_0')(X)
    X = Activation('relu')(X)
    # MAXPOOL
    X = MaxPooling3D(pool_size=(2, 2, 2),
                     strides=(1, 2, 2),
                     padding='valid',
                     name='max_pool_0')(X)
    # DROPOUT
    if use_dropout:
        X = Dropout(0.4)(X)

    # CONV LAYER 2
    # CONV -> BN -> RELU Block applied to X
    X = Conv3D(64, (3, 3, 3),
               strides=(1, 1, 1),
               name='conv_1',
               kernel_initializer=kernel_initializer,
               bias_initializer=bias_initializer)(X)
    X = BatchNormalization(axis=4, name='bn_1')(X)
    X = Activation('relu')(X)
    # MAXPOOL
    X = MaxPooling3D(pool_size=(2, 2, 2),
                     strides=(2, 2, 2),
                     padding='valid',
                     name='max_pool_1')(X)
    # DROPOUT
    if use_dropout:
```

```
            X = Dropout(0.5)(X)

        # CONV LAYER 3
        # CONV -> BN -> RELU Block applied to X
        X = Conv3D(128, (3, 3, 3), strides=(1, 1, 1),
                   name='conv_2', kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(X)
        X = BatchNormalization(axis=4, name='bn_2')(X)
        X = Activation('relu')(X)
        # MAXPOOL
        X = MaxPooling3D(pool_size=(2, 2, 2),
                         strides=(2, 2, 2),
                         padding='valid',
                         name='max_pool_2')(X)
        # DROPOUT
        if use_dropout:
            X = Dropout(prob_dropout)(X)

        # CONV LAYER 4
        # CONV -> BN -> RELU Block applied to X
        X = Conv3D(128, (1, 1, 1),
                   strides=(1, 1, 1),
                   name='conv_3',
                   kernel_initializer=kernel_initializer,
                   bias_initializer=bias_initializer)(X)
        X = Activation('relu')(X)
        # MAXPOOL
        X = MaxPooling3D(pool_size=(1, 1, 1),
                         strides=(1, 1, 1),
                         padding='valid',
                         name='max_pool_3')(X)
        # DROPOUT
        if use_dropout:
            X = Dropout(0.6)(X)

        # CONV LAYER 5
        X = Conv3D(256, (1, 1, 1),
                   strides=(1, 1, 1),
                   activation="relu",
                   name="last_conv")(X)
        X = Activation('relu')(X)
        # MAXPOOL
        X = MaxPooling3D(pool_size=(1, 1, 1),
                         strides=(1, 1, 1),
                         padding='valid',
                         name='last_max_pool')(X)

        # FC LAYER 1
        # FLATTEN X (means convert it to a vector) + FULLYCONNECTED
        X = Flatten()(X)

        if n_classes <= 2:
            X = Dense(1, activation='sigmoid', name='fc',
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)
        else:
            X = Dense(n_classes, activation='softmax', name='fc',
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)

        # COMPLETE CONV NETWORK
        # Create model. This creates your Keras model instance, you'll use
        # this instance to train/test the model.
        model = Model(inputs=X_input, outputs=X, name='Model_1')

        return model

    # ---------------------------------- Adaptative 1 ----------------------------------#

    def model_adaptative_1(input_shape,
                           n_classes,
                           space,
                           kernel_initializer,
                           bias_initializer):

        def convolution_block(X, stage, num_block,
                              filter_size, num_filters,
                              l2_regularizer,
                              kernel_initializer,
                              bias_initializer,
                              padding):

            conv_name = 'conv_' + str(stage) + '_' + str(num_block)
            pool_name = 'pool' + str(stage) + str(num_block)
            bn_name = 'bn' + str(stage) + str(num_block)

            X = Conv3D(num_filters, filter_size,
                       strides=1,
                       padding=padding,
                       name=conv_name,
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       kernel_regularizer=regularizers.l2(l2_regularizer))(X)

            X = BatchNormalization(axis=4, name=bn_name)(X)

            X = Activation('relu')(X)

            X = MaxPooling3D(pool_size=3,
```

```
                      strides=1,
                      padding=padding,
                      name=pool_name)(X)
    return X

# Get number of convolutional blocks per stage
num_conv_1 = space['num_conv_1']
num_conv_2 = space['num_conv_2']
num_conv_3 = space['num_conv_3']
num_conv_4 = space['num_conv_4']

# ---------------------------- START ---------------------------------

# Input placeholder as a tensor with shape input_shape.
X_input = Input(shape=input_shape)
X = X_input

l2_regularizer_0 = 0
stage = '0'
num_block = '0'

X = convolution_block(X, stage,
                      num_block, 3, 32,
                      l2_regularizer_0,
                      kernel_initializer,
                      bias_initializer,
                      padding='same')

# ---------------------------- STAGE 1 -------------------------------
F1 = space['F1']
l2_regularizer_1 = space['l2_regularizer_1']
stage = '1'
filter_size_1 = space['filter_size_1']

for num_block in range(0, num_conv_1):
    X = convolution_block(X, stage,
                          num_block, filter_size_1, F1,
                          l2_regularizer_1,
                          kernel_initializer,
                          bias_initializer,
                          padding='same')

# ---------------------------- STAGE 2 -------------------------------
F2 = space['F2']
l2_regularizer_2 = space['l2_regularizer_2']
stage = '2'
filter_size_2 = space['filter_size_2']

for num_block in range(0, num_conv_2):
    X = convolution_block(X, stage,
                          num_block, filter_size_2, F2,
                          l2_regularizer_2,
                          kernel_initializer,
                          bias_initializer,
                          padding='same')

# ---------------------------- STAGE 3 -------------------------------
F3 = space['F3']
l2_regularizer_3 = space['l2_regularizer_3']
stage = '3'
filter_size_3 = space['filter_size_3']

for num_block in range(0, num_conv_3):
    X = convolution_block(X, stage,
                          num_block, filter_size_3, F3,
                          l2_regularizer_3,
                          kernel_initializer,
                          bias_initializer,
                          padding='valid')

# ---------------------------- STAGE 4 -------------------------------
F4 = space['F4']
l2_regularizer_4 = space['l2_regularizer_4']
stage = '4'
filter_size_4 = space['filter_size_4']

for num_block in range(0, num_conv_4):
    X = convolution_block(X, stage,
                          num_block, filter_size_4, F4,
                          l2_regularizer_4,
                          kernel_initializer,
                          bias_initializer,
                          padding='valid')

# ------------------------- AVERAGE POOL -------------------------------
if space['avg_pool']:
    X = AveragePooling3D(2, name='avg_pool')(X)

# ------------------------- FULLY CONNECTED ----------------------------
X = Flatten()(X)

# Add extra dense layers
for num_layer in range(0, space['extra_dense']):
    extra_units = space['extra_units'] + 1
    X = Dense(extra_units, activation='relu', name='fc_' + str(num_layer),
              kernel_initializer=kernel_initializer,
              bias_initializer=bias_initializer)(X)

if n_classes <= 2:
```

```
            X = Dense(1, activation='sigmoid', name='fc_final',
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)
        else:
            X = Dense(n_classes, activation='softmax', name='fc_final',
                      kernel_initializer=kernel_initializer,
                      bias_initializer=bias_initializer)(X)

        model = Model(inputs=X_input, outputs=X, name='Model_adptative_1')

        return model

    # ---------------------------------- ResNet-Inception V2 ----------------------------------#

    def model_resnetinception_v2_adaptative(self,
                                            input_shape,
                                            n_classes,
                                            kernel_initializer,
                                            bias_initializer,
                                            space=None,
                                            include_top=True,
                                            pooling=None):

        """Instantiates the Inception-ResNet v2 architecture.

        # Arguments
            input_shape: optional shape tuple, only to be specified
            include_top: whether to include the fully-connected layer at the top of the network.
            pooling: Optional pooling mode for feature extraction
                when `include_top` is `False`.
                - `None` means that the output of the model will be
                    the 4D tensor output of the last convolutional layer.
                - `'avg'` means that global average pooling
                    will be applied to the output of the
                    last convolutional layer, and thus
                    the output of the model will be a 2D tensor.
                - `'max'` means that global max pooling will be applied.
            classes: optional number of classes to classify images
                into, only to be specified if `include_top` is `True`, and
                if no `weights` argument is specified.

        # Returns
            A Keras `Model` instance.

        # Raises
            ValueError: in case of invalid argument for `weights`,
                or invalid input shape.
        """

        # -------------------------- Convolution + Batch Normalization -------------------------#

        def conv3d_bn(x,
                      filters,
                      kernel_size,
                      kernel_initializer,
                      bias_initializer,
                      l2_regularizer = 0,
                      strides=1,
                      padding='same',
                      activation='relu',
                      use_bias=False,
                      name=None):

            """Utility function to apply conv + BN.

            # Arguments
                x: input tensor.
                filters: filters in `Conv3D`.
                kernel_size: kernel size as in `Conv3D`.
                strides: strides in `Conv3D`.
                padding: padding mode in `Conv3D`.
                activation: activation in `Conv3D`.
                use_bias: whether to use a bias in `Conv3D`.
                name: name of the ops; will become `name + '_ac'` for the activation
                    and `name + '_bn'` for the batch norm layer.

            # Returns
                Output tensor after applying `Conv3D` and `BatchNormalization`.
            """

            x = Conv3D(filters,
                       kernel_size,
                       kernel_initializer=kernel_initializer,
                       bias_initializer=bias_initializer,
                       strides=strides,
                       padding=padding,
                       use_bias=use_bias,
                       name=name)(x)

            if not use_bias:
                bn_axis = 4
                bn_name = None if name is None else name + '_bn'
                x = BatchNormalization(axis=bn_axis,
                                       scale=False,
                                       name=bn_name)(x)

            if activation is not None:
                ac_name = None if name is None else name + '_ac'
                x = Activation(activation, name=ac_name)(x)
```

```
    return x

# ------------------------- Inception - ResNet Block ------------------------#

def inception_resnet_block(x,
                           scale,
                           block_type,
                           block_idx,
                           kernel_initializer,
                           bias_initializer,
                           l2_regularizer,
                           activation='relu'):

    """Adds a Inception-ResNet block.

    This function builds 3 types of Inception-ResNet blocks mentioned
    in the paper, controlled by the `block_type` argument (which is the
    block name used in the official TF-slim implementation):
        - Inception-ResNet-A: `block_type='block35'`
        - Inception-ResNet-B: `block_type='block17'`
        - Inception-ResNet-C: `block_type='block8'`

    # Arguments
        x: input tensor.
        scale: scaling factor to scale the residuals (i.e., the output of
            passing `x` through an inception module) before adding them
            to the shortcut branch.
            Let `r` be the output from the residual branch,
            the output of this block will be `x + scale * r`.
        block_type: `'block35'`, `'block17'` or `'block8'`, determines
            the network structure in the residual branch.
        block_idx: an `int` used for generating layer names.
            The Inception-ResNet blocks
            are repeated many times in this network.
            We use `block_idx` to identify
            each of the repetitions. For example,
            the first Inception-ResNet-A block
            will have `block_type='block35', block_idx=0`,
            and the layer names will have
            a common prefix `'block35_0'`.
        activation: activation function to use at the end of the block
            (see [activations](../activations.md)).
            When `activation=None`, no activation is applied
            (i.e., "linear" activation: `a(x) = x`).

    # Returns
        Output tensor for the block.

    # Raises
        ValueError: if `block_type` is not one of `'block35'`,
            `'block17'` or `'block8'`.
    """

    if block_type == 'block35':
        branch_0 = conv3d_bn(x, 32, 1, kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branch_1 = conv3d_bn(x, 32, 1, kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branch_1 = conv3d_bn(branch_1, 32, 3, kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branch_2 = conv3d_bn(x, 32, 1, kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branch_2 = conv3d_bn(branch_2, 48, 3, kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branch_2 = conv3d_bn(branch_2, 64, 3, kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branches = [branch_0, branch_1, branch_2]
    elif block_type == 'block17':
        branch_0 = conv3d_bn(x, 32, 1, kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branch_1 = conv3d_bn(x, 16, 1, kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branch_1 = conv3d_bn(branch_1, 24, [1, 1, 7], kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branch_1 = conv3d_bn(branch_1, 32, [7, 1, 1], kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branches = [branch_0, branch_1]
    elif block_type == 'block8':
        branch_0 = conv3d_bn(x, 32, 1, kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branch_1 = conv3d_bn(x, 32, 1, kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branch_1 = conv3d_bn(branch_1, 46, [1, 1, 3], kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branch_1 = conv3d_bn(branch_1, 64, [3, 1, 1], kernel_initializer, bias_initializer, l2_regularizer=l2_regularizer)
        branches = [branch_0, branch_1]
    else:
        raise ValueError('Unknown Inception-ResNet block type. '
                         'Expects "block35", "block17" or "block8", '
                         'but got: ' + str(block_type))

    block_name = block_type + '_' + str(block_idx)

    channel_axis = 4

    mixed = Concatenate(axis=channel_axis, name=block_name + '_mixed')(branches)

    up = conv3d_bn(mixed,
                   backend.int_shape(x)[channel_axis],
                   1,
                   kernel_initializer,
                   bias_initializer,
                   activation=None,
                   use_bias=True,
                   name=block_name + '_conv')

    x = Lambda(lambda inputs, scale: inputs[0] + inputs[1] * scale,
               output_shape=backend.int_shape(x)[1:],
               arguments={'scale': scale},
               name=block_name)([x, up])
```

```
    if activation is not None:
        x = Activation(activation, name=block_name + '_ac')(x)
    return x

# ------------------------- Start of CNN Network ------------------------#

x_input = Input(shape=input_shape)
x = x_input
channel_axis = 4

#  Get space parameters
num_block35 = space['num_block35']
num_block17 = space['num_block17']
num_block8 = space['num_block8']
l2_reg_block8 = space['l2_reg_block8']
l2_reg_block17 = space['l2_reg_block17']
l2_reg_block35 = space['l2_reg_block35']
dropout = space['dropout']

# Stem block: 11 x 11 x 11 x 32
x = conv3d_bn(x, 8, 3, kernel_initializer, bias_initializer, padding='valid')
x = conv3d_bn(x, 8, 3, kernel_initializer, bias_initializer, padding='valid')
x = conv3d_bn(x, 16, 3, kernel_initializer, bias_initializer)
x = MaxPooling3D(3, strides=1)(x)
x = conv3d_bn(x, 16, 1, kernel_initializer, bias_initializer, padding='valid')
x = conv3d_bn(x, 32, 3, kernel_initializer, bias_initializer, padding='valid')
x = MaxPooling3D(3, strides=2)(x)

# Mixed 5b (Inception-A block): 11 x 11 x 11 x 80
branch_0 = conv3d_bn(x, 32, 1, kernel_initializer, bias_initializer)
branch_1 = conv3d_bn(x, 16, 1, kernel_initializer, bias_initializer)
branch_1 = conv3d_bn(branch_1, 20, 5, kernel_initializer, bias_initializer)
branch_2 = conv3d_bn(x, 16, 1, kernel_initializer, bias_initializer)
branch_2 = conv3d_bn(branch_2, 20, 3, kernel_initializer, bias_initializer)
branch_2 = conv3d_bn(branch_2, 20, 3, kernel_initializer, bias_initializer)
branch_pool = AveragePooling3D(3, strides=1, padding='same')(x)
branch_pool = conv3d_bn(branch_pool, 20, 1, kernel_initializer, bias_initializer)
branches = [branch_0, branch_1, branch_2, branch_pool]
x = Concatenate(axis=channel_axis, name='mixed_5b')(branches)

# 10x block35 (Inception-ResNet-A block): 11 x 11 x 11 x 80
for block_idx in range(0, num_block35):
    x = inception_resnet_block(x,
                               scale=0.17,
                               block_type='block35',
                               block_idx=block_idx,
                               kernel_initializer=kernel_initializer,
                               bias_initializer=bias_initializer,
                               l2_regularizer=l2_reg_block35)

# Mixed 6a (Reduction-A block): 9 x 9 x 9 x 160
branch_0 = conv3d_bn(x, 60, 3, kernel_initializer, bias_initializer, strides=1, padding='valid')
branch_1 = conv3d_bn(x, 40, 1, kernel_initializer, bias_initializer)
branch_1 = conv3d_bn(branch_1, 40, 3, kernel_initializer, bias_initializer)
branch_1 = conv3d_bn(branch_1, 40, 3, kernel_initializer, bias_initializer, strides=1, padding='valid')
branch_pool = MaxPooling3D(3, strides=1, padding='valid')(x)
branches = [branch_0, branch_1, branch_pool]
x = Concatenate(axis=channel_axis, name='mixed_6a')(branches)

# 20x block17 (Inception-ResNet-B block): 9 x 9 x 9 x 160
for block_idx in range(num_block17):
    x = inception_resnet_block(x,
                               scale=0.1,
                               block_type='block17',
                               block_idx=block_idx,
                               kernel_initializer=kernel_initializer,
                               bias_initializer=bias_initializer,
                               l2_regularizer=l2_reg_block17)

# Mixed 7a (Reduction-B block): 7 x 7 x 7 x 280
branch_0 = conv3d_bn(x, 20, 1, kernel_initializer, bias_initializer)
branch_0 = conv3d_bn(branch_0, 40, 3, kernel_initializer, bias_initializer, strides=1, padding='valid')
branch_1 = conv3d_bn(x, 20, 1, kernel_initializer, bias_initializer)
branch_1 = conv3d_bn(branch_1, 40, 3, kernel_initializer, bias_initializer, strides=1, padding='valid')
branch_2 = conv3d_bn(x, 20, 1, kernel_initializer, bias_initializer)
branch_2 = conv3d_bn(branch_2, 35, 3, kernel_initializer, bias_initializer)
branch_2 = conv3d_bn(branch_2, 40, 3, kernel_initializer, bias_initializer, strides=1, padding='valid')
branch_pool = MaxPooling3D(3, strides=1, padding='valid')(x)
branches = [branch_0, branch_1, branch_2, branch_pool]
x = Concatenate(axis=channel_axis, name='mixed_7a')(branches)

# 10x block8 (Inception-ResNet-C block): 7 x 7 x 7 x 280
for block_idx in range(num_block8):
    x = inception_resnet_block(x,
                               scale=0.2,
                               block_type='block8',
                               block_idx=block_idx,
                               kernel_initializer=kernel_initializer,
                               bias_initializer=bias_initializer,
                               l2_regularizer=l2_reg_block8)
x = inception_resnet_block(x,
                           scale=1.,
                           activation=None,
                           block_type='block8',
                           block_idx=10,
                           kernel_initializer=kernel_initializer,
                           bias_initializer=bias_initializer,
                           l2_regularizer=l2_reg_block8)
```

```
        # Final convolution block: 7 x 7 x 7 x 140
        x = conv3d_bn(x, 140, 1, kernel_initializer, bias_initializer, name='conv_7b')

        if include_top:
            # Classification block
            x = GlobalAveragePooling3D(name='avg_pool')(x)
            x = Dropout(dropout)(x)

            if n_classes <= 2:
                x = Dense(1, activation='sigmoid', name='fc_final',
                          kernel_initializer=kernel_initializer,
                          bias_initializer=bias_initializer)(x)
            else:
                x = Dense(n_classes, activation='softmax', name='fc_final',
                          kernel_initializer=kernel_initializer,
                          bias_initializer=bias_initializer)(x)

        else:
            if pooling == 'avg':
                x = GlobalAveragePooling3D()(x)
            elif pooling == 'max':
                x = GlobalMaxPooling3D()(x)

        # Create model.
        model = Model(inputs=x_input, outputs=x, name='inception_resnet_v2')

        return model
```

DNN_Models.py

```
from keras.models import Model
from keras.layers import Input, Dense, Dropout
from keras import regularizers

class Bayesian:

    def __init__(self):
        pass

    # CNN MODEL - Similiar to Vgg - 16
    def model_adaptative_1(input_shape,
                           space,
                           n_blocks,
                           kernel_initializer,
                           bias_initializer):

        # Block of m layers and n units
        def dnn_block(x_i, num_block):

            l2_regularizer = space['l2_regularizer_{0}'.format(num_block)]
            use_dropout = space['use_dropout']

            for num_layer in range(0, space['dense_{0}'.format(num_block)]):
                extra_units = space['units_{0}'.format(num_block)] + 1
                x_i = Dense(extra_units,
                            activation='relu',
                            use_bias=True,
                            name='fc_' + 'block_{0}_'.format(num_block) + str(num_layer),
                            kernel_initializer=kernel_initializer,
                            bias_initializer=bias_initializer,
                            kernel_regularizer=regularizers.l2(l2_regularizer))(x_i)

            if use_dropout:
                x_i = Dropout(space['dropout'])(x_i)

            return x_i

        # Input placeholder as a tensor with shape input_shape.
        x_input = Input(shape=input_shape)
        x = x_input

        # Create n blocks of dense layers
        for i in range(n_blocks):
            x = dnn_block(x, str(i))

        # Final layer
        x = Dense(1, activation='sigmoid',
                  name='fc_final',
                  kernel_initializer=kernel_initializer,
                  bias_initializer=bias_initializer)(x)

        return Model(inputs=x_input, outputs=x, name='Model_adptative_1')
```

cnn_gan.py

```
from __future__ import print_function, division

from keras.layers import Input, Dense, Reshape, Flatten, Dropout
from keras.layers import BatchNormalization, Activation
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling3D, Conv3D
from keras.models import Sequential, Model
from keras.optimizers import Adam

import os
import math
import numpy as np
import matplotlib.pyplot as plt

# ------------------------------------------------ INPUTS ------------------------------------------------

train_epochs = 5000
train_batch_size = 32
saving_interval = 100
model_version = 2
smooth_labeling = True

detector = 'Nodule'
nodule_class = 'nodules'
threshold = 2
radiologist = 3

lr = 0.0002
beta_1 = 0.5
beta_2 = 0.999

# ------------------------------------------------ Folders ------------------------------------------------

train_path = 'E:/Base de datos maestría/LIDC-IDRI/Train Data/Train Data {0} Detector/{0} Detector 2 ' \
             'Classes Generator GANS/'.format(detector) + nodule_class + '/'

saving_path = 'E:/Base de datos maestría/LIDC-IDRI/Models/Models {0} Detector/Models of {0} 2 Classes ' \
              'GAN/Model {1} {2}/'.format(detector, model_version, nodule_class)

saving_gen_path = saving_path + nodule_class + '_generated_gan/'

# Check path existence
if not os.path.isdir(saving_gen_path):
    os.makedirs(saving_gen_path)

if not os.path.isdir(train_path):
    os.makedirs(train_path)


# ------------------------------------------------ CLASSES ------------------------------------------------

class DCGAN:

    def __init__(self):
        # Input shape
        self.img_rows = 32
        self.img_cols = 32
        self.img_depth = 32
        self.channels = 1
        self.img_shape = (self.img_rows, self.img_cols, self.img_depth, self.channels)
        self.latent_dim = 100
        self.discriminator = None
        self.generator = None
        self.combined = None
        self.generator_losses = []
        self.discriminator_losses_real = []
        self.discriminator_losses_fake = []
        self.discriminator_accuracy_fake = []
        self.discriminator_accuracy_real = []

    def build_model(self):
        optimizer = Adam(lr=lr, beta_1=beta_1, beta_2=beta_2)

        # Build and compile the discriminator
        self.discriminator = self.build_discriminator()
        self.discriminator.compile(loss='binary_crossentropy',
                                   optimizer=optimizer,
                                   metrics=['accuracy'])

        # Build the generator
        self.generator = self.build_generator()

        # The generator takes noise as input and generates imgs
        input_generator = Input(shape=(self.latent_dim,))
        img = self.generator(input_generator)

        # For the combined model we will only train the generator
        self.discriminator.trainable = False
```

```
        # The discriminator takes generated images as input and determines validity
        result_discriminator = self.discriminator(img)

        # The combined model  (stacked generator and discriminator)
        # Trains the generator to fool the discriminator
        self.combined = Model(input_generator, result_discriminator)
        self.combined.summary()
        self.combined.compile(loss='binary_crossentropy', optimizer=optimizer)

    def build_generator(self):

        model = Sequential()

        model.add(Dense(128 * 8 * 8 * 8, activation="relu", input_dim=self.latent_dim))
        model.add(Reshape((8, 8, 8, 128)))
        model.add(UpSampling3D())
        model.add(Conv3D(128, kernel_size=3, padding="same"))
        model.add(BatchNormalization(momentum=0.8))
        model.add(Activation("relu"))
        model.add(UpSampling3D())
        model.add(Conv3D(64, kernel_size=3, padding="same"))
        model.add(BatchNormalization(momentum=0.8))
        model.add(Activation("relu"))
        model.add(Conv3D(self.channels, kernel_size=3, padding="same"))
        model.add(Activation("tanh"))

        model.summary()

        noise = Input(shape=(self.latent_dim,))
        img = model(noise)

        return Model(noise, img)

    def build_discriminator(self):

        model = Sequential()

        model.add(Conv3D(32, kernel_size=3, strides=2, input_shape=self.img_shape, padding="same"))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))
        model.add(Conv3D(64, kernel_size=3, strides=2, padding="same"))
        model.add(BatchNormalization(momentum=0.8))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))
        model.add(Conv3D(128, kernel_size=3, strides=2, padding="same"))
        model.add(BatchNormalization(momentum=0.8))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))
        model.add(Conv3D(256, kernel_size=3, strides=1, padding="same"))
        model.add(BatchNormalization(momentum=0.8))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))
        model.add(Flatten())
        model.add(Dense(1, activation='sigmoid'))

        model.summary()

        img = Input(shape=self.img_shape)
        validity = model(img)

        return Model(img, validity)

    def train(self, epochs, batch_size=128, save_interval=10):

        # Load the dataset
        x_train = np.array([np.load(train_path + '/' + nod) for nod in os.listdir(train_path) if nod.endswith('.npy')])
        x_train = np.expand_dims(x_train, axis=4)

        # Adversarial ground truths
        valid = np.ones((batch_size, 1))
        fake = np.zeros((batch_size, 1))

        for epoch in range(epochs):

            # ---------------------
            #  Train Discriminator
            # ---------------------

            # Select a random half of images
            idx = np.random.randint(0, x_train.shape[0], batch_size)
            imgs = x_train[idx]

            # Sample noise and generate a batch of new images
            noise = np.random.normal(0, 1, (batch_size, self.latent_dim))
            gen_imgs = self.generator.predict(noise)

            # Adversarial ground truths change labels on each pair of epochs smoothing labels
            if smooth_labeling:
                valid = np.random.uniform(0.7, 1, (batch_size, 1))
```

```
                fake = np.random.uniform(0, 0.3, (batch_size, 1))

            # Train the discriminator (real classified as ones and generated as zeros)
            d_loss_real = self.discriminator.train_on_batch(imgs, valid)
            d_loss_fake = self.discriminator.train_on_batch(gen_imgs, fake)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
            self.discriminator_losses_real.append(d_loss_real[0])
            self.discriminator_losses_fake.append(d_loss_fake[0])
            self.discriminator_accuracy_real.append(d_loss_real[1])
            self.discriminator_accuracy_fake.append(d_loss_fake[1])

            # ---------------------
            #  Train Generator
            # ---------------------

            # Train the generator (wants discriminator to mistake images as real)
            g_loss = self.combined.train_on_batch(noise, valid)
            self.generator_losses.append(g_loss)

            # Plot the progress
            print("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" % (epoch, d_loss[0], 100 * d_loss[1], g_loss))

            # If at save interval => save generated image samples
            if epoch % save_interval == 0:
                self.save_imgs(epoch)
                self.save_losses(epoch)

        self.generator.save_weights(saving_path + 'generator_weights.h5')
        self.discriminator.save_weights(saving_path + 'discriminator_weights.h5')

    def load_model(self, model_path):
        self.build_model()
        self.generator.load_weights(model_path + 'generator_weights.h5')
        self.discriminator.load_weights(model_path + 'discriminator_weights.h5')

    def generate_data(self, num_data):
        noise = np.random.normal(0, 1, (num_data, self.latent_dim))
        return self.generator.predict(noise)

    def save_imgs(self, epoch):

        # Generate 100 images
        r, c = 10, 10
        noise = np.random.normal(0, 1, (r * c, self.latent_dim))
        gen_imgs = self.generator.predict(noise)

        # Create saving folder if it doesn't
        if not os.path.isdir(saving_gen_path):
            os.makedirs(saving_gen_path)

        # Show 25 slices
        fig, axs = plt.subplots(r, c)
        cnt = 0
        for i in range(r):
            for j in range(c):
                axs[i, j].imshow(gen_imgs[cnt, math.floor(gen_imgs.shape[1] / 2), :, :, 0], cmap='bone')
                axs[i, j].axis('off')
                cnt += 1
        fig.savefig(saving_gen_path + "epoch %d.png" % epoch)
        plt.close()

    def save_losses(self, epoch):

        f, ax = plt.subplots()
        ax.set_title('Generator and Discriminator Losses')
        ax.set_xlabel('epochs')
        ax.set_ylabel('Loss')
        plt.plot(self.generator_losses)
        plt.plot(self.discriminator_losses_real)
        plt.plot(self.discriminator_losses_fake)
        plt.gca().legend(('Generator Loss', 'Discriminator Loss - Real', 'Discriminator Loss - Fake'))
        f.savefig(saving_gen_path + "loss %d.png" % epoch)
        plt.close()

    @staticmethod
    def save_parameters():

        # Save all the specified parameters of this training in a '.txt' file
        parameters_folder = (saving_path + '/parameters.txt')

        f = open(parameters_folder, 'w')
        f.write('GENERAL PARAMETERS\n')
        f.write('   number of radiologists: {}\n'.format(str(radiologist)))
        f.write('   threshold: {}\n'.format(str(threshold)))
        f.write('   train_epochs: {}\n'.format(str(train_epochs)))
        f.write('   train_batch_size: {}\n'.format(str(train_batch_size)))
        f.write('   smooth_labeling: {}\n'.format(str(smooth_labeling)))
        f.write('\n')
        f.write('OPTIMIZER PARAMETERS\n')
        f.write('   lr: {}\n'.format(str(lr)))
```

```
        f.write('   beta1: {}\n'.format(str(beta_1)))
        f.write('   beta2: {}\n'.format(str(beta_2)))
        f.write('\n')


# ------------------------------------------------ MAIN ------------------------------------------------

if __name__ == '__main__':
    dcgan = DCGAN()
    dcgan.save_parameters()
    dcgan.build_model()
    dcgan.train(epochs=train_epochs, batch_size=train_batch_size, save_interval=saving_interval)
```

create_model_3D_generative_bayesian_cross_validation.py

```
from hyperopt import hp, fmin, tpe, STATUS_OK, Trials
from classes_lidc import custom_Keras_Metrics, dataTransformer, NoduleGenerator, MalignancyGenerator
from CNN_Models import Bayesian
from keras import initializers
import time
from keras.optimizers import Adam, Adagrad, SGD
from keras.utils import plot_model
import pickle
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import roc_curve, auc
import pylab as pl
import pandas as pd
import os
from sklearn.model_selection import StratifiedKFold
from cnn_gan import DCGAN
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras.callbacks import TensorBoard
from keras.models import load_model
from random import shuffle
import gc
import autopy
import math

# ------------------------------GENERAL INPUTS -------------------------------#

# How many models to test during optimization phase
max_evals = 7
num_model = 'adaptative_1'
version = '8_3'

# Data inputs
detector = 'FP Reduction'

# OTHER DETECTORS
positive_data = 'nodules'
negative_data = 'non_nodules'

# FP REDUCTION
positive_data = 'TP_model_{0}_version_8_2'.format(num_model)
negative_data = 'FP_model_{0}_version_8'.format(num_model, version)

take_screen_shot = True

# Cross validation
cv_validate = False
cv_search = True
k_folds = 10
cv = cv_validate or cv_search

# Use generator
use_generator = False
data_source = 'more_than_1_radiologist'
train_dir = 'train_augmented'  # Subfolder where training data will be retrieved
test_dir = 'test'  # Subfolder where validation data will be retrieved

# Positive augmentation
augment_pos = False
angles_pos = [90, 180, 270]
axis_rot_pos = ['X', 'Y', 'Z']
dcgan_pos_data = 'nodules'

# Negative augmentation
augment_neg = False
angles_neg = [90, 180, 270]
axis_rot_neg = ['X', 'Y', 'Z']
dcgan_neg_data = 'non_nodule_train'

# GAN Version and option to augment
augment_GAN = False
dcgan_version = '1'
detector_gan = 'Nodule'

# Load saved trials object or new
load_saved_trials = False
trials_steps = 9  # Additional trials

# Load saved model to keep training it or load last best model training
load_saved = False
load_best = False
load_checkpoint = False
num_saved_model = 19
saved_checkpoint = 0

# How many epochs will the best model train?
train_epochs = 30
train_batch_size = 32

# Define the loss calculation
loss = 'binary_crossentropy'
```

```
# Calculate AUC
compute_AUC = True

# Assign more weight during training to specific class?
weight_classes = False
weight_neg_ratio = 1.
weight_pos_ratio = 2.

# ------------------------------CALLBACK INPUTS ------------------------------#

# ---------- CheckPoints -----------
save_best_only = True
monitor = 'val_loss'  # Metric to monitor
saving_per_epoch = 5
mode_checkpoints = 'min'

# --------- Early Stopping ----------
use_early_stop = False
min_delta = 0.001  # Minimum improvement required
patience = 25  # num of epochs with no improvement then training will be stopped
verbose = 1
mode_early = 'auto'
baseline = None  # value for the monitored quantity to reach

# restore model weights from the epoch with the best value of the monitored quantity
restore_best_weights = False

saved_model = ''
if load_saved or load_checkpoint:
    saved_model += str(num_saved_model)

# ------------------------------OPTIMIZER INPUTS ------------------------------#

space_dict = {}

lr_min = 0.0005
lr_max = 0.001
lr = [0.0001]

optimiz_epochs_max = 0  # max epoch
optimiz_epochs = [0]  # choice epochs

optimizers = ['adam']
batch_size = [32]  # max batch size to sample
extra_layers = 3  # max number of extra layers to try
extra_units = 15

decay_min = 0.01
decay_max = 0.05
decay = [0.01]

beta1_min = 0.9
beta1_max = 0.9
beta1 = [0.9]

beta2_min = 0.999
beta2_max = 0.999
beta2 = [0.999]

epsilon_min = 0.00000001
epsilon_max = 0.00000001
epsilon = [0.0001]

momentum_min = 0.9
momentum_max = 0.9
momentum = [0.9]

# For GoogleNet
num_inception = [1, 2, 3]
F1_inception = [4, 8, 16]
F2_inception = [4, 8, 16]
F3_inception = [4, 8, 16]
B1 = [4, 8, 16]
B2 = [4, 8, 16]
B3 = [4, 8, 16]
L2_s1_min = 0
L2_s1_max = 0.001
L2_s2_min = 0
L2_s2_max = 0.001

# For ResNet
num_conv = [1, 2]
num_identity = [1, 2]
num_stages = 5
filter_size = [3, 5]
F1_res = [4, 8, 16]
F2_res = [4, 8, 16]
F3_res = [4, 8, 16]
L2_conv_min = 0.0
L2_conv_max = 0.001
L2_identity_min = 0
L2_identity_max = 0.001
```

```
# For General Adaptative 1
# Intervals based on first adaptative run
avg_pool = [1]

num_conv_1 = 4
F1_adap_1 = [4, 8, 16]
l2_regularizer_1 = [0.0001, 0.0008]
filter_size_1 = [1]

num_conv_2 = 3
F2_adap_1 = [4, 8, 16]
l2_regularizer_2 = [0.0001, 0.0008]
filter_size_2 = [3]

num_conv_3 = 1
F3_adap_1 = [16, 32, 64]
l2_regularizer_3 = [0.00001, 0.0001]
filter_size_3 = [3]

num_conv_4 = 1
F4_adap_1 = [16, 32, 64]
l2_regularizer_4 = [0.0001, 0.001]
filter_size_4 = [1, 3]

F1 = F1_adap_1
F2 = F2_adap_1
F3 = F3_adap_1
F4 = F4_adap_1

# For ResNetInception V2 Adaptative
num_block35 = 10
num_block17 = 20
num_block8 = 10
l2_reg_block8 = [0, 0.001]
l2_reg_block17 = [0, 0.001]
l2_reg_block35 = [0, 0.001]
dropout = [0, 0.001]

# ----------------------------SPACE INPUTS --------------------------------#

# Sampling space  GoogleNet
space_googlenet = {
    'lr': hp.loguniform('lr', lr_min, lr_max),
    'batch_size': hp.choice('batch_size', batch_size),
    'decay': hp.loguniform('decay', decay_min, decay_max),
    'extra_units': hp.randint('extra_units', extra_units),
    'extra_dense': hp.randint('extra_layers', extra_layers),
    'epochs': hp.choice('epochs', optimiz_epochs),

    'L2_s1': hp.uniform('L2_s1', L2_s1_min, L2_s1_max),
    'L2_s2': hp.uniform('L2_s2', L2_s2_min, L2_s2_max),

    'optimizer': hp.choice('optimizer', optimizers),
    'beta1': hp.choice('beta1', beta1),
    'beta2': hp.choice('beta2', beta2),
    'epsilon': hp.choice('epsilon', epsilon),
    'momentum': hp.choice('momentum', momentum),

    'num_inception': hp.choice('num_inception', num_inception),

    'F1': hp.choice('F1', F1_inception),
    'F2': hp.choice('F2', F2_inception),
    'F3': hp.choice('F3', F3_inception),

    'B1': hp.choice('B1', B1),
    'B2': hp.choice('B2', B2),
    'B3': hp.choice('B3', B3)
}

# Sampling space  RESNET
space_resnet = {
    'lr': hp.loguniform('lr', lr_min, lr_max),
    'batch_size': hp.choice('batch_size', batch_size),
    'decay': hp.loguniform('decay', decay_min, decay_max),
    'extra_units': hp.randint('extra_units', extra_units),
    'extra_dense': hp.randint('extra_layers', extra_layers),
    'epochs': hp.choice('epochs', optimiz_epochs),

    'L2_conv': hp.uniform('L2_conv', L2_conv_min, L2_conv_max),
    'L2_identity': hp.uniform('L2_identity', L2_identity_min, L2_identity_max),

    'optimizer': hp.choice('optimizer', optimizers),
    'beta1': hp.choice('beta1', beta1),
    'beta2': hp.choice('beta2', beta2),
    'epsilon': hp.choice('epsilon', epsilon),
    'momentum': hp.choice('momentum', momentum),

    'num_conv': hp.choice('num_conv', num_conv),
    'num_identity': hp.choice('num_identity', num_identity),
    'num_stages': hp.randint('num_stages', num_stages),
    'filter_size': hp.choice('filter_size', filter_size),
```

```
    # Double for each stage
    'F1': hp.choice('F1', F1),
    'F2': hp.choice('F2', F2),
    'F3': hp.choice('F3', F3),
}

# Sampling space  RESNET
space_adaptative_1 = {
    'lr': hp.uniform('lr', lr_min, lr_max),
    'batch_size': hp.choice('batch_size', batch_size),
    'decay': hp.uniform('decay', decay_min, decay_max),
    'epochs': hp.choice('epochs', optimiz_epochs),
    'avg_pool': hp.choice('avg_pool', avg_pool),
    'extra_units': hp.randint('extra_units', extra_units),
    'extra_dense': hp.randint('extra_layers', extra_layers),

    'optimizer': hp.choice('optimizer', optimizers),
    'beta1': hp.choice('beta1', beta1),
    'beta2': hp.choice('beta2', beta2),
    'epsilon': hp.choice('epsilon', epsilon),
    'momentum': hp.choice('momentum', momentum),

    'F1': hp.choice('F1', F1),
    'F2': hp.choice('F2', F2),
    'F3': hp.choice('F3', F3),
    'F4': hp.choice('F4', F4),

    'num_conv_1': hp.randint('num_conv_1', num_conv_1),
    'num_conv_2': hp.randint('num_conv_2', num_conv_2),
    'num_conv_3': hp.randint('num_conv_3', num_conv_3),
    'num_conv_4': hp.randint('num_conv_4', num_conv_4),

    'l2_regularizer_1': hp.uniform('l2_regularizer_1', l2_regularizer_1[0], l2_regularizer_1[1]),
    'l2_regularizer_2': hp.uniform('l2_regularizer_2', l2_regularizer_2[0], l2_regularizer_2[1]),
    'l2_regularizer_3': hp.uniform('l2_regularizer_3', l2_regularizer_3[0], l2_regularizer_3[1]),
    'l2_regularizer_4': hp.uniform('l2_regularizer_4', l2_regularizer_4[0], l2_regularizer_4[1]),

    'filter_size_1': hp.choice('filter_size_1', filter_size_1),
    'filter_size_2': hp.choice('filter_size_2', filter_size_2),
    'filter_size_3': hp.choice('filter_size_3', filter_size_3),
    'filter_size_4': hp.choice('filter_size_4', filter_size_4)
}

space_resnet_inception_v2 = {
    'lr': hp.uniform('lr', lr_min, lr_max),
    'batch_size': hp.choice('batch_size', batch_size),
    'decay': hp.uniform('decay', decay_min, decay_max),
    'epochs': hp.choice('epochs', optimiz_epochs),

    'optimizer': hp.choice('optimizer', optimizers),
    'beta1': hp.choice('beta1', beta1),
    'beta2': hp.choice('beta2', beta2),
    'epsilon': hp.choice('epsilon', epsilon),

    'num_block35': hp.randint('num_block35', num_block35),
    'num_block17': hp.randint('num_block17', num_block17),
    'num_block8': hp.randint('num_block8', num_block8),

    'l2_reg_block8': hp.uniform('l2_reg_block8', l2_reg_block8[0], l2_reg_block8[1]),
    'l2_reg_block17': hp.uniform('l2_reg_block17', l2_reg_block17[0], l2_reg_block17[1]),
    'l2_reg_block35': hp.uniform('l2_reg_block35', l2_reg_block35[0], l2_reg_block35[1]),

    'dropout': hp.uniform('dropout', dropout[0], dropout[1])
}

space_bayesian = space_adaptative_1

# --------------------------- DIMENSION INPUTS -------------------------------#

# Dimensions and parameters of data
data_dims = 32
n_classes = 2
n_channels = 1
shuffle_data = True
target_size = (data_dims, data_dims, data_dims)
input_size = (data_dims, data_dims, data_dims, 1)  # For CNN

resample_data = True  # If data is not created yet, you want to resample it?
train_split = 0.7  # Percentage of training data from all data

# Folder where the model, checkpoints and settings will be saved
saving_folder = ('Models/Models {0} Detector/'
                 + 'Models of {0} Detectors 3D with Generator/'
                 + 'Model{1}_Version{2}/Size({3}, {3}, {3}) cv {4}').format(detector, num_model, version, data_dims, cv)

# Folder where data is located
data_path = 'E:/Base de datos maestría/LIDC-IDRI/Train Data/Train Data {0} Detector/{0} Detector 2 ' \
            'Classes Generator GANS/'.format(detector)

if not os.path.isdir(saving_folder):
    os.makedirs(saving_folder)

# Save screen shots
```

```
if take_screen_shot:
    if not load_best and not load_checkpoint and not load_saved and not load_saved_trials:
        for i in range(12):
            print('Enter 1 when ready to take screen shot {0}'.format(i + 1))
            take_shot = input()
            if take_shot:
                screen_shot = autopy.bitmap.capture_screen()
                screen_shot.save(saving_folder + '/screen_shot {0}.png'.format(i + 1))

# ---------------------- GENERATOR PARAMETERS INPUTS  -------------------------#
if use_generator:
    generator_training_folder = ('Train Data/Train Data Nodule Detector/Train Data Nodule Detector '
                                 + "3D Custom Generator 2 Classes/train_test_data_size_"
                                 + str(target_size)
                                 + data_source
                                 + '/')

    # Folder where all the training data is located. The training generator will
    # get all the data from here.
    dims = len(target_size)
    data_folder_train = (generator_training_folder + train_dir)

    # Dictionary containing all previous specified parameters for training generator
    params_train = {'data_folder': data_folder_train,
                    'dim': target_size,
                    'batch_size': batch_size[0],
                    'n_classes': n_classes,
                    'n_channels': n_channels,
                    'shuffle_data': shuffle_data,
                    'augment_negative': False,
                    'augment_positive': False,
                    'save': False,
                    'data_type': 'train'}

    # Folder where all the validation data is located. The validation generator
    # will get all the data from here.
    data_folder_test = (generator_training_folder + test_dir)

    # Dictionary containing all previous specified parameters for validation generator
    params_test = {'data_folder': data_folder_test,
                   'dim': target_size,
                   'batch_size': batch_size[0],
                   'n_classes': n_classes,
                   'n_channels': n_channels,
                   'shuffle_data': shuffle_data,
                   'augment_negative': False,
                   'augment_positive': False,
                   'save': False,
                   'data_type': 'test'}

    # Names of the folders containing the pos and neg classes
    nodule_dir = 'nodules'  # positive class folder name
    non_nodule_dir = 'non_nodules'  # negative class folder name

    if detector == 'Nodule':
        # Initialize training and validation generators
        train_generator = NoduleGenerator(**params_train)
        validation_generator = NoduleGenerator(**params_test)

    if detector == 'Malignancy':
        train_generator = MalignancyGenerator(**params_train)
        validation_generator = MalignancyGenerator(**params_test)

    # Define iterations of training and validation
    steps_per_epoch = train_generator.steps_per_epoch
    validation_steps = validation_generator.steps_per_epoch

# --------------------- PARAMETER INITIALIZERS INPUTS  ----------------------#

kernel_initializer = initializers.glorot_normal(seed=None)
bias_initializer = initializers.glorot_normal(seed=None)

# ---------------------------------- MODEL FOLDER ---------------------------#

# Define model name based on its version and number
model_name = 'Model{}_Version{}_Size({}, {}, {})'.format(num_model,
                                                         str(version),
                                                         str(data_dims),
                                                         str(data_dims),
                                                         str(data_dims))
model_folder = saving_folder + '/models'

if not os.path.isdir(model_folder):
    os.makedirs(model_folder)

# -------------------------- CREATE SPACE DICTIONARY  ----------------------#

if not load_saved and not load_best and not load_checkpoint:
    space_dict['optimiz_epochs_max'] = [optimiz_epochs_max]
    space_dict['optimiz_epochs'] = optimiz_epochs

    optimizers = ['adam']
    space_dict['optimizers'] = optimizers
```

```
    space_dict['batch_size'] = batch_size
    space_dict['extra_layers'] = [extra_layers]
    space_dict['extra_units'] = [extra_units]

    space_dict['decay_min'] = [decay_min]
    space_dict['decay_max'] = [decay_max]
    space_dict['decay'] = decay

    space_dict['L2_conv_min'] = [L2_conv_min]
    space_dict['L2_conv_max'] = [L2_conv_max]

    space_dict['L2_identity_min'] = [L2_identity_min]
    space_dict['L2_identity_max'] = [L2_identity_max]

    space_dict['filter_size'] = filter_size

    space_dict['num_identity'] = num_identity
    space_dict['num_conv'] = num_conv
    space_dict['num_stages'] = [num_stages]

    space_dict['beta1_min'] = [beta1_min]
    space_dict['beta1_max'] = [beta1_max]
    space_dict['beta1'] = beta1

    space_dict['beta2_min'] = [beta2_min]
    space_dict['beta2_max'] = [beta2_max]
    space_dict['beta2'] = beta2

    space_dict['epsilon_min'] = [epsilon_min]
    space_dict['epsilon_max'] = [epsilon_max]
    space_dict['epsilon'] = epsilon

    space_dict['momentum_min'] = [momentum_min]
    space_dict['momentum_max'] = [momentum_max]
    space_dict['momentum'] = momentum

    space_dict['avg_pool'] = [avg_pool]

    space_dict['num_conv_1'] = [num_conv_1]
    space_dict['F1'] = [F1]
    space_dict['l2_regularizer_1'] = [l2_regularizer_1]

    space_dict['num_conv_2'] = [num_conv_2]
    space_dict['F2'] = [F2]
    space_dict['l2_regularizer_2'] = [l2_regularizer_2]

    space_dict['num_conv_3'] = [num_conv_3]
    space_dict['F3'] = [F3]
    space_dict['l2_regularizer_3'] = [l2_regularizer_3]

    space_dict['num_conv_4'] = [num_conv_4]
    space_dict['F4'] = [F4]
    space_dict['l2_regularizer_4'] = [l2_regularizer_4]

    print('Saving space dictionary')
    pickle.dump(space_dict, open(saving_folder + "/search_space_dict.p", "wb"))
    df = pd.DataFrame.from_dict(space_dict, orient='index')
    df.to_excel(saving_folder + '/search_space_dict.xlsx')

# --------------------------------- LOAD DATA ------------------------------#

if not use_generator:

    # Train data
    x_train_pos = np.load(data_path + positive_data + '_train.npy')
    if len(x_train_pos.shape) is not 5:
        x_train_pos = np.expand_dims(x_train_pos, axis=5)

    x_train_neg = np.load(data_path + negative_data + '_train.npy')
    if len(x_train_neg.shape) is not 5:
        x_train_neg = np.expand_dims(x_train_neg, axis=5)

    x_train_g = np.concatenate((x_train_pos, x_train_neg), axis=0)
    y_train_g = np.concatenate((np.ones((x_train_pos.shape[0], 1)), np.zeros((x_train_neg.shape[0], 1))), axis=0)

    if len(x_train_g.shape) is not 5:
        x_train_g = np.expand_dims(x_train_g, axis=5)

    # Release memory
    del x_train_pos, x_train_neg
    gc.collect()

    # Test data
    x_test_pos = np.load(data_path + positive_data + '_test.npy')
    if len(x_test_pos.shape) is not 5:
        x_test_pos = np.expand_dims(x_test_pos, axis=5)

    x_test_neg = np.load(data_path + negative_data + '_test.npy')
    if len(x_test_neg.shape) is not 5:
        x_test_neg = np.expand_dims(x_test_neg, axis=5)

    x_test_g = np.concatenate((x_test_pos, x_test_neg), axis=0)
    y_test_g = np.concatenate((np.ones((x_test_pos.shape[0], 1)), np.zeros((x_test_neg.shape[0], 1))), axis=0)
```

```
    if len(x_test_g.shape) is not 5:
        x_test_g = np.expand_dims(x_test_g, axis=5)

    # Release memory
    del x_test_pos, x_test_neg
    gc.collect()

    # Shuffle training data
    indices = [i for i in range(x_train_g.shape[0])]
    shuffle(indices)
    x_train_g = x_train_g[indices, :, :, :, :]
    y_train_g = y_train_g[indices, :]

    # Shuffle test data
    indices = [i for i in range(x_test_g.shape[0])]
    shuffle(indices)
    x_test_g = x_test_g[indices, :, :, :, :]
    y_test_g = y_test_g[indices, :]

    if cv_validate or cv_search:
        x_data = np.concatenate((x_train_g, x_test_g), axis=0)
        y_data = np.concatenate((y_train_g, y_test_g), axis=0)
        # Release memory
        del x_train_g, x_test_g, y_test_g, y_train_g
        gc.collect()

if cv_search:
    # Get cross validation folds
    folds = list(StratifiedKFold(n_splits=k_folds, shuffle=True).split(x_data, y_data))

# If augment get the Generative Adversarial Network model and the data Transformer
if augment_pos and augment_GAN:
    # Create GAN for positive data
    dcgan_pos_path = 'E:/Base de datos maestría/LIDC-IDRI/Models/Models {0} Detector/Models of {0} 2 Classes ' \
                     'GAN/Model {1} {2}/'.format(detector_gan, dcgan_version, dcgan_pos_data)
    dcgan_pos = DCGAN()
    dcgan_pos.load_model(dcgan_pos_path)

if augment_neg and augment_GAN:
    # Create GAN for negative data
    dcgan_neg_path = 'E:/Base de datos maestría/LIDC-IDRI/Models/Models {0} Detector/Models of {0} 2 Classes ' \
                     'GAN/Model {1} {2}/'.format(detector_gan, dcgan_version, dcgan_neg_data)
    dcgan_neg = DCGAN()
    dcgan_neg.load_model(dcgan_neg_path)

if augment_neg or augment_pos:
    # Create data Transformer
    dt = dataTransformer()

# ---------------------------------- OBJECTIVE -------------------------------#

# Specify if the training will give more weight to either positive or negative
if weight_classes:
    class_weight = {0: weight_neg_ratio,
                    1: weight_pos_ratio}
else:
    class_weight = {0: 1.,
                    1: 1.}

# Define the metrics that will be calculated in the model
f1 = custom_Keras_Metrics.f1
sens = custom_Keras_Metrics.sensitivity
spec = custom_Keras_Metrics.specificity
fpr = custom_Keras_Metrics.falsePositiveRate
metrics = ['accuracy', f1, sens, spec, fpr]

# Get the model from file: "classes_lidc"
model_str = 'model_' + str(num_model)
model_method = getattr(Bayesian, model_str)

if not load_saved and not load_best and not load_checkpoint:
    n_trial = 0
else:
    n_trial = saved_checkpoint + 1

# Loading trials
if not os.path.isdir(saving_folder + '/logs'):
    os.makedirs(saving_folder + '/logs')

# Star from zero
if not load_saved_trials:
    trials = Trials()
else:
    try:
        trials = pickle.load(open(saving_folder + '/trials.p', "rb"))
        max_evals = len(trials.trials) + trials_steps
        n_trial = len(trials.trials) + 1
        print('Max evals: {0}'.format(max_evals))
    except ValueError as error:
        print(error)
        trials = Trials()
```

```
def objective(space):
    """
    Model providing function:
    Create Keras model with double curly brackets dropped-in as needed.
    Return value has to be a valid python dictionary with two customary keys:
        - loss: Specify a numeric evaluation metric to be minimized
        - status: Just use STATUS_OK and see hyperopt documentation if not feasible
    The last one is optional, though recommended, namely:
        - model: specify the model just created so that we can later use it again.
    """

    # ---------------------------- OPTIMIZER # ----------------------------#
    global n_trial, trials, k_folds, cv

    callbacks_temp = []
    optimizer_temp = None

    if space['optimizer'] is 'adam':
        optimizer_temp = Adam(lr=space['lr'],
                              beta_1=space['beta1'],
                              beta_2=space['beta2'],
                              epsilon=space['epsilon'],
                              decay=space['decay'])

    if space['optimizer'] is 'adagrad':
        optimizer_temp = Adagrad(lr=space['lr'],
                                 epsilon=space['epsilon'],
                                 decay=space['decay'])

    if space['optimizer'] is 'sgd':
        optimizer_temp = SGD(lr=space['lr'],
                             momentum=space['momentum'],
                             decay=space['decay'])

    model_temp = model_method(input_shape=input_size,
                              n_classes=n_classes,
                              space=space,
                              kernel_initializer=kernel_initializer,
                              bias_initializer=bias_initializer)

    # Compile model
    model_temp.compile(optimizer=optimizer_temp,
                       loss=loss,
                       metrics=metrics)

    # Show model summary
    model_temp.summary()

    # ---------------------------------- CROSS VALIDATION ----------------------------------
    if cv_search:

        global x_data, y_data

        # Get cross validation folds
        folds = list(StratifiedKFold(n_splits=k_folds, shuffle=True).split(x_data, y_data))

        # Initialize a list of cross validation losses
        cross_losses = []

        # Train model in each fold of cross validation
        for j, (train_idx, val_idx) in enumerate(folds):

            print('\nFold ', j)

            # Create TensorBoard Callback
            callbacks_temp = []
            name = 'trial {0} cv {1} '.format(n_trial, j) + str(int(time.time()))
            tensorboard_temp = TensorBoard(log_dir=saving_folder + '/logs/{}'.format(name))
            callbacks_temp.append(tensorboard_temp)

            # Get data for specific k fold
            x_train = x_data[train_idx]
            y_train = y_data[train_idx]
            x_test = x_data[val_idx]
            y_test = y_data[val_idx]

            # Get positive data
            pos_idx = np.where(y_train[:, 0] == 1)
            x_pos = x_train[pos_idx]
            num_pos = x_pos.shape[0]

            # Get negative data
            neg_idx = np.where(y_train[:, 0] == 0)
            x_neg = x_train[neg_idx]
            num_neg = x_neg.shape[0]

            # Release memory
            del x_train
            gc.collect()

            # Augment training data
            if augment_pos:
                print('\n   Augmenting positive data with rotations...')
```

```
            x_pos = dt.rotate_data(x_pos[:, :, :, :, 0], axis_rot_pos, angles_pos)
            x_pos = np.expand_dims(x_pos, axis=4)
            num_pos = x_pos.shape[0]

        if augment_neg:
            print('\n   Augmenting negative data with rotations...')
            x_neg = dt.rotate_data(x_neg[:, :, :, :, 0], axis_rot_neg, angles_neg)
            x_neg = np.expand_dims(x_neg, axis=4)
            num_neg = x_neg.shape[0]

        if augment_neg or augment_pos and augment_GAN:
            # Generate positive or negative data according to class imbalance
            generated = []

            if num_pos > num_neg and augment_neg:
                print('\n   Augmenting negative data with GAN...')
                generated = dcgan_neg.generate_data(num_pos - num_neg)
                x_neg = np.concatenate((x_neg, generated), axis=0)
            elif num_pos < num_neg and augment_pos:
                print('\n   Augmenting positive data with GAN...')
                generated = dcgan_pos.generate_data(num_neg - num_pos)
                x_pos = np.concatenate((x_pos, generated), axis=0)

            # Release memory
            del generated
            gc.collect()

        # Create training data and append both positive and negative examples
        x_train = np.zeros(((num_pos + num_neg,) + x_pos.shape[1:]), dtype=np.float32)

        x_train[0:num_pos] = x_pos
        del x_pos
        gc.collect()

        x_train[num_pos:] = x_neg
        del x_neg
        gc.collect()

        # Create augmented labels for training data and concatenate positive with negative
        y_train = np.concatenate((np.ones((num_pos, 1)), np.zeros((num_neg, 1))), axis=0)

        # Shuffle augmented training data
        indices = [i for i in range(x_train.shape[0])]
        shuffle(indices)
        x_train = x_train[indices, :, :, :, :]
        y_train = y_train[indices, :]

        # Train and validate model
        hist = model_temp.fit(x=x_train,
                              y=y_train,
                              batch_size=space['batch_size'],
                              epochs=space['epochs'] + 1,
                              validation_data=(x_test, y_test),
                              verbose=1,
                              class_weight=class_weight,
                              callbacks=callbacks_temp)

        # Release memory
        del x_test, x_train
        gc.collect()

        # Save validation loss for this k-fold
        val_loss = hist.history['val_loss'][-1]
        cross_losses.append(val_loss)
        print('Validation loss of {0} fold: {1}', j, val_loss)

    # Get average of all losses
    val_loss = np.mean(cross_losses)
    print('Average cross validation loss: ', val_loss)

# ---------------------------------- NO CROSS VALIDATION ----------------------------------
else:

    # ---------------------------------- NO DATA GENERATOR ----------------------------------
    if not use_generator:

        global x_train_g, y_train_g, x_test_g, y_test_g

        # Create TensorBoard Callback
        name = 'trial {} '.format(n_trial) + str(int(time.time()))
        tensorboard_temp = TensorBoard(log_dir=saving_folder + '/logs/{}'.format(name))
        callbacks_temp.append(tensorboard_temp)

        # Get positive data
        pos_idx = np.where(y_train_g[:, 0] == 1)
        x_pos = x_train_g[pos_idx]
        num_pos = x_pos.shape[0]

        # Get negative data
        neg_idx = np.where(y_train_g[:, 0] == 0)
        x_neg = x_train_g[neg_idx]
        num_neg = x_neg.shape[0]
```

```
            # Augment training data
            if augment_pos:
                print('\n   Augmenting positive data with rotations...')
                x_pos = dt.rotate_data(x_pos[:, :, :, :, 0], axis_rot_pos, angles_pos)
                x_pos = np.expand_dims(x_pos, axis=4)
                num_pos = x_pos.shape[0]

            if augment_neg:
                print('\n   Augmenting negative data with rotations...')
                x_neg = dt.rotate_data(x_neg[:, :, :, :, 0], axis_rot_neg, angles_neg)
                x_neg = np.expand_dims(x_neg, axis=4)
                num_neg = x_neg.shape[0]

            if augment_neg or augment_pos and augment_GAN:
                # Generate positive or negative data according to class imbalance
                generated = []

                if num_pos > num_neg and augment_neg:
                    print('\n   Augmenting negative data with GAN...')
                    generated = dcgan_neg.generate_data(num_pos - num_neg)
                    x_neg = np.concatenate((x_neg, generated), axis=0)
                elif num_pos < num_neg and augment_pos:
                    print('\n   Augmenting positive data with GAN...')
                    generated = dcgan_pos.generate_data(num_neg - num_pos)
                    x_pos = np.concatenate((x_pos, generated), axis=0)

                # Release memory
                del generated
                gc.collect()

            # Create augmented labels for training data and concatenate positive with negative
            x_train = np.concatenate((x_pos, x_neg), axis=0)
            y_train = np.concatenate((np.ones((x_pos.shape[0], 1)), np.zeros((x_neg.shape[0], 1))), axis=0)

            # Release memory
            del x_pos, x_neg
            gc.collect()

            # Shuffle augmented training data
            indices = [i for i in range(x_train.shape[0])]
            shuffle(indices)
            x_train = x_train[indices, :, :, :, :]
            y_train = y_train[indices, :]

            # Train and validate model
            hist = model_temp.fit(x=x_train,
                                  y=y_train,
                                  batch_size=space['batch_size'],
                                  epochs=space['epochs'] + 1,
                                  validation_data=(x_test_g, y_test_g),
                                  verbose=1,
                                  class_weight=class_weight,
                                  callbacks=callbacks_temp)

            # Release memory
            del x_train
            gc.collect()

            # Trying to minimize the last validation loss at the end of epoch
            val_loss = hist.history['val_loss'][-1]
            print('Validation loss: ', val_loss)

        # ---------------------------------- USE DATA GENERATOR ----------------------------------
        else:

            # Create TensorBoard Callback
            name = 'trial {} '.format(n_trial) + str(int(time.time()))
            tensorboard_temp = TensorBoard(log_dir=saving_folder + '/logs/{}'.format(name))
            callbacks_temp.append(tensorboard_temp)

            # Train and validate model
            hist = model_temp.fit_generator(train_generator,
                                            epochs=space['epochs'] + 1,
                                            validation_data=validation_generator,
                                            steps_per_epoch=steps_per_epoch,
                                            validation_steps=validation_steps,
                                            verbose=1,
                                            use_multiprocessing=False,
                                            workers=20,
                                            max_queue_size=20,
                                            class_weight=class_weight,
                                            callbacks=callbacks_temp)

            # Trying to minimize the last validation loss at the end of epoch
            val_loss = hist.history['val_loss'][-1]
            print('Validation loss: ', val_loss)

    # Save model
    model_temp.save(model_folder + '/' + 'model ' + str(n_trial) + '.h5')

    # Save trials object
    pickle.dump(trials, open(saving_folder + "/trials.p", "wb"))
```

```
    n_trial += 1

    return {'loss': val_loss, 'n_trial': n_trial, 'status': STATUS_OK}

# ------------------------------ MAIN PROGRAM -------------------------------#

# Initialize model
model = None

# If we are not training but searching
if not load_saved and not load_best and not load_checkpoint:

    # ------------------------- SEARCH FOR BEST MODEL -------------------------#
    best = fmin(objective,
                space=space_bayesian,
                algo=tpe.suggest,
                max_evals=max_evals,
                trials=trials)

    # ---------------------- SAVE OPTIMIZATION RESULTS ----------------------#

    # Save all the specified parameters of this training in a '.txt' file
    optimization_folder = (saving_folder + '/optimization_results.xlsx')

    opt_results = {'loss': [x['loss'] if 'loss' in x.keys() else 'nan' for x in trials.results ]}

    for key, value in trials.idxs_vals[1].items():
        opt_results[key] = trials.idxs_vals[1][key]

    decay_results = pd.DataFrame(opt_results)
    decay_results.to_excel(optimization_folder)

    # ----------------------------- SAVE BEST MODEL -----------------------------#

    # Get best model
    n_model = trials.best_trial['result']['n_trial']
    best = trials.best_trial['misc']['vals']
    model_path = (model_folder
                  + '/model '
                  + str(n_model - 1)
                  + '.h5')
    model = load_model(model_path, custom_objects={'f1': f1,
                                                   'sensitivity': sens,
                                                   'specificity': spec,
                                                   'falsePositiveRate': fpr})

    # Save the model architecture as image
    plot_model(model, to_file=saving_folder + '/architecture.png')

    # Print the details of the CNN architecture and save it
    model.summary()
    summary_folder = (saving_folder + '/model_summary.txt')
    with open(summary_folder, 'w') as f:
        # Pass the file handle in as a lambda function to make it callable
        model.summary(print_fn=lambda x: f.write(x + '\n'))

    # Save model architecture allowing it to be used later
    print('Saving model {}...'.format(model_name))
    model.save(saving_folder + '/best model ' + str(n_model) + '.h5')

    # ----------------------------- SAVE TRIALS -----------------------------#

    # Save trials object
    pickle.dump(trials, open(saving_folder + "/trials.p", "wb"))

    # ------------------------- SAVE BEST MODEL SETTINGS  --------------------#

    # Save all the specified parameters of this training in a '.txt' file
    parameters_folder = (saving_folder + '/parameters.txt')

    if not os.path.isdir(saving_folder):
        os.makedirs(saving_folder)

    f = open(parameters_folder, 'w')
    f.write('------------ BAYESIAN OPTIMIZATION BEST MODEL ------------\n')
    f.write('\n')

    f.write('TRAINING PARAMETERS\n')
    f.write('   epochs: {}\n'.format(str(optimiz_epochs[best['epochs'][0]])))
    f.write('   batch_size: {}\n'.format(str(batch_size[best['batch_size'][0]])))
    f.write('   learning_rate: {}\n'.format(str(best['lr'][0])))
    f.write('   decay: {}\n'.format(str(best['decay'][0])))
    f.write('   extra dense layers: {}\n'.format(str(best['extra_layers'][0])))
    f.write('   extra units for dense layers : {}\n'.format(str(best['extra_units'][0])))
    f.write('\n')

    f.write('OPTIMIZER PARAMETERS\n')
    f.write('   optimizer: {}\n'.format(str(optimizers[best['optimizer'][0]])))
    f.write('   beta 1: {}\n'.format(str(beta1[best['beta1'][0]])))
    f.write('   beta 2: {}\n'.format(str(beta2[best['beta2'][0]])))
    f.write('   epsilon: {}\n'.format(str(epsilon[best['epsilon'][0]])))
    f.write('   momentum: {}\n'.format(str(momentum[best['momentum'][0]])))
    f.write('\n')
```

```
    f.write('DATA PARAMETERS\n')
    f.write('   suffle data in every batch: {}\n'.format(str(shuffle_data)))
    f.write('   input_size: {}\n'.format(str(input_size)))
    f.write('   train_split: {}\n'.format(str(train_split)))
    f.write('\n')

    f.close()

# ---------------------------- CALLBACKS SAVING  ---------------------------#

callbacks = []

# Create checkpoints folder for best model
if not os.path.isdir(saving_folder + '/' + 'best model checkpoints{}'.format(saved_model)):
    os.makedirs(saving_folder + '/' + 'best model checkpoints{}'.format(saved_model))

# Initialize frequency and folder to save checkpoints
checkpoint_folder = (saving_folder
                     + '/best model checkpoints{}'.format(saved_model)
                     + '/{epoch:02d}.hd5f')

checkpoints = ModelCheckpoint(checkpoint_folder,
                              monitor=monitor,
                              save_weights_only=False,
                              save_best_only=save_best_only,
                              period=saving_per_epoch,
                              mode='min',
                              verbose=1)
callbacks.append(checkpoints)

# Tensorboard callbacks
tensorboard = TensorBoard(log_dir=saving_folder + '/logs/{}'.format('best_model{}'.format(saved_model)))
callbacks.append(tensorboard)

# Create early stopping callback
if use_early_stop:
    earyly_stopping = EarlyStopping(monitor=monitor,
                                    min_delta=min_delta,
                                    patience=patience,
                                    verbose=verbose,
                                    mode=mode_early,
                                    baseline=baseline,
                                    restore_best_weights=restore_best_weights)
    callbacks.append(earyly_stopping)

# ---------------------------- KEEP TRAINING BEST MODEL ----------------------#

# Load previous version
if load_saved:
    model_path = (model_folder
                  + '/model '
                  + str(saved_model)
                  + '.h5')
    model = load_model(model_path, custom_objects={'f1': f1,
                                                   'sensitivity': sens,
                                                   'specificity': spec,
                                                   'falsePositiveRate': fpr})

if load_best or load_checkpoint:
    model_path = (saving_folder
                  + '/best model checkpoints{}'.format(saved_model)
                  + '/'
                  + str(saved_checkpoint)
                  + '.hd5f')
    model = load_model(model_path, custom_objects={'f1': f1,
                                                   'sensitivity': sens,
                                                   'specificity': spec,
                                                   'falsePositiveRate': fpr})

# ---------------------------------- CROSS VALIDATION ----------------------------------
if cv_validate:
    # Get cross validation folds
    folds = list(StratifiedKFold(n_splits=k_folds, shuffle=True).split(x_data, y_data))

    # Initialize list of metrics
    cross_histories, cross_val_losses = [], []
    tp_rate = []
    fp_rate = []
    thresholds_avg = []
    roc_aucs = []

    # ------------------------------ ITERATE CROSS-VALIDATION -------------------------------
    for j, (train_idx, val_idx) in enumerate(folds):

        print('\nFold ', j)

        # ------------------------------ CALLBACKS -------------------------------
        callbacks = []

        checkpoint_folder = (saving_folder
                             + '/best model checkpoints{0}/cv{1} '.format(saved_model, j)
                             + '{epoch:02d}.hd5f')
```

```
checkpoints = ModelCheckpoint(checkpoint_folder,
                              monitor=monitor,
                              save_weights_only=False,
                              save_best_only=save_best_only,
                              period=saving_per_epoch,
                              mode='min',
                              verbose=1)
callbacks.append(checkpoints)

# Tensorboard callbacks
tensorboard = TensorBoard(
    log_dir=saving_folder + '/logs/{}'.format('best_model{0}cv{1}'.format(saved_model, j)))
callbacks.append(tensorboard)

# ------------------------------ DATA -----------------------------

# Get data for specific k fold
x_train = x_data[train_idx]
y_train = y_data[train_idx]
x_test = x_data[val_idx]
y_test = y_data[val_idx]

# Get positive data
pos_idx = np.where(y_train[:, 0] == 1)
x_pos = x_train[pos_idx]
num_pos = x_pos.shape[0]

# Get negative data
neg_idx = np.where(y_train[:, 0] == 0)
x_neg = x_train[neg_idx]
num_neg = x_neg.shape[0]

# Release memory
del x_train
gc.collect()

# ------------------------------ AUGMENTATION -----------------------------
if augment_pos:
    print('\n   Augmenting positive data with rotations...')
    x_pos = dt.rotate_data(x_pos[:, :, :, :, 0], axis_rot_pos, angles_pos)
    x_pos = np.expand_dims(x_pos, axis=4)
    num_pos = x_pos.shape[0]

if augment_neg:
    print('\n   Augmenting negative data with rotations...')
    x_neg = dt.rotate_data(x_neg[:, :, :, :, 0], axis_rot_neg, angles_neg)
    x_neg = np.expand_dims(x_neg, axis=4)
    num_neg = x_neg.shape[0]

if augment_neg or augment_pos and augment_GAN:
    # Generate positive or negative data according to class imbalance
    generated = []

    if num_pos > num_neg and augment_neg:
        print('\n   Augmenting negative data with GAN...')
        generated = dcgan_neg.generate_data(num_pos - num_neg)
        x_neg = np.concatenate((x_neg, generated), axis=0)
    elif num_pos < num_neg and augment_pos:
        print('\n   Augmenting positive data with GAN...')
        generated = dcgan_pos.generate_data(num_neg - num_pos)
        x_pos = np.concatenate((x_pos, generated), axis=0)

    # Release memory
    del generated
    gc.collect()

# Create augmented labels for training data and concatenate positive with negative
x_train = np.concatenate((x_pos, x_neg), axis=0)
y_train = np.concatenate((np.ones((x_pos.shape[0], 1)), np.zeros((x_neg.shape[0], 1))), axis=0)

# Release memory
del x_pos, x_neg
gc.collect()

# Shuffle augmented training data
indices = [i for i in range(x_train.shape[0])]
shuffle(indices)
x_train = x_train[indices, :, :, :, :]
y_train = y_train[indices, :]

# ------------------------------ TRAINING -----------------------------

# Train and validate model
history = model.fit(x=x_train,
                    y=y_train,
                    batch_size=train_batch_size,
                    epochs=train_epochs,
                    validation_data=(x_test, y_test),
                    class_weight=class_weight,
                    verbose=1,
                    callbacks=callbacks)
```

```
        # Release memory
        del x_train
        gc.collect()

        # Save validation loss for this k-fold
        cross_histories.append(history.history)
        val_loss = history.history['val_loss'][-1]
        cross_val_losses.append(val_loss)
        print('Validation loss of {0} fold: {1}', j, val_loss)

        # ------------------------------ COMPUTE AUC -------------------------------#
        if compute_AUC:
            # Compute predictions
            print('Predicting instances...\n')
            true_labels = []
            num = 0
            predictions = model.predict(x_test)

            # Compute AUC
            print('Computing AUC... \n')
            fp_rate, tp_rate, thresholds = roc_curve(y_test, predictions)

            # Get tpr_file file path
            tpr_file = saving_folder + '/tp_rate model {} cv'.format(saved_model) + str(j) + '.p'

            # Get fpr_file file path
            fpr_file = saving_folder + '/fp_rate model {} cv'.format(saved_model) + str(j) + '.p'

            # Get AUC file path
            auc_file = saving_folder + '/auc model {} cv'.format(saved_model) + str(j) + '.p'

            # Get thresholds file path
            thresholds_file = saving_folder + '/thresholds model {} cv'.format(saved_model) + str(j) + '.p'

            # Get optimal threshold file path
            opt_threshold_file = saving_folder + '/optimal threshold model {} cv'.format(saved_model) + str(j) + '.p'

            roc_auc = auc(fp_rate, tp_rate)

            roc_aucs.append(roc_auc)

            # Determine best threshold value: tpr high and fpr low
            # tf = tpr - (1-fpr) is zero or near to zero is the optimal cut off point
            k = np.arange(len(tp_rate))  # index for table
            roc = pd.DataFrame({'fp_rate': pd.Series(fp_rate, index=k),
                                'tp_rate': pd.Series(tp_rate, index=k),
                                '1-fp_rate': pd.Series(1 - fp_rate, index=k),
                                'tf': pd.Series(tp_rate - (1 - fp_rate), index=k),
                                'threshold': pd.Series(thresholds, index=k)})
            roc_t = roc.ix[(roc.tf - 0).abs().argsort()[:1]]
            opt_threshold = list(roc_t['threshold'])

            # Save results
            pickle.dump(tp_rate, open(tpr_file, "wb"))
            pickle.dump(fp_rate, open(fpr_file, "wb"))
            pickle.dump(thresholds_avg, open(thresholds_file, "wb"))
            pickle.dump(roc_auc, open(auc_file, "wb"))
            pickle.dump(opt_threshold, open(opt_threshold_file, "wb"))

            # Plot AUC
            plt.figure(j)
            plt.ylim(0, 1.2)
            plt.plot([0, 1], [0, 1], 'k--')
            plt.plot(fp_rate, tp_rate, label='Área = {:.6f}'.format(roc_auc))
            plt.xlabel('Average false positive rate')
            plt.ylabel('Average true positive rate')
            plt.title('Average ROC Curve')
            plt.legend(loc='best')
            plt.show()
            plt.savefig(saving_folder + '/' + 'AUC best model {0} cv {1}.png'.format(saved_model, j))
            plt.close(fig='all')

            # Graficar mejor umbral
            plt.figure(j)
            fig, ax = pl.subplots()
            pl.plot(roc['tp_rate'])
            pl.plot(roc['1-fp_rate'], color='red')
            pl.xlabel('Average 1-False Positive Rate')
            pl.ylabel('Average True Positive Rate')
            pl.title('Receiver operating characteristic')
            ax.set_xticklabels([])
            plt.savefig(saving_folder + '/' + 'Mejor Umbral best model {0} cv {1}.png'.format(saved_model, j))
            plt.close(fig='all')

        # Release memory
        del x_test
        gc.collect()

    # ------------------------------ SAVE RESULTS -------------------------------#

    # Get history file path
    history_file = saving_folder + '/model_cross_histories {}'.format(saved_model) + '.p'
    pickle.dump(cross_histories, open(history_file, "wb"))
```

```
    # Get mean AUC
    auc_file = saving_folder + '/mean_AUC {}'.format(saved_model) + '.p'
    pickle.dump(np.mean(roc_aucs), open(auc_file, "wb"))

    # Get average of all losses
    val_loss = np.mean(cross_val_losses)

    print('Average cross validation loss: ', val_loss)

    # Save model architecture allowing it to be used later
    print('Saving best model after {}...'.format(model_name))
    model.save(saving_folder + '/best model {}'.format(saved_model) + '.h5')

    # ------------------------------ PLOT RESULTS -------------------------------#

    # summarize history for accuracy
    avg_data = np.sum(np.array([h['acc'] for h in cross_histories]), axis=0) / k_folds
    avg_val = np.sum(np.array([h['val_acc'] for h in cross_histories]), axis=0) / k_folds

    plt.figure(1)
    plt.plot(avg_data)
    plt.plot(avg_val)
    plt.title('Precisión promedio del modelo {}'.format(num_model))
    plt.ylabel('Precisión')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving accuracy plot...')
    plt.savefig(saving_folder + '/' + 'Accuracy_plot best model {}.png'.format(saved_model))

    # summarize history for loss
    avg_data = np.sum(np.array([h['loss'] for h in cross_histories]), axis=0) / k_folds
    avg_val = np.sum(np.array([h['val_loss'] for h in cross_histories]), axis=0) / k_folds

    plt.figure(2)
    plt.plot(avg_data)
    plt.plot(avg_val)
    plt.title('Pérdida logarítmica promedio del modelo {}'.format(num_model))
    plt.ylabel('Pérdida logarítmica')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving loss plot...')
    plt.savefig(saving_folder + '/' + 'Loss_plot best model {}.png'.format(saved_model))

    # summarize history for F1 score
    avg_data = np.sum(np.array([h['f1'] for h in cross_histories]), axis=0) / k_folds
    avg_val = np.sum(np.array([h['val_f1'] for h in cross_histories]), axis=0) / k_folds

    plt.figure(3)
    plt.plot(avg_data)
    plt.plot(avg_val)
    plt.title('Puntaje F1 promedio del modelo {}'.format(num_model))
    plt.ylabel('Puntaje F1')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving F1 score plot...')
    plt.savefig(saving_folder + '/' + 'F1_score_plot best model {}.png'.format(saved_model))

    # summarize history for sensitivity
    avg_data = np.sum(np.array([h['specificity'] for h in cross_histories]), axis=0) / k_folds
    avg_val = np.sum(np.array([h['val_specificity'] for h in cross_histories]), axis=0) / k_folds

    plt.figure(4)
    plt.plot(avg_data)
    plt.plot(avg_val)
    plt.title('Especificidad promedio del modelo {}'.format(num_model))
    plt.ylabel('Especificidad [ TP / (TP + FN)  ]')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving specificity plot...')
    plt.savefig(saving_folder + '/' + 'specificity_plot best model {}.png'.format(saved_model))

    # summarize history for specificity
    avg_data = np.sum(np.array([h['sensitivity'] for h in cross_histories]), axis=0) / k_folds
    avg_val = np.sum(np.array([h['val_sensitivity'] for h in cross_histories]), axis=0) / k_folds

    plt.figure(5)
    plt.plot(avg_data)
    plt.plot(avg_val)
    plt.title('Sensibilidad promedio del modelo {}'.format(num_model))
    plt.ylabel('Sensibilidad [ TN / (TN + FP)  ]')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving sensitivity plot...')
    plt.savefig(saving_folder + '/' + 'sensitivity_plot best model {}.png'.format(saved_model))

# ---------------------------------- NO CROSS VALIDATION ----------------------------------
```

```
else:

    # ---------------------------------- CLASSIC TRAINING ----------------------------------

    if not use_generator:

        if cv_search:

            # Release memory
            #del x_data, y_data
            #gc.collect()

            # Positive data
            x_train_pos = np.load(data_path + positive_data + '_train.npy')
            if len(x_train_pos.shape) is not 5:
                x_train_pos = np.expand_dims(x_train_pos, axis=5)

            x_train_neg = np.load(data_path + negative_data + '_train.npy')
            if len(x_train_neg.shape) is not 5:
                x_train_neg = np.expand_dims(x_train_neg, axis=5)

            x_train_g = np.concatenate((x_train_pos, x_train_neg), axis=0)
            y_train_g = np.concatenate((np.ones((x_train_pos.shape[0], 1)), np.zeros((x_train_neg.shape[0], 1))), axis=0)

            # Release memory
            del x_train_pos, x_train_neg
            gc.collect()

            # Negative data
            x_test_pos = np.load(data_path + positive_data + '_test.npy')
            if len(x_test_pos.shape) is not 5:
                x_test_pos = np.expand_dims(x_test_pos, axis=5)

            x_test_neg = np.load(data_path + negative_data + '_test.npy')
            if len(x_test_neg.shape) is not 5:
                x_test_neg = np.expand_dims(x_test_neg, axis=5)

            x_test_g = np.concatenate((x_test_pos, x_test_neg), axis=0)
            y_test_g = np.concatenate((np.ones((x_test_pos.shape[0], 1)), np.zeros((x_test_neg.shape[0], 1))), axis=0)

            # Release memory
            del x_test_pos, x_test_neg
            gc.collect()

            # Shuffle training data
            indices = [i for i in range(x_train_g.shape[0])]
            shuffle(indices)
            x_train_g = x_train_g[indices, :, :, :, :]
            y_train_g = y_train_g[indices, :]

            # Shuffle test data
            indices = [i for i in range(x_test_g.shape[0])]
            shuffle(indices)
            x_test_g = x_test_g[indices, :, :, :, :]
            y_test_g = y_test_g[indices, :]

        # Train and validate model
        history = model.fit(x=x_train_g,
                            y=y_train_g,
                            batch_size=train_batch_size,
                            epochs=train_epochs,
                            validation_data=(x_test_g, y_test_g),
                            class_weight=class_weight,
                            verbose=1,
                            callbacks=callbacks)

        # Get history file path
        history_file = saving_folder + '/model_history {}'.format(saved_model) + '.p'

        # Save history of training
        saved_history = history.history
        pickle.dump(saved_history, open(history_file, "wb"))

        # Save model architecture allowing it to be used later
        print('Saving best model after {}...'.format(model_name))
        model.save(saving_folder + '/best model {}'.format(saved_model) + '.h5')

        # ------------------------------ COMPUTE AUC -------------------------------#

        if compute_AUC:
            # Get tpr_file file path
            tpr_file = saving_folder + '/tp_rate model {}'.format(saved_model) + '.p'

            # Get fpr_file file path
            fpr_file = saving_folder + '/fp_rate model {}'.format(saved_model) + '.p'

            # Get AUC file path
            auc_file = saving_folder + '/auc model {}'.format(saved_model) + '.p'

            # Get thresholds file path
            thresholds_file = saving_folder + '/thresholds model {}'.format(saved_model) + '.p'

            # Get optimal threshold file path
```

```
        opt_threshold_file = saving_folder + '/optimal threshold model {}'.format(saved_model) + '.p'

        # Compute predictions
        print('')
        print('Computing predictions: ')
        print('')

        true_labels = []
        num = 0

        print('Predicting instances...\n')
        predictions = model.predict(x_test_g)

        # Compute AUC
        print('Computing AUC... \n')
        fp_rate, tp_rate, thresholds = roc_curve(y_test_g, predictions)
        auc = auc(fp_rate, tp_rate)

        # Determine best threshold value: tpr high and fpr low
        # tf = tpr - (1-fpr) is zero or near to zero is the optimal cut off point
        i = np.arange(len(tp_rate))  # index for table
        roc = pd.DataFrame({'fp_rate': pd.Series(fp_rate, index=i), 'tp_rate': pd.Series(tp_rate, index=i),
                            '1-fp_rate': pd.Series(1 - fp_rate, index=i),
                            'tf': pd.Series(tp_rate - (1 - fp_rate), index=i),
                            'threshold': pd.Series(thresholds, index=i)})
        roc_t = roc.ix[(roc.tf - 0).abs().argsort()[:1]]
        opt_threshold = list(roc_t['threshold'])

        # Save results
        pickle.dump(tp_rate, open(tpr_file, "wb"))
        pickle.dump(fp_rate, open(fpr_file, "wb"))
        pickle.dump(thresholds, open(thresholds_file, "wb"))
        pickle.dump(auc, open(auc_file, "wb"))
        pickle.dump(opt_threshold, open(opt_threshold_file, "wb"))

        # Plot AUC
        plt.figure(7)
        plt.plot([0, 1], [0, 1], 'k--')
        plt.plot(fp_rate, tp_rate, label='Área = {:.3f}'.format(auc))
        plt.xlabel('False positive rate')
        plt.ylabel('True positive rate')
        plt.title('Curva ROC')
        plt.legend(loc='best')
        plt.show()
        plt.savefig(saving_folder + '/' + 'AUC best model {}.png'.format(saved_model))

        # Graficar mejor umbral
        fig, ax = pl.subplots()
        pl.plot(roc['tp_rate'])
        pl.plot(roc['1-fp_rate'], color='red')
        pl.xlabel('1-False Positive Rate')
        pl.ylabel('True Positive Rate')
        pl.title('Receiver operating characteristic')
        ax.set_xticklabels([])
        plt.savefig(saving_folder + '/' + 'Mejor Umbral best model {}.png'.format(saved_model))

    # ------------------------------ PLOT RESULTS --------------------------------#

    # summarize history for accuracy
    plt.figure(1)
    plt.plot(saved_history['acc'])
    plt.plot(saved_history['val_acc'])
    plt.title('Precisión del modelo {}'.format(num_model))
    plt.ylabel('Precisión')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving accuracy plot...')
    plt.savefig(saving_folder + '/' + 'Accuracy_plot best model {}.png'.format(saved_model))

    # summarize history for loss
    plt.figure(2)
    plt.plot(saved_history['loss'])
    plt.plot(saved_history['val_loss'])
    plt.title('Pérdida logarítmica del modelo {}'.format(num_model))
    plt.ylabel('Pérdida logaritmica')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving loss plot...')
    plt.savefig(saving_folder + '/' + 'Loss_plot best model {}.png'.format(saved_model))

    # summarize history for F1 score
    plt.figure(3)
    plt.plot(saved_history['f1'])
    plt.plot(saved_history['val_f1'])
    plt.title('Puntaje F1 del modelo {}'.format(num_model))
    plt.ylabel('Puntaje F1')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving F1 score plot...')
    plt.savefig(saving_folder + '/' + 'F1_score_plot best model {}.png'.format(saved_model))
```

```
        # summarize history for sensitivity
        plt.figure(4)
        plt.plot(saved_history['specificity'])
        plt.plot(saved_history['val_specificity'])
        plt.title('Especificidad del modelo {}'.format(num_model))
        plt.ylabel('Especificidad [ TP / (TP + FN)  ]')
        plt.xlabel('Iteración')
        plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
        plt.show()
        print('Saving specificity plot...')
        plt.savefig(saving_folder + '/' + 'specificity_plot best model {}.png'.format(saved_model))

        # summarize history for specificity
        plt.figure(5)
        plt.plot(saved_history['sensitivity'])
        plt.plot(saved_history['val_sensitivity'])
        plt.title('Sensibilidad del modelo {}'.format(num_model))
        plt.ylabel('Sensibilidad [ TN / (TN + FP)  ]')
        plt.xlabel('Iteración')
        plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
        plt.show()
        print('Saving sensitivity plot...')
        plt.savefig(saving_folder + '/' + 'sensitivity_plot best model {}.png'.format(saved_model))

    # ---------------------------------- GENERATOR TRAINING ---------------------------------- #

    else:

        # Train and validate the best model a little bit longer
        history = model.fit_generator(train_generator,
                                      epochs=train_epochs,
                                      validation_data=validation_generator,
                                      steps_per_epoch=steps_per_epoch,
                                      validation_steps=validation_steps,
                                      verbose=1,
                                      use_multiprocessing=False,
                                      workers=20,
                                      max_queue_size=20,
                                      class_weight=class_weight,
                                      callbacks=callbacks)

        # Get history file path
        history_file = saving_folder + '/model_history {}'.format(saved_model) + '.p'

        # Save history of training
        saved_history = history.history
        pickle.dump(saved_history, open(history_file, "wb"))

        # Save model architecture allowing it to be used later
        print('Saving best model after {}...'.format(model_name))
        model.save(saving_folder + '/best model {}'.format(saved_model) + '.h5')

        # ------------------------------ COMPUTE AUC -------------------------------#

        if compute_AUC:
            # Get tpr_file file path
            tpr_file = saving_folder + '/tp_rate model {}'.format(saved_model) + '.p'

            # Get fpr_file file path
            fpr_file = saving_folder + '/fp_rate model {}'.format(saved_model) + '.p'

            # Get AUC file path
            auc_file = saving_folder + '/auc model {}'.format(saved_model) + '.p'

            # Get thresholds file path
            thresholds_file = saving_folder + '/thresholds model {}'.format(saved_model) + '.p'

            # Get optimal threshold file path
            opt_threshold_file = saving_folder + '/optimal threshold model {}'.format(saved_model) + '.p'

            # Compute predictions
            print('')
            print('Computing predictions: ')
            print('')

            true_labels = []
            num = 0

            print('Predicting instances...\n')

            if 'y_test_g' not in globals() or 'x_test_g' not in globals():

                files_test_pos = os.listdir(data_folder_test + '/' + positive_data)
                files_test_neg = os.listdir(data_folder_test + '/' + negative_data)

                x_test_pos = np.array([np.load(data_folder_test + '/' + positive_data + '/' + x) for x in files_test_pos])
                x_test_neg = np.array([np.load(data_folder_test + '/' + negative_data + '/' + x) for x in files_test_neg])

                x_test_g = np.concatenate((x_test_pos, x_test_neg), axis=0)
                x_test_g = np.expand_dims(x_test_g, axis=4)
                y_test_g = np.concatenate((np.ones((x_test_pos.shape[0], 1)), np.zeros((x_test_neg.shape[0], 1))), axis=0)
```

```
            # Shuffle training data
            indices = [i for i in range(x_train_g.shape[0])]
            shuffle(indices)
            x_train_g = x_train_g[indices]
            y_train_g = y_train_g[indices]

        predictions = model.predict(x_test_g)

        # Compute AUC
        print('Computing AUC... \n')
        fp_rate, tp_rate, thresholds = roc_curve(y_test_g, predictions)
        auc = auc(fp_rate, tp_rate)

        # Determine best threshold value: tpr high and fpr low
        # tf = tpr - (1-fpr) is zero or near to zero is the optimal cut off point
        i = np.arange(len(tp_rate))  # index for table
        roc = pd.DataFrame({'fp_rate': pd.Series(fp_rate, index=i), 'tp_rate': pd.Series(tp_rate, index=i),
                            '1-fp_rate': pd.Series(1 - fp_rate, index=i),
                            'tf': pd.Series(tp_rate - (1 - fp_rate), index=i),
                            'threshold': pd.Series(thresholds, index=i)})
        roc_t = roc.ix[(roc.tf - 0).abs().argsort()[:1]]
        opt_threshold = list(roc_t['threshold'])

        # Save results
        pickle.dump(tp_rate, open(tpr_file, "wb"))
        pickle.dump(fp_rate, open(fpr_file, "wb"))
        pickle.dump(thresholds, open(thresholds_file, "wb"))
        pickle.dump(auc, open(auc_file, "wb"))
        pickle.dump(opt_threshold, open(opt_threshold_file, "wb"))

        # Plot AUC
        plt.figure(7)
        plt.plot([0, 1], [0, 1], 'k--')
        plt.plot(fp_rate, tp_rate, label='Área = {:.3f}'.format(auc))
        plt.xlabel('False positive rate')
        plt.ylabel('True positive rate')
        plt.title('Curva ROC')
        plt.legend(loc='best')
        plt.show()
        plt.savefig(saving_folder + '/' + 'AUC best model {}.png'.format(saved_model))

        # Graficar mejor umbral
        fig, ax = pl.subplots()
        pl.plot(roc['tp_rate'])
        pl.plot(roc['1-fp_rate'], color='red')
        pl.xlabel('1-False Positive Rate')
        pl.ylabel('True Positive Rate')
        pl.title('Receiver operating characteristic')
        ax.set_xticklabels([])
        plt.savefig(saving_folder + '/' + 'Mejor Umbral best model {}.png'.format(saved_model))

    # ------------------------------ PLOT RESULTS -------------------------------#

    # summarize history for accuracy
    plt.figure(1)
    plt.plot(saved_history['acc'])
    plt.plot(saved_history['val_acc'])
    plt.title('Precisión del modelo {}'.format(num_model))
    plt.ylabel('Precisión')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving accuracy plot...')
    plt.savefig(saving_folder + '/' + 'Accuracy_plot best model {}.png'.format(saved_model))

    # summarize history for loss
    plt.figure(2)
    plt.plot(saved_history['loss'])
    plt.plot(saved_history['val_loss'])
    plt.title('Pérdida logarítmica del modelo {}'.format(num_model))
    plt.ylabel('Pérdida logarítmica')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving loss plot...')
    plt.savefig(saving_folder + '/' + 'Loss_plot best model {}.png'.format(saved_model))

    # summarize history for F1 score
    plt.figure(3)
    plt.plot(saved_history['f1'])
    plt.plot(saved_history['val_f1'])
    plt.title('Puntaje F1 del modelo {}'.format(num_model))
    plt.ylabel('Puntaje F1')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving F1 score plot...')
    plt.savefig(saving_folder + '/' + 'F1_score_plot best model {}.png'.format(saved_model))

    # summarize history for sensitivity
    plt.figure(4)
    plt.plot(saved_history['specificity'])
    plt.plot(saved_history['val_specificity'])
```

```
        plt.title('Especificidad del modelo {}'.format(num_model))
        plt.ylabel('Especificidad [ TP / (TP + FN)  ]')
        plt.xlabel('Iteración')
        plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
        plt.show()
        print('Saving specificity plot...')
        plt.savefig(saving_folder + '/' + 'specificity_plot best model {}.png'.format(saved_model))

        # summarize history for specificity
        plt.figure(5)
        plt.plot(saved_history['sensitivity'])
        plt.plot(saved_history['val_sensitivity'])
        plt.title('Sensibilidad del modelo {}'.format(num_model))
        plt.ylabel('Sensibilidad [ TN / (TN + FP)  ]')
        plt.xlabel('Iteración')
        plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
        plt.show()
        print('Saving sensitivity plot...')
        plt.savefig(saving_folder + '/' + 'sensitivity_plot best model {}.png'.format(saved_model))
```

create_model_cancer_generative_bayesian_cross_validation.py

```
from hyperopt import hp, fmin, tpe, STATUS_OK, Trials
from classes_lidc import custom_Keras_Metrics, dataTransformer, NoduleGenerator, MalignancyGenerator
from DNN_Models import Bayesian
from keras import initializers
import time
from keras.optimizers import Adam, Adagrad, SGD
from keras.utils import plot_model
import pickle
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import roc_curve, auc
import pylab as pl
import pandas as pd
import os
from sklearn.model_selection import StratifiedKFold
from cnn_gan import DCGAN
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras.callbacks import TensorBoard
from keras.models import load_model
from random import shuffle
import gc
import autopy
import math

# ------------------------------GENERAL INPUTS -------------------------------#

# How many models to test during optimization phase
max_evals = 20
optimiz_epochs = 30  # choice epochs
num_model = 'adaptative_1'
version = 1

# Optimizer inputs
beta1 = 0.9
beta2 = 0.999
epsilon = 0.00000001

# Data inputs
detector = 'Cancer'
data_dims = 4
batch_size = 32

# Classes
positive_data = 'malignant_adaptative_1_8_3_thr0.1 0.65'
negative_data = 'benign_adaptative_1_8_3_thr0.1 0.65'

take_screen_shot = False

# Cross validation
cv_validate = False
cv_search = True
k_folds = 10
cv = cv_validate or cv_search

# Load saved trials object or new
load_saved_trials = False
trials_steps = 5  # Additional trials

# Load saved model to keep training it or load last best model training
load_saved = False
load_best = False
load_checkpoint = False
num_saved_model = 19
saved_checkpoint = 0

# How many epochs will the best model train?
train_epochs = 50
train_batch_size = 32

# Define the loss calculation
loss = 'binary_crossentropy'

# Calculate AUC
compute_AUC = True

# Assign more weight during training to specific class?
weight_classes = False
weight_neg_ratio = 1.
weight_pos_ratio = 3.

# ------------------------------CALLBACK INPUTS -------------------------------#

# ---------- CheckPoints ------------
save_best_only = True
monitor = 'val_loss'  # Metric to monitor
saving_per_epoch = 5
```

```
mode_checkpoints = 'min'

# --------- Early Stopping ----------
use_early_stop = False
min_delta = 0.001  # Minimum improvement required
patience = 25  # num of epochs with no improvement then training will be stopped
verbose = 1
mode_early = 'auto'
baseline = None  # value for the monitored quantity to reach

# restore model weights from the epoch with the best value of the monitored quantity
restore_best_weights = False

saved_model = ''
if load_saved or load_checkpoint:
    saved_model += str(num_saved_model)

# ----------------------------OPTIMIZER INPUTS ----------------------------#

space_dict = {}

n_blocks = 5

lr_min = 0.0005
lr_max = 0.001

decay_min = 0.01
decay_max = 0.05

num_dense = [1, 2, 3, 4, 5]
num_units = 1000

l2_regularizer_min = 0.0001
l2_regularizer_max = 0.01

use_dropout = [0, 1]
dropout_min = 0
dropout_max = 0.01

# ----------------------------SPACE INPUTS --------------------------------#

space_adaptative_1 = {
    'lr': hp.uniform('lr', lr_min, lr_max),
    'decay': hp.uniform('decay', decay_min, decay_max),

    'use_dropout': hp.choice('use_dropout', use_dropout),
    'dropout': hp.uniform('dropout', dropout_min, dropout_max),

    'dense_0': hp.choice('dense_0', num_dense),
    'dense_1': hp.choice('dense_1', num_dense),
    'dense_2': hp.choice('dense_2', num_dense),
    'dense_3': hp.choice('dense_3', num_dense),
    'dense_4': hp.choice('dense_4', num_dense),

    'units_0': hp.randint('units_0', num_units),
    'units_1': hp.randint('units_1', num_units),
    'units_2': hp.randint('units_2', num_units),
    'units_3': hp.randint('units_3', num_units),
    'units_4': hp.randint('units_4', num_units),

    'l2_regularizer_0': hp.uniform('l2_regularizer_0', l2_regularizer_min, l2_regularizer_max),
    'l2_regularizer_1': hp.uniform('l2_regularizer_1', l2_regularizer_min, l2_regularizer_max),
    'l2_regularizer_2': hp.uniform('l2_regularizer_2', l2_regularizer_min, l2_regularizer_max),
    'l2_regularizer_3': hp.uniform('l2_regularizer_3', l2_regularizer_min, l2_regularizer_max),
    'l2_regularizer_4': hp.uniform('l2_regularizer_4', l2_regularizer_min, l2_regularizer_max)
}

space_bayesian = space_adaptative_1

# --------------------------- DIMENSION INPUTS -----------------------------#
# Dimensions and parameters of data
n_classes = 2
shuffle_data = True
target_size = data_dims
input_size = (data_dims,)  # For DNN

resample_data = True  # If data is not created yet, you want to resample it?
train_split = 0.7  # Percentage of training data from all data

# Folder where the model, checkpoints and settings will be saved
saving_folder = ('Models/Models {0} Detector/'
                 + 'Models of {0} Detectors 3D with Generator/'
                 + 'Model{1}_Version{2}/Size({3}) cv {4}').format(detector, num_model, version, data_dims, cv)

# Folder where data is located
data_path = 'E:/Base de datos maestría/LIDC-IDRI/Train Data/Train Data {0} Detector/{0} Detector 2 ' \
            'Classes Generator GANS/'.format(detector)

if not os.path.isdir(saving_folder):
```

```
    os.makedirs(saving_folder)

# Save screen shots
if take_screen_shot:
    if not load_best and not load_checkpoint and not load_saved and not load_saved_trials:
        for i in range(12):
            print('Enter 1 when ready to take screen shot {0}'.format(i + 1))
            take_shot = input()
            if take_shot:
                screen_shot = autopy.bitmap.capture_screen()
                screen_shot.save(saving_folder + '/screen_shot {0}.png'.format(i + 1))


# -------------------- PARAMETER INITIALIZERS INPUTS  ----------------------#

kernel_initializer = initializers.glorot_normal(seed=None)
bias_initializer = initializers.glorot_normal(seed=None)


# ---------------------------------- MODEL FOLDER --------------------------#

# Define model name based on its version and number
model_name = 'Model{}_Version{}_Size({})'.format(num_model,
                                                 str(version),
                                                 str(data_dims),
                                                 str(data_dims),
                                                 str(data_dims))
model_folder = saving_folder + '/models'

if not os.path.isdir(model_folder):
    os.makedirs(model_folder)

# -------------------------- CREATE SPACE DICTIONARY  ----------------------#

if not load_saved and not load_best and not load_checkpoint:
    space_dict['optimiz_epochs'] = optimiz_epochs

    space_dict['lr_min'] = lr_min
    space_dict['lr_max'] = lr_max

    space_dict['batch_size'] = batch_size
    space_dict['dense'] = num_dense
    space_dict['extra_units'] = num_units

    space_dict['decay_min'] = decay_min
    space_dict['decay_max'] = decay_max

    space_dict['L2_min'] = l2_regularizer_min
    space_dict['L2_max'] = l2_regularizer_max

    space_dict['use_dropout'] = use_dropout
    space_dict['dropout_min'] = dropout_min
    space_dict['dropout_max'] = dropout_max

    space_dict['beta1'] = beta1
    space_dict['beta2'] = beta2

    space_dict['epsilon'] = epsilon

    print('Saving space dictionary')
    pickle.dump(space_dict, open(saving_folder + "/search_space_dict.p", "wb"))
    df = pd.DataFrame.from_dict(space_dict, orient='index')
    df.to_excel(saving_folder + '/search_space_dict.xlsx')

# ------------------------------- LOAD DATA ------------------------------#

# Load data
x_pos = np.load(data_path + positive_data + '.npy')
x_neg = np.load(data_path + negative_data + '.npy')
x_data = np.concatenate([x_pos, x_neg], axis=0)
y_data = np.concatenate((np.ones((x_data.shape[0], 1)), np.zeros((x_data.shape[0], 1))), axis=0)

# Shuffle data
indices = [i for i in range(x_data.shape[0])]
shuffle(indices)
x_data = x_data[indices]
y_data = y_data[indices]

# Training data
x_train_g = x_data[0:math.floor(0.7*x_data.shape[0])]
y_train_g = y_data[0:math.floor(0.7*y_data.shape[0])]

# Training data
x_test_g = x_data[0:math.floor(0.7*x_data.shape[0])]
y_test_g = y_data[0:math.floor(0.7*y_data.shape[0])]

if cv_search:
    # Get cross validation folds
    folds = list(StratifiedKFold(n_splits=k_folds, shuffle=True).split(x_data, y_data))
```

```
# --------------------------------- OBJECTIVE -------------------------------#

# Specify if the training will give more weight to either positive or negative
if weight_classes:
    class_weight = {0: weight_neg_ratio,
                    1: weight_pos_ratio}
else:
    class_weight = {0: 1.,
                    1: 1.}

# Define the metrics that will be calculated in the model
f1 = custom_Keras_Metrics.f1
sens = custom_Keras_Metrics.sensitivity
spec = custom_Keras_Metrics.specificity
fpr = custom_Keras_Metrics.falsePositiveRate
metrics = ['accuracy', f1, sens, spec, fpr]

# Get the model from file: "classes_lidc"
model_str = 'model_' + str(num_model)
model_method = getattr(Bayesian, model_str)

if not load_saved and not load_best and not load_checkpoint:
    n_trial = 0
else:
    n_trial = saved_checkpoint + 1

# Loading trials
if not os.path.isdir(saving_folder + '/logs'):
    os.makedirs(saving_folder + '/logs')

# Star from zero
if not load_saved_trials:
    trials = Trials()
else:
    try:
        trials = pickle.load(open(saving_folder + '/trials.p', "rb"))
        max_evals = len(trials.trials) + max_evals
    except ValueError as error:
        print(error)
        trials = Trials()

def objective(space):
    """
    Model providing function:
    Create Keras model with double curly brackets dropped-in as needed.
    Return value has to be a valid python dictionary with two customary keys:
        - loss: Specify a numeric evaluation metric to be minimized
        - status: Just use STATUS_OK and see hyperopt documentation if not feasible
    The last one is optional, though recommended, namely:
        - model: specify the model just created so that we can later use it again.
    """

    # ------------------------------ OPTIMIZER # ------------------------------#
    global n_trial, trials, k_folds, cv

    callbacks_temp = []

    optimizer = Adam(lr=space['lr'],
                     beta_1=beta1,
                     beta_2=beta2,
                     epsilon=epsilon,
                     decay=space['decay'])

    model_temp = model_method(input_shape=input_size,
                              space=space,
                              n_blocks=n_blocks,
                              kernel_initializer=kernel_initializer,
                              bias_initializer=bias_initializer)

    # Compile model
    model_temp.compile(optimizer=optimizer,
                       loss=loss,
                       metrics=metrics)

    # Show model summary
    model_temp.summary()

    # ---------------------------------- CROSS VALIDATION ----------------------------------
    if cv_search:

        global x_data, y_data

        # Get cross validation folds
        folds = list(StratifiedKFold(n_splits=k_folds, shuffle=True).split(x_data, y_data))

        # Initialize a list of cross validation losses
        cross_losses = []
```

```
        # Train model in each fold of cross validation
        for j, (train_idx, val_idx) in enumerate(folds):

            print('\nFold ', j)

            # Create TensorBoard Callback
            callbacks_temp = []
            name = 'trial {0} cv {1} '.format(n_trial, j) + str(int(time.time()))
            tensorboard_temp = TensorBoard(log_dir=saving_folder + '/logs/{}'.format(name))
            callbacks_temp.append(tensorboard_temp)

            # Get data for specific k fold
            x_train = x_data[train_idx]
            y_train = y_data[train_idx]
            x_test = x_data[val_idx]
            y_test = y_data[val_idx]

            # Get positive data
            pos_idx = np.where(y_train[:, 0] == 1)
            x_pos = x_train[pos_idx]
            num_pos = x_pos.shape[0]

            # Get negative data
            neg_idx = np.where(y_train[:, 0] == 0)
            x_neg = x_train[neg_idx]
            num_neg = x_neg.shape[0]

            # Release memory
            del x_train
            gc.collect()

            # Create augmented labels for training data and concatenate positive with negative
            x_train = np.concatenate((x_pos, x_neg), axis=0)
            y_train = np.concatenate((np.ones((x_pos.shape[0], 1)), np.zeros((x_neg.shape[0], 1))), axis=0)

            # Release memory
            del x_pos, x_neg
            gc.collect()

            # Shuffle augmented training data
            indices = [i for i in range(x_train.shape[0])]
            shuffle(indices)
            x_train = x_train[indices, :]
            y_train = y_train[indices, :]

            # Train and validate model
            hist = model_temp.fit(x=x_train,
                                  y=y_train,
                                  batch_size=batch_size,
                                  epochs=optimiz_epochs,
                                  validation_data=(x_test, y_test),
                                  verbose=1,
                                  class_weight=class_weight,
                                  callbacks=callbacks_temp)

            # Release memory
            del x_test, x_train
            gc.collect()

            # Save validation loss for this k-fold
            val_loss = hist.history['val_loss'][-1]
            cross_losses.append(val_loss)
            print('Validation loss of {0} fold: {1}', j, val_loss)

        # Get average of all losses
        val_loss = np.mean(cross_losses)
        print('Average cross validation loss: ', val_loss)

    # ---------------------------------- NO CROSS VALIDATION ----------------------------------
    else:

        # Create TensorBoard Callback
        name = 'trial {} '.format(n_trial) + str(int(time.time()))
        tensorboard_temp = TensorBoard(log_dir=saving_folder + '/logs/{}'.format(name))
        callbacks_temp.append(tensorboard_temp)

        indices = [i for i in range(x_data.shape[0])]
        shuffle(indices)
        x_data = x_data[indices, :]
        y_data = y_data[indices, :]

        # Get data for specific k fold
        x_train = x_data[0:math.ceil(train_split * x_data.shape[0])]
        y_train = y_data[0:math.ceil(train_split * y_data.shape[0])]
        x_test = x_data[math.floor(train_split * x_data.shape[0]):]
        y_test = y_data[math.floor(train_split * y_data.shape[0]):]

        # Get positive data
```

```
        pos_idx = np.where(y_train[:, 0] == 1)
        x_pos = x_train[pos_idx]
        num_pos = x_pos.shape[0]

        # Get negative data
        neg_idx = np.where(y_train[:, 0] == 0)
        x_neg = x_train[neg_idx]
        num_neg = x_neg.shape[0]

        # Release memory
        del x_train
        gc.collect()

        # Create augmented labels for training data and concatenate positive with negative
        x_train = np.concatenate((x_pos, x_neg), axis=0)
        y_train = np.concatenate((np.ones((x_pos.shape[0], 1)), np.zeros((x_neg.shape[0], 1))), axis=0)

        # Release memory
        del x_pos, x_neg
        gc.collect()

        # Shuffle augmented training data
        indices = [i for i in range(x_train.shape[0])]
        shuffle(indices)
        x_train = x_train[indices, :]
        y_train = y_train[indices, :]

        # Train and validate model
        hist = model_temp.fit(x=x_train,
                              y=y_train,
                              batch_size=space['batch_size'],
                              epochs=space['epochs'] + 1,
                              validation_data=(x_test, y_test),
                              verbose=1,
                              class_weight=class_weight,
                              callbacks=callbacks_temp)

        # Release memory
        del x_test, x_train
        gc.collect()

        # Trying to minimize the last validation loss at the end of epoch
        val_loss = hist.history['val_loss'][-1]
        print('Validation loss: ', val_loss)

    # Save model
    model_temp.save(model_folder + '/' + 'model ' + str(n_trial) + '.h5')

    # Save trials object
    pickle.dump(trials, open(saving_folder + "/trials.p", "wb"))

    n_trial += 1

    return {'loss': val_loss, 'n_trial': n_trial, 'status': STATUS_OK}


# ------------------------------- MAIN PROGRAM -------------------------------#

# Initialize model
model = None

# If we are not training but searching
if not load_saved and not load_best and not load_checkpoint:

    # ------------------------ SEARCH FOR BEST MODEL ------------------------#
    best = fmin(objective,
                space=space_bayesian,
                algo=tpe.suggest,
                max_evals=max_evals,
                trials=trials)

    # ---------------------- SAVE OPTIMIZATION RESULTS ----------------------#

    # Save all the specified parameters of this training in a '.txt' file
    optimization_folder = (saving_folder + '/optimization_results.xlsx')

    opt_results = {'loss': [x['loss'] for x in trials.results]}

    for key, value in trials.idxs_vals[1].items():
        opt_results[key] = trials.idxs_vals[1][key]

    decay_results = pd.DataFrame(opt_results)
    decay_results.to_excel(optimization_folder)

    # ---------------------------- SAVE BEST MODEL ---------------------------#

    # Get best model
    n_model = trials.best_trial['result']['n_trial']
    best = trials.best_trial['misc']['vals']
```

```
    model_path = (model_folder
                  + '/model '
                  + str(n_model - 1)
                  + '.h5')
    model = load_model(model_path, custom_objects={'f1': f1,
                                                   'sensitivity': sens,
                                                   'specificity': spec,
                                                   'falsePositiveRate': fpr})

    # Save the model architecture as image
    plot_model(model, to_file=saving_folder + '/architecture.png')

    # Print the details of the CNN architecture and save it
    model.summary()
    summary_folder = (saving_folder + '/model_summary.txt')
    with open(summary_folder, 'w') as f:
        # Pass the file handle in as a lambda function to make it callable
        model.summary(print_fn=lambda x: f.write(x + '\n'))

    # Save model architecture allowing it to be used later
    print('Saving model {}...'.format(model_name))
    model.save(saving_folder + '/best model ' + str(n_model) + '.h5')

    # ----------------------------- SAVE TRIALS -------------------------------#

    # Save trials object
    pickle.dump(trials, open(saving_folder + "/trials.p", "wb"))

    # -------------------------- SAVE BEST MODEL SETTINGS  --------------------#

    # Save all the specified parameters of this training in a '.txt' file
    parameters_folder = (saving_folder + '/parameters.txt')

    if not os.path.isdir(saving_folder):
        os.makedirs(saving_folder)

    f = open(parameters_folder, 'w')
    f.write('------------ BAYESIAN OPTIMIZATION BEST MODEL ------------\n')
    f.write('\n')

    f.write('TRAINING PARAMETERS\n')
    f.write('   epochs: {}\n'.format(str(optimiz_epochs[best['epochs'][0]])))
    f.write('   batch_size: {}\n'.format(str(batch_size[best['batch_size'][0]])))
    f.write('   learning_rate: {}\n'.format(str(best['lr'][0])))
    f.write('   decay: {}\n'.format(str(best['decay'][0])))
    f.write('   extra dense layers: {}\n'.format(str(best['extra_layers'][0])))
    f.write('   extra units for dense layers : {}\n'.format(str(best['extra_units'][0])))
    f.write('\n')

    f.write('OPTIMIZER PARAMETERS\n')
    f.write('   beta 1: {}\n'.format(str(beta1[best['beta1'][0]])))
    f.write('   beta 2: {}\n'.format(str(beta2[best['beta2'][0]])))
    f.write('   epsilon: {}\n'.format(str(epsilon[best['epsilon'][0]])))
    f.write('\n')

    f.write('DATA PARAMETERS\n')
    f.write('   suffle data in every batch: {}\n'.format(str(shuffle_data)))
    f.write('   input_size: {}\n'.format(str(input_size)))
    f.write('   train_split: {}\n'.format(str(train_split)))
    f.write('\n')

    f.close()

# ------------------------------ CALLBACKS SAVING  ---------------------------#

callbacks = []

# Create checkpoints folder for best model
if not os.path.isdir(saving_folder + '/' + 'best model checkpoints{}'.format(saved_model)):
    os.makedirs(saving_folder + '/' + 'best model checkpoints{}'.format(saved_model))

# Initialize frequency and folder to save checkpoints
checkpoint_folder = (saving_folder
                     + '/best model checkpoints{}'.format(saved_model)
                     + '/{epoch:02d}.hd5f')

checkpoints = ModelCheckpoint(checkpoint_folder,
                              monitor=monitor,
                              save_weights_only=False,
                              save_best_only=save_best_only,
                              period=saving_per_epoch,
                              mode='min',
                              verbose=1)
callbacks.append(checkpoints)

# Tensorboard callbacks
tensorboard = TensorBoard(log_dir=saving_folder + '/logs/{}'.format('best_model{}'.format(saved_model)))
callbacks.append(tensorboard)
```

```
# Create early stopping callback
if use_early_stop:
    earyly_stopping = EarlyStopping(monitor=monitor,
                                    min_delta=min_delta,
                                    patience=patience,
                                    verbose=verbose,
                                    mode=mode_early,
                                    baseline=baseline,
                                    restore_best_weights=restore_best_weights)
    callbacks.append(earyly_stopping)

# ---------------------------- KEEP TRAINING BEST MODEL ----------------------#

# Load previous version
if load_saved:
    model_path = (model_folder
                  + '/model '
                  + str(saved_model)
                  + '.h5')
    model = load_model(model_path, custom_objects={'f1': f1,
                                                   'sensitivity': sens,
                                                   'specificity': spec,
                                                   'falsePositiveRate': fpr})

if load_best or load_checkpoint:
    model_path = (saving_folder
                  + '/best model checkpoints{}'.format(saved_model)
                  + '/'
                  + str(saved_checkpoint)
                  + '.hd5f')
    model = load_model(model_path, custom_objects={'f1': f1,
                                                   'sensitivity': sens,
                                                   'specificity': spec,
                                                   'falsePositiveRate': fpr})

# ---------------------------------- CROSS VALIDATION ----------------------------------
if cv_validate:
    # Get cross validation folds
    folds = list(StratifiedKFold(n_splits=k_folds, shuffle=True).split(x_data, y_data))

    # Initialize list of metrics
    cross_histories, cross_val_losses = [], []
    tp_rate = []
    fp_rate = []
    thresholds_avg = []
    roc_aucs = []

    # ------------------------------ ITERATE CROSS-VALIDATION -------------------------------
    for j, (train_idx, val_idx) in enumerate(folds):

        print('\nFold ', j)

        # ------------------------------ CALLBACKS -----------------------------
        callbacks = []

        checkpoint_folder = (saving_folder
                             + '/best model checkpoints{0}/cv{1} '.format(saved_model, j)
                             + '{epoch:02d}.hd5f')

        checkpoints = ModelCheckpoint(checkpoint_folder,
                                      monitor=monitor,
                                      save_weights_only=False,
                                      save_best_only=save_best_only,
                                      period=saving_per_epoch,
                                      mode='min',
                                      verbose=1)
        callbacks.append(checkpoints)

        # Tensorboard callbacks
        tensorboard = TensorBoard(
            log_dir=saving_folder + '/logs/{}'.format('best_model{0}cv{1}'.format(saved_model, j)))
        callbacks.append(tensorboard)

        # ------------------------------ DATA ----------------------------

        # Get data for specific k fold
        x_train = x_data[train_idx]
        y_train = y_data[train_idx]
        x_test = x_data[val_idx]
        y_test = y_data[val_idx]

        # Get positive data
        pos_idx = np.where(y_train[:, 0] == 1)
        x_pos = x_train[pos_idx]
        num_pos = x_pos.shape[0]

        # Get negative data
        neg_idx = np.where(y_train[:, 0] == 0)
        x_neg = x_train[neg_idx]
```

```
        num_neg = x_neg.shape[0]

        # Release memory
        del x_train
        gc.collect()

        # Create augmented labels for training data and concatenate positive with negative
        x_train = np.concatenate((x_pos, x_neg), axis=0)
        y_train = np.concatenate((np.ones((x_pos.shape[0], 1)), np.zeros((x_neg.shape[0], 1))), axis=0)

        # Release memory
        del x_pos, x_neg
        gc.collect()

        # Shuffle augmented training data
        indices = [i for i in range(x_train.shape[0])]
        shuffle(indices)
        x_train = x_train[indices, :]
        y_train = y_train[indices, :]

        # ------------------------------ TRAINING -----------------------------

        # Train and validate model
        history = model.fit(x=x_train,
                            y=y_train,
                            batch_size=train_batch_size,
                            epochs=train_epochs,
                            validation_data=(x_test, y_test),
                            class_weight=class_weight,
                            verbose=1,
                            callbacks=callbacks)

        # Release memory
        del x_train
        gc.collect()

        # Save validation loss for this k-fold
        cross_histories.append(history.history)
        val_loss = history.history['val_loss'][-1]
        cross_val_losses.append(val_loss)
        print('Validation loss of {0} fold: {1}', j, val_loss)

        # ------------------------------ COMPUTE AUC -------------------------------#
        if compute_AUC:
            # Compute predictions
            print('Predicting instances...\n')
            true_labels = []
            num = 0
            predictions = model.predict(x_test)

            # Compute AUC
            print('Computing AUC... \n')
            fp_rate, tp_rate, thresholds = roc_curve(y_test, predictions)

            # Get tpr_file file path
            tpr_file = saving_folder + '/tp_rate model {} cv'.format(saved_model) + str(j) + '.p'

            # Get fpr_file file path
            fpr_file = saving_folder + '/fp_rate model {} cv'.format(saved_model) + str(j) + '.p'

            # Get AUC file path
            auc_file = saving_folder + '/auc model {} cv'.format(saved_model) + str(j) + '.p'

            # Get thresholds file path
            thresholds_file = saving_folder + '/thresholds model {} cv'.format(saved_model) + str(j) + '.p'

            # Get optimal threshold file path
            opt_threshold_file = saving_folder + '/optimal threshold model {} cv'.format(saved_model) + str(j) + '.p'

            roc_auc = auc(fp_rate, tp_rate)

            roc_aucs.append(roc_auc)

            # Determine best threshold value: tpr high and fpr low
            # tf = tpr - (1-fpr) is zero or near to zero is the optimal cut off point
            k = np.arange(len(tp_rate))  # index for table
            roc = pd.DataFrame({'fp_rate': pd.Series(fp_rate, index=k),
                                'tp_rate': pd.Series(tp_rate, index=k),
                                '1-fp_rate': pd.Series(1 - fp_rate, index=k),
                                'tf': pd.Series(tp_rate - (1 - fp_rate), index=k),
                                'threshold': pd.Series(thresholds, index=k)})
            roc_t = roc.ix[(roc.tf - 0).abs().argsort()[:1]]
            opt_threshold = list(roc_t['threshold'])

            # Save results
            pickle.dump(tp_rate, open(tpr_file, "wb"))
            pickle.dump(fp_rate, open(fpr_file, "wb"))
            pickle.dump(thresholds_avg, open(thresholds_file, "wb"))
            pickle.dump(roc_auc, open(auc_file, "wb"))
```

```
            pickle.dump(opt_threshold, open(opt_threshold_file, "wb"))

            # Plot AUC
            plt.figure(j)
            plt.ylim(0, 1.2)
            plt.plot([0, 1], [0, 1], 'k--')
            plt.plot(fp_rate, tp_rate, label='Área = {:.6f}'.format(roc_auc))
            plt.xlabel('Average false positive rate')
            plt.ylabel('Average true positive rate')
            plt.title('Average ROC Curve')
            plt.legend(loc='best')
            plt.show()
            plt.savefig(saving_folder + '/' + 'AUC best model {0} cv {1}.png'.format(saved_model, j))
            plt.close(fig='all')

            # Graficar mejor umbral
            plt.figure(j)
            fig, ax = pl.subplots()
            pl.plot(roc['tp_rate'])
            pl.plot(roc['1-fp_rate'], color='red')
            pl.xlabel('Average 1-False Positive Rate')
            pl.ylabel('Average True Positive Rate')
            pl.title('Receiver operating characteristic')
            ax.set_xticklabels([])
            plt.savefig(saving_folder + '/' + 'Mejor Umbral best model {0} cv {1}.png'.format(saved_model, j))
            plt.close(fig='all')

        # Release memory
        del x_test
        gc.collect()

    # ------------------------------ SAVE RESULTS -------------------------------#

    # Get history file path
    history_file = saving_folder + '/model_cross_histories {}'.format(saved_model) + '.p'
    pickle.dump(cross_histories, open(history_file, "wb"))

    # Get mean AUC
    auc_file = saving_folder + '/mean_AUC {}'.format(saved_model) + '.p'
    pickle.dump(np.mean(roc_aucs), open(auc_file, "wb"))

    # Get average of all losses
    val_loss = np.mean(cross_val_losses)

    print('Average cross validation loss: ', val_loss)

    # Save model architecture allowing it to be used later
    print('Saving best model after {}...'.format(model_name))
    model.save(saving_folder + '/best model {}'.format(saved_model) + '.h5')

    # ------------------------------ PLOT RESULTS -------------------------------#

    # summarize history for accuracy
    avg_data = np.sum(np.array([h['acc'] for h in cross_histories]), axis=0) / k_folds
    avg_val = np.sum(np.array([h['val_acc'] for h in cross_histories]), axis=0) / k_folds

    plt.figure(1)
    plt.plot(avg_data)
    plt.plot(avg_val)
    plt.title('Precisión promedio del modelo {}'.format(num_model))
    plt.ylabel('Precisión')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving accuracy plot...')
    plt.savefig(saving_folder + '/' + 'Accuracy_plot best model {}.png'.format(saved_model))

    # summarize history for loss
    avg_data = np.sum(np.array([h['loss'] for h in cross_histories]), axis=0) / k_folds
    avg_val = np.sum(np.array([h['val_loss'] for h in cross_histories]), axis=0) / k_folds

    plt.figure(2)
    plt.plot(avg_data)
    plt.plot(avg_val)
    plt.title('Pérdida logarítmica promedio del modelo {}'.format(num_model))
    plt.ylabel('Pérdida logarítmica')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving loss plot...')
    plt.savefig(saving_folder + '/' + 'Loss_plot best model {}.png'.format(saved_model))

    # summarize history for F1 score
    avg_data = np.sum(np.array([h['f1'] for h in cross_histories]), axis=0) / k_folds
    avg_val = np.sum(np.array([h['val_f1'] for h in cross_histories]), axis=0) / k_folds

    plt.figure(3)
    plt.plot(avg_data)
    plt.plot(avg_val)
```

```
    plt.title('Puntaje F1 promedio del modelo {}'.format(num_model))
    plt.ylabel('Puntaje F1')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving F1 score plot...')
    plt.savefig(saving_folder + '/' + 'F1_score_plot best model {}.png'.format(saved_model))

    # summarize history for sensitivity
    avg_data = np.sum(np.array([h['specificity'] for h in cross_histories]), axis=0) / k_folds
    avg_val = np.sum(np.array([h['val_specificity'] for h in cross_histories]), axis=0) / k_folds

    plt.figure(4)
    plt.plot(avg_data)
    plt.plot(avg_val)
    plt.title('Especificidad promedio del modelo {}'.format(num_model))
    plt.ylabel('Especificidad [ TP / (TP + FN)  ]')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving specificity plot...')
    plt.savefig(saving_folder + '/' + 'specificity_plot best model {}.png'.format(saved_model))

    # summarize history for specificity
    avg_data = np.sum(np.array([h['sensitivity'] for h in cross_histories]), axis=0) / k_folds
    avg_val = np.sum(np.array([h['val_sensitivity'] for h in cross_histories]), axis=0) / k_folds

    plt.figure(5)
    plt.plot(avg_data)
    plt.plot(avg_val)
    plt.title('Sensibilidad promedio del modelo {}'.format(num_model))
    plt.ylabel('Sensibilidad [ TN / (TN + FP)  ]')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving sensitivity plot...')
    plt.savefig(saving_folder + '/' + 'sensitivity_plot best model {}.png'.format(saved_model))

# ---------------------------------- NO CROSS VALIDATION ----------------------------------

else:

    # Train and validate model
    history = model.fit(x=x_train_g,
                        y=y_train_g,
                        batch_size=train_batch_size,
                        epochs=train_epochs,
                        validation_data=(x_test_g, y_test_g),
                        class_weight=class_weight,
                        verbose=1,
                        callbacks=callbacks)

    # Get history file path
    history_file = saving_folder + '/model_history {}'.format(saved_model) + '.p'

    # Save history of training
    saved_history = history.history
    pickle.dump(saved_history, open(history_file, "wb"))

    # Save model architecture allowing it to be used later
    print('Saving best model after {}...'.format(model_name))
    model.save(saving_folder + '/best model {}'.format(saved_model) + '.h5')

    # ------------------------------ COMPUTE AUC --------------------------------#

    if compute_AUC:
        # Get tpr_file file path
        tpr_file = saving_folder + '/tp_rate model {}'.format(saved_model) + '.p'

        # Get fpr_file file path
        fpr_file = saving_folder + '/fp_rate model {}'.format(saved_model) + '.p'

        # Get AUC file path
        auc_file = saving_folder + '/auc model {}'.format(saved_model) + '.p'

        # Get thresholds file path
        thresholds_file = saving_folder + '/thresholds model {}'.format(saved_model) + '.p'

        # Get optimal threshold file path
        opt_threshold_file = saving_folder + '/optimal threshold model {}'.format(saved_model) + '.p'

        # Compute predictions
        print('')
        print('Computing predictions: ')
        print('')

        true_labels = []
        num = 0
```

```
        print('Predicting instances...\n')
        predictions = model.predict(x_test_g)

        # Compute AUC
        print('Computing AUC... \n')
        fp_rate, tp_rate, thresholds = roc_curve(y_test_g, predictions)
        auc = auc(fp_rate, tp_rate)

        # Determine best threshold value: tpr high and fpr low
        # tf = tpr - (1-fpr) is zero or near to zero is the optimal cut off point
        i = np.arange(len(tp_rate))  # index for table
        roc = pd.DataFrame({'fp_rate': pd.Series(fp_rate, index=i), 'tp_rate': pd.Series(tp_rate, index=i),
                            '1-fp_rate': pd.Series(1 - fp_rate, index=i),
                            'tf': pd.Series(tp_rate - (1 - fp_rate), index=i),
                            'threshold': pd.Series(thresholds, index=i)})
        roc_t = roc.ix[(roc.tf - 0).abs().argsort()[:1]]
        opt_threshold = list(roc_t['threshold'])

        # Save results
        pickle.dump(tp_rate, open(tpr_file, "wb"))
        pickle.dump(fp_rate, open(fpr_file, "wb"))
        pickle.dump(thresholds, open(thresholds_file, "wb"))
        pickle.dump(auc, open(auc_file, "wb"))
        pickle.dump(opt_threshold, open(opt_threshold_file, "wb"))

        # Plot AUC
        plt.figure(7)
        plt.plot([0, 1], [0, 1], 'k--')
        plt.plot(fp_rate, tp_rate, label='Área = {:.3f}'.format(auc))
        plt.xlabel('False positive rate')
        plt.ylabel('True positive rate')
        plt.title('Curva ROC')
        plt.legend(loc='best')
        plt.show()
        plt.savefig(saving_folder + '/' + 'AUC best model {}.png'.format(saved_model))

        # Graficar mejor umbral
        fig, ax = pl.subplots()
        pl.plot(roc['tp_rate'])
        pl.plot(roc['1-fp_rate'], color='red')
        pl.xlabel('1-False Positive Rate')
        pl.ylabel('True Positive Rate')
        pl.title('Receiver operating characteristic')
        ax.set_xticklabels([])
        plt.savefig(saving_folder + '/' + 'Mejor Umbral best model {}.png'.format(saved_model))

    # ------------------------------ PLOT RESULTS -------------------------------#

    # summarize history for accuracy
    plt.figure(1)
    plt.plot(saved_history['acc'])
    plt.plot(saved_history['val_acc'])
    plt.title('Precisión del modelo {}'.format(num_model))
    plt.ylabel('Precisión')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving accuracy plot...')
    plt.savefig(saving_folder + '/' + 'Accuracy_plot best model {}.png'.format(saved_model))

    # summarize history for loss
    plt.figure(2)
    plt.plot(saved_history['loss'])
    plt.plot(saved_history['val_loss'])
    plt.title('Pérdida logarítmica del modelo {}'.format(num_model))
    plt.ylabel('Pérdida logarítmica')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving loss plot...')
    plt.savefig(saving_folder + '/' + 'Loss_plot best model {}.png'.format(saved_model))

    # summarize history for F1 score
    plt.figure(3)
    plt.plot(saved_history['f1'])
    plt.plot(saved_history['val_f1'])
    plt.title('Puntaje F1 del modelo {}'.format(num_model))
    plt.ylabel('Puntaje F1')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving F1 score plot...')
    plt.savefig(saving_folder + '/' + 'F1_score_plot best model {}.png'.format(saved_model))

    # summarize history for sensitivity
    plt.figure(4)
    plt.plot(saved_history['specificity'])
    plt.plot(saved_history['val_specificity'])
    plt.title('Especificidad del modelo {}'.format(num_model))
```

```
    plt.ylabel('Especificidad [ TP / (TP + FN)  ]')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving specificity plot...')
    plt.savefig(saving_folder + '/' + 'specificity_plot best model {}.png'.format(saved_model))

    # summarize history for specificity
    plt.figure(5)
    plt.plot(saved_history['sensitivity'])
    plt.plot(saved_history['val_sensitivity'])
    plt.title('Sensibilidad del modelo {}'.format(num_model))
    plt.ylabel('Sensibilidad [ TN / (TN + FP)  ]')
    plt.xlabel('Iteración')
    plt.legend(['datos de entrenamiento', 'datos de validación'], loc='upper left')
    plt.show()
    print('Saving sensitivity plot...')
    plt.savefig(saving_folder + '/' + 'sensitivity_plot best model {}.png'.format(saved_model))
```

slide_nodule_detector_3D.py

```
from classes_lidc import Utilities, PreProcessor, Detector, custom_Keras_Metrics
from keras.models import load_model, Model
import tensorflow as tf
import os
import time
from sklearn.externals import joblib
import pandas as pd

# ----------------------------GENERAL INPUTS ------------------------------#

# Model Inputs
num_model = 'adaptative_1'
version = 8
detector = 'Nodule'
model_name = 'best model'
cv = True

# FPR Model Inputs
use_fp_reduction = True
num_model_2 = 'adaptative_1'
version_2 = '8'
detector_2 = 'FP Reduction'
model_name_2 = 'best model'
cv_2 = True

# SVM for FPR Model Inputs
use_svm = False
num_model_svm = 'adaptative_1'
version_svm = '8_2'
cv_svm = False
detector_svm = 'FP Reduction'

# Malignancy Model Inputs
use_malignancy = True
num_model_3 = 'adaptative_1'
version_3 = 1
detector_3 = 'Malignancy'
model_name_3 = 'best model 14'
cv_3 = True

# Spiculation Model Inputs
use_spiculation = True
num_model_4 = 'adaptative_1'
version_4 = 1
detector_4 = 'Spiculation'
model_name_4 = 'best model'
cv_4 = True

# Lobulation Model Inputs
use_lobulation = True
num_model_5 = 'adaptative_1'
version_5 = 1
detector_5 = 'Lobulation'
model_name_5 = 'best model'
cv_5 = True

# Sliding inputs
target_size = (32, 32, 32)  # Get model input size
stride = (5, 10, 10)
threshold = 0.1
threshold_2 = 0.01
threshold_3 = 0.5
threshold_4 = 0.5
threshold_5 = 0.5
save_slides = True
save_fp = True
save_tp = True
graphics_show = False

# Patient inputs
patient_dir = 'E:/Base de datos maestría/LIDC-IDRI/DOI'
patient_range = (13, 22)
augment_data = False
get_diagnosed = True
num_diagnosed = 10
start_diagnosed = 16

# Same true labels assigned by how many radiologists
num_radiologists = 3

# ------------------------------- MAIN ---------------------------------#

# Filter window dimensions based on the input  that the model accepts
data_dims = target_size[0]
filter_size = target_size  # This is based on the analysis with nodule shapes

# Folder where the model, checkpoints and settings will be saved
model_folder = 'Model{0}_Version{1}/Size({2}, {2}, {2}) cv {3}'.format(num_model, version, data_dims, cv)
```

```
model_folder_2 = 'Model{0}_Version{1}/Size({2}, {2}, {2}) cv {3}'.format(num_model_2, version_2, data_dims, cv_2)
model_folder_svm = 'Model{0}_Version{1}/Size({2}, {2}, {2}) cv {3}'.format(num_model_svm, version_svm, data_dims,
                                                                           cv_svm)
model_folder_3 = 'Model{0}_Version{1}/Size({2}, {2}, {2}) cv {3}'.format(num_model_3, version_3, data_dims, cv_3)
model_folder_4 = 'Model{0}_Version{1}/Size({2}, {2}, {2}) cv {3}'.format(num_model_4, version_4, data_dims, cv_4)
model_folder_5 = 'Model{0}_Version{1}/Size({2}, {2}, {2}) cv {3}'.format(num_model_5, version_5, data_dims, cv_5)

saving_folder_slides = ('E:/Base de datos maestría/LIDC-IDRI/Models/Models {0} Detector/'
                        + 'Models of {0} Detectors 3D with Generator/'
                        + model_folder).format(detector)

saving_folder_slides_2 = ('E:/Base de datos maestría/LIDC-IDRI/Models/Models {0} Detector/'
                          + 'Models of {0} Detectors 3D with Generator/'
                          + model_folder_2).format(detector_2)

saving_folder_slides_3 = ('E:/Base de datos maestría/LIDC-IDRI/Models/Models {0} Detector/'
                          + 'Models of {0} Detectors 3D with Generator/'
                          + model_folder_3).format(detector_3)

saving_folder_slides_4 = ('E:/Base de datos maestría/LIDC-IDRI/Models/Models {0} Detector/'
                          + 'Models of {0} Detectors 3D with Generator/'
                          + model_folder_4).format(detector_4)

saving_folder_slides_5 = ('E:/Base de datos maestría/LIDC-IDRI/Models/Models {0} Detector/'
                          + 'Models of {0} Detectors 3D with Generator/'
                          + model_folder_5).format(detector_5)

# Folder where model is located
model_path = saving_folder_slides + '/' + model_name + '.h5'
model_path_2 = saving_folder_slides_2 + '/' + model_name_2 + '.h5'
model_path_3 = saving_folder_slides_3 + '/' + model_name_3 + '.h5'
model_path_4 = saving_folder_slides_4 + '/' + model_name_4 + '.h5'
model_path_5 = saving_folder_slides_5 + '/' + model_name_5 + '.h5'

# Folder where svm model is located
if use_svm:
    model_path_svm = ('E:/Base de datos maestría/LIDC-IDRI/Models/Models {0} Detector/'
                      + 'Models SVM/'
                      + model_folder_svm).format(detector_svm)

# Folder where the generated FP will be saved
if not use_fp_reduction:
    saving_folder = 'E:/Base de datos maestría/LIDC-IDRI/Train Data/Train Data {0} Detector/'.format(detector)
    saving_folder_fp = saving_folder + 'Model{0}_Version{1}/FP'.format(num_model, version)
    saving_folder_tp = saving_folder + 'Model{0}_Version{1}/TP'.format(num_model, version)
else:
    saving_folder = 'E:/Base de datos maestría/LIDC-IDRI/Train Data/Train Data {0} Detector/'.format(detector_2)
    saving_folder_fp = saving_folder + 'Model{0}_Version{1}/FP'.format(num_model_2, version_2)
    saving_folder_tp = saving_folder + 'Model{0}_Version{1}/TP'.format(num_model_2, version_2)

# Define metrics
f1 = custom_Keras_Metrics.f1
sensitivity = custom_Keras_Metrics.sensitivity
specificity = custom_Keras_Metrics.specificity
fpr = custom_Keras_Metrics.falsePositiveRate

# Load utilities and patients
detector = Detector()
u = Utilities()
p = PreProcessor()

# Get list of patients
patient_ids = os.listdir(patient_dir)
start_p = patient_range[0]
end_p = patient_range[1]

# Get diagnosed patients
if get_diagnosed:
    file_diagnosed = pd.read_excel('E:/Base de datos maestría/LIDC-IDRI/diagnosed_patients.xlsx')
    malignant = file_diagnosed['Malignant']
    benign = file_diagnosed['Benign']
    list_diagnosed = malignant.to_list()[0:num_diagnosed]
    list_diagnosed.extend(benign.to_list()[0:num_diagnosed])
    list_diagnosed = list_diagnosed[start_diagnosed:]
    list_diagnosed = [x for x in list_diagnosed if not pd.isnull(x)]
    print(list_diagnosed)

# Load model
graph = tf.Graph()
with graph.as_default():
    model = load_model(model_path, custom_objects={'f1': f1,
                                                   'sensitivity': sensitivity,
                                                   'specificity': specificity,
                                                   'falsePositiveRate': fpr})

    if use_fp_reduction:
        model_2 = load_model(model_path_2, custom_objects={'f1': f1,
                                                           'sensitivity': sensitivity,
                                                           'specificity': specificity,
                                                           'falsePositiveRate': fpr})
    else:
```

```
        model_2 = None

    if use_svm:
        svm = joblib.load(model_path_svm + '/SVM.pkl')
        model_2 = None
    else:
        svm = None

    if use_malignancy:
        model_3 = load_model(model_path_3, custom_objects={'f1': f1,
                                                           'sensitivity': sensitivity,
                                                           'specificity': specificity,
                                                           'falsePositiveRate': fpr})
    else:
        model_3 = None

    if use_spiculation:
        model_4 = load_model(model_path_4, custom_objects={'f1': f1,
                                                           'sensitivity': sensitivity,
                                                           'specificity': specificity,
                                                           'falsePositiveRate': fpr})

    else:
        model_4 = None

    if use_lobulation:
        model_5 = load_model(model_path_5, custom_objects={'f1': f1,
                                                           'sensitivity': sensitivity,
                                                           'specificity': specificity,
                                                           'falsePositiveRate': fpr})
    else:
        model_5 = None

if use_svm:
    results_folder = 'ResultsSVM{1}cv{2}/'.format(num_model_svm, version_svm, cv_svm)
else:
    results_folder = 'Results/'

if not os.path.isdir(saving_folder_slides + '/' + results_folder):
    os.makedirs(saving_folder_slides + '/' + results_folder)

# Iterate over all patients
for i, patient_id in enumerate(patient_ids):

    # Change string to int
    num_id = int(patient_id.split('-')[2])

    if get_diagnosed:
        flag = patient_id in list_diagnosed
    else:
        flag = start_p <= num_id <= end_p

    # Check if it is in range
    if flag:

        patient_path = '{0}strides{1} thr{2} {3} {4} {5} {6}'.format(patient_id, stride,
                                                                     str(threshold),
                                                                     str(threshold_2),
                                                                     str(threshold_3),
                                                                     str(threshold_4),
                                                                     str(threshold_5))

        if use_fp_reduction:
            saving_path_slides = saving_folder_slides_2 + '/' + results_folder + patient_path
        else:
            saving_path_slides = saving_folder_slides + '/' + results_folder + patient_path

        if use_malignancy:
            saving_path_slides_2 = saving_folder_slides_3 + '/' + results_folder + patient_path
        else:
            saving_path_slides_2 = None

        if use_lobulation:
            saving_path_slides_3 = saving_folder_slides_4 + '/' + results_folder + patient_path
        else:
            saving_path_slides_3 = None

        if use_spiculation:
            saving_path_slides_4 = saving_folder_slides_5 + '/' + results_folder + patient_path
        else:
            saving_path_slides_4 = None

        saving_path_fp = saving_folder_fp + '/' + results_folder + patient_path
        saving_path_tp = saving_folder_tp + '/' + results_folder + patient_path

        # Load patient
        patient = u.loadPatient(patient_dir, patient_id)

        saving_path_slides = saving_path_slides.replace('LIDC-IDRI-', '')
        saving_path_fp = saving_path_fp.replace('LIDC-IDRI-', '')
```

```
        start = time.time()
        results, results_2, results_3, results_4, results_5, true_pos_detections, false_pos_detections, true_labels = \
            detector.slide_window_3d(
                model, model_2, model_3,
                model_4, model_5, svm,
                patient, graph,
                saving_path_slides,
                saving_path_fp,
                saving_path_tp,
                use_fp_reduction=use_fp_reduction,
                use_svm=use_svm,
                use_malignancy=use_malignancy,
                use_spiculation=use_spiculation,
                use_lobulation=use_lobulation,
                data_dims=data_dims,
                filter_size=filter_size,
                stride=stride,
                detection_threshold=threshold,
                detection_threshold_2=threshold_2,
                save_slides=save_slides,
                save_fp=save_fp,
                save_tp=save_tp,
                graphics_show=graphics_show,
                augment_data=augment_data,
                num_radiologists=num_radiologists)
        end = time.time()

        parameters_file = saving_path_slides + '/parameters.txt'
        f = open(parameters_file, 'w')
        f.write('PARAMETERS\n')
        f.write('   strides: {}\n'.format(str(stride)))
        f.write('   number of radiologists agreements: {}\n'.format(str(num_radiologists)))
        f.write('   detection threshold: {}\n'.format(str(threshold)))
        f.write('   detection threshold 2: {}\n'.format(str(threshold_2)))
        f.write('   time elapsed: {}\n'.format(str(end - start)))
        f.close()
```

detector_class

```
from classes_lidc import Extractor, dataTransformer, PreProcessor
import numpy as np
import os
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib import rc
import scipy.misc


class Detector(object):

    def __init__(self):
        self.no_attribute = None

    @staticmethod
    def augment(data):
        # Create data transformer to augment data
        data_transformer = dataTransformer()
        augmented = np.empty((4,) + data.shape)
        augmented[0] = data

        # Rotate in X axis
        rot90_X = data_transformer.rot_tf(data, angle=90, axis=0)
        augmented[1] = rot90_X
        # Rotate in Y axis
        rot90_Y = data_transformer.rot_tf(data, angle=90, axis=1)
        augmented[2] = rot90_Y
        # Rotate in Z axis
        rot90_Z = data_transformer.rot_tf(data, angle=90, axis=2)
        augmented[3] = rot90_Z

        return augmented

    @staticmethod
    def is_false_positive(detection_cube, true_labels,
                          threshold=0.5):
        """
            Returns if the 3D detection coincides with any of the true
            labels in the scan
        """
        # Create the extractor
        e = Extractor()

        # Compute IoU for each possible ground truth label
        for true_label in true_labels:
            iou = e.IoU_Volume(true_label, detection_cube)
            print('IOU TP = {0}'.format(iou))
            if iou >= threshold:
                # If there is a big intersection then it is a true pos
                return False

        # If no intersection was detected with the ground truth labels
        # then it is a false positive
        return True

    @staticmethod
    def update_true_labels(detection_cube, true_labels, z_coords,
                           threshold=0.5):

        """
            Returns the updated true labels
        """
        # Create the extractor
        e = Extractor()

        # Compute IoU for each possible ground truth label
        for i, true_label in enumerate(true_labels):
            iou = e.IoU_Volume(true_label, detection_cube)
            if iou > threshold:
                # If there is a big intersection then it is a true pos, remove true label
                true_labels.pop(i)
                z_coords.pop(i)
                return true_labels

        return true_labels

    @staticmethod
    def get_z_coords_labels(true_labels):
        """
            Extracts all the Z coordinates from all nodules present
            in the scan, this is done to plot the contours of the nodule
            during the sliding window process
        """
        z_coords = []
        for label in true_labels:
            minZ = label[4]
            maxZ = label[5]
            z_Range = [minZ, maxZ]
            z_coords.append(z_Range)
```

```
        return z_coords

    @staticmethod
    def detect_nodules_in_slice(slice_coord, z_coords, true_labels):
        """
            Returns arrays of X and Y coords of nodules that are in current
            slice. This is used to plot contours of those nodules.
        """
        coordsXY = []
        for i, z_coord in enumerate(z_coords):
            minZ = z_coord[0]
            maxZ = z_coord[1]
            # If slice coord is between the Z range, nodule is there, therefore
            # extract X and Y coords
            if minZ <= slice_coord <= maxZ:
                minX = true_labels[i][0]
                maxX = true_labels[i][1]
                minY = true_labels[i][2]
                maxY = true_labels[i][3]
                coordXY = [minX, maxX, minY, maxY]
                coordsXY.append(coordXY)

        return coordsXY

    @staticmethod
    def is_new_detection(detection_cube, detected_nodules, threshold=0.3):
        """
            Returns a bool that specifies if the detected nodule is a new
            detection or it was already detected
        """
        e = Extractor()

        # Iterate over the coordinates of the detected nodules
        for detected in detected_nodules:
            # Get the iou
            iou = e.IoU_Volume(detected, detection_cube)

            # If IoU is greater than threshold
            if iou > threshold:
                # No new detection
                return False
        # New detection
        return True

    @staticmethod
    def is_new_tp(detection_cube, detected_nodules, threshold=0.3):
        """
            Returns a bool that specifies if the detected nodule is a new
            TP or it was already detected
        """
        e = Extractor()

        # Iterate over the coordinates of the detected nodules
        for detected in detected_nodules:
            # Get the iou
            iou = e.IoU_Volume(detected, detection_cube)
            # If IoU is greater than threshold
            if iou > threshold:
                # No new detection
                return False
        # New detection
        return True

    def slide_window_3d(self, model, model_2, model_3, model_4, model_5, svm, patient, graph,
                        saving_path_slides,
                        saving_path_fp,
                        saving_path_tp,
                        use_fp_reduction=True,
                        use_svm=True,
                        use_malignancy=True,
                        use_spiculation=True,
                        use_lobulation=True,
                        data_dims=None,
                        filter_size=(60, 60, 60),
                        stride=(10, 10, 10),
                        detection_threshold=0.5,
                        detection_threshold_2=0.5,
                        save_slides=False,
                        save_fp=False,
                        save_tp=False,
                        graphics_show=False,
                        augment_data=False,
                        num_radiologists=3):

        """

          :rtype: object
          """

        true_pos_detections = []
        false_pos_detections = []

        # Get resampled scan pixels
```

```
        p = PreProcessor()
        scan_pixels = patient.get_scan_pixels()
        scan = patient.get_patient_scan()
        scan_pixels, resize_factor = p.resampleData(scan_pixels, scan)

        # Get true labels
        true_labels = patient.get_true_labels(resize_factor=resize_factor, num_radiologists=num_radiologists)
        true_labels_temp = patient.get_true_labels(resize_factor=resize_factor, num_radiologists=num_radiologists)

        if type(true_labels) is list:
            num_nodules = len(true_labels)
        else:
            # There are no big nodules.
            if type(true_labels) is 0:
                num_nodules = 0

        # Get all z coordinates of all nodules in scan. Used for plotting
        z_coords = self.get_z_coords_labels(true_labels)
        z_coords_temp = self.get_z_coords_labels(true_labels)

        # Initialize number of true positives and false positives
        true_pos = 0
        false_pos = 0

        # Initialize a list to keep track of the detected nodules
        # this list contains the coordinates of the detected nodules
        detected_nodules = []

        # Get the number of steps to iterate in each dimension of the scan
        stepsX = (scan_pixels.shape[0] - filter_size[0])
        stepsY = (scan_pixels.shape[1] - filter_size[1])
        stepsZ = (scan_pixels.shape[2] - filter_size[2])
        stepsX = int(stepsX)
        stepsY = int(stepsY)
        stepsZ = int(stepsZ)

        # Initialize the matrix of results
        results = np.empty((stepsX, stepsY, stepsZ))
        results_2 = np.empty((stepsX, stepsY, stepsZ))
        results_3 = np.empty((stepsX, stepsY, stepsZ))
        results_4 = np.empty((stepsX, stepsY, stepsZ))
        results_5 = np.empty((stepsX, stepsY, stepsZ))

        # Iterate over the 3 dimensions
        print('Sliding the window over the patient scan...')

        # Initialize number of detections
        num_detections = 0

        # Change font size
        rc('font', size=4)

        # Check saving folders
        print(saving_path_slides)
        print(saving_path_fp)

        if save_slides:
            if not os.path.isdir(saving_path_slides):
                os.makedirs(saving_path_slides)

        if save_fp:
            if not os.path.isdir(saving_path_fp):
                os.makedirs(saving_path_fp)

        if save_tp:
            if not os.path.isdir(saving_path_tp):
                os.makedirs(saving_path_tp)

        # Iterate over the scan
        for i in range(0, stepsX, stride[0]):

            # Initialize max prediction
            max_detection_value = 0
            max_detection_value_2 = 0
            max_detection_loc = [0, 0]
            max_detection_loc_2 = [0, 0]

            # Plot the middle slice of the 3D window
            first_slice = i
            middle_slice = i + int((filter_size[0]) / 2)
            last_slice = i + filter_size[0]

            # Create all graphics on image of 3D scan
            if save_slides or graphics_show:
                fig, axarr = plt.subplots(3, 2)
                plt.subplots_adjust(top=0.92, bottom=0.08,
                                    left=0.10, right=0.95,
                                    hspace=0.35, wspace=0.35)
                fig.suptitle((' Num of detections: {}'.format(num_detections))
                             + (' Max prediction: {}'.format(max_detection_value))
                             + (' Max prediction 2: {}'.format(max_detection_value_2))
                             + (' Num FP: {}'.format(false_pos))
                             + (' Num TP: {}'.format(true_pos))
```

```
            + (' Num of nodules: {}'.format(num_nodules)))

        first_img = scan_pixels[first_slice, :, :]
        axarr[0][0].title.set_text('First Slice {}'.format(first_slice))
        axarr[0][0].set_xlabel('X')
        axarr[0][0].set_ylabel('Y')
        axarr[0][0].imshow(first_img)
        first_ax_img = axarr[0][1].imshow(first_img)

        middle_img = scan_pixels[middle_slice, :, :]
        axarr[1][0].title.set_text('Middle Slice {}'.format(middle_slice))
        axarr[1][0].set_xlabel('X')
        axarr[1][0].set_ylabel('Y')
        axarr[1][0].imshow(middle_img)
        middle_ax_img = axarr[1][1].imshow(middle_img)

        last_img = scan_pixels[last_slice, :, :]
        axarr[2][0].title.set_text('Last Slice {}'.format(last_slice))
        axarr[2][0].set_xlabel('X')
        axarr[2][0].set_ylabel('Y')
        axarr[2][0].imshow(last_img)
        last_ax_img = axarr[2][1].imshow(last_img)

        # Initialize rectangles for max predition
        rectangle_max_1 = patches.Rectangle((0, 0),
                                            filter_size[1],
                                            filter_size[2],
                                            linewidth=1,
                                            edgecolor='y',
                                            facecolor='none')
        rectangle_max_2 = patches.Rectangle((0, 0),
                                            filter_size[1],
                                            filter_size[2],
                                            linewidth=1,
                                            edgecolor='y',
                                            facecolor='none')
        rectangle_max_3 = patches.Rectangle((0, 0),
                                            filter_size[1],
                                            filter_size[2],
                                            linewidth=1,
                                            edgecolor='g',
                                            facecolor='none')

        axarr[0][0].add_patch(rectangle_max_1)
        axarr[1][0].add_patch(rectangle_max_2)
        axarr[2][0].add_patch(rectangle_max_3)

        if use_fp_reduction:
            # Initialize rectangles for max predition 2
            rectangle_max_1_2 = patches.Rectangle((0, 0),
                                                  filter_size[1],
                                                  filter_size[2],
                                                  linewidth=1,
                                                  edgecolor='g',
                                                  facecolor='none')
            rectangle_max_2_2 = patches.Rectangle((0, 0),
                                                  filter_size[1],
                                                  filter_size[2],
                                                  linewidth=1,
                                                  edgecolor='g',
                                                  facecolor='none')
            rectangle_max_3_2 = patches.Rectangle((0, 0),
                                                  filter_size[1],
                                                  filter_size[2],
                                                  linewidth=1,
                                                  edgecolor='g',
                                                  facecolor='none')
            axarr[0][0].add_patch(rectangle_max_1_2)
            axarr[1][0].add_patch(rectangle_max_2_2)
            axarr[2][0].add_patch(rectangle_max_3_2)

        # If a nodule is in either the first, middle or last slice
        # add rectangle
        first_coords = self.detect_nodules_in_slice(first_slice,
                                                    z_coords,
                                                    true_labels)
        middle_coords = self.detect_nodules_in_slice(middle_slice,
                                                     z_coords,
                                                     true_labels)
        last_coords = self.detect_nodules_in_slice(last_slice,
                                                   z_coords,
                                                   true_labels)

        # Add rectangles of nodule in first slice
        for coords in first_coords:
            minX = coords[0]
            minY = coords[2]
            width = coords[1] - minX
            height = coords[3] - minY
            rectangle_first = patches.Rectangle((minX, minY),
                                                width,
                                                height,
                                                linewidth=1,
```

```
                                                edgecolor='c',
                                                facecolor='none')
                    axarr[0][0].add_patch(rectangle_first)

                # Add rectangles of nodule in middle slice
                for coords in middle_coords:
                    minX = coords[0]
                    minY = coords[2]
                    width = coords[1] - minX
                    height = coords[3] - minY
                    rectangle_middle = patches.Rectangle((minX, minY),
                                                         width,
                                                         height,
                                                         linewidth=1,
                                                         edgecolor='c',
                                                         facecolor='none')
                    axarr[1][0].add_patch(rectangle_middle)

                # Add rectangles of nodule in last slice
                for coords in last_coords:
                    minX = coords[0]
                    minY = coords[2]
                    width = coords[1] - minX
                    height = coords[3] - minY
                    rectangle_last = patches.Rectangle((minX, minY),
                                                       width,
                                                       height,
                                                       linewidth=1,
                                                       edgecolor='c',
                                                       facecolor='none')
                    axarr[2][0].add_patch(rectangle_last)

            # Iterate over current slice
            for j in range(0, stepsY, stride[1]):
                for k in range(0, stepsZ, stride[2]):

                    # Extract current piece of scan to be analyzed by the CNN
                    extracted = scan_pixels[i:i + filter_size[0], j:j + filter_size[1], k:k + filter_size[2]]

                    # Ensure that the extracted piece of the scan corresponds
                    # to the input dimensions of the CNN
                    if data_dims != filter_size[0]:
                        resizeFactor = data_dims / filter_size
                        extracted = scipy.ndimage.interpolation.zoom(extracted,
                                                                     resizeFactor,
                                                                     mode='nearest')
                    # Augment data if wanted
                    if augment_data:
                        augmented = self.augment(extracted)
                        extracted_reshaped = augmented.reshape(augmented.shape + (1,))
                    else:
                        extracted_reshaped = extracted.reshape((1,) + extracted.shape + (1,))

                    # Predict  and initialize TensorFlow graph
                    with graph.as_default():
                        results[i, j, k] = model.predict(extracted_reshaped)

                        if use_fp_reduction:
                            if use_svm:
                                x = extracted_reshaped.reshape(32 * 32 * 32)
                                x = np.expand_dims(x, axis=0)
                                results_2[i, j, k] = svm.predict_proba(x)[0][1]
                            else:
                                results_2[i, j, k] = model_2.predict(extracted_reshaped)

                    # Save max value
                    if results[i, j, k] > max_detection_value:
                        # Save the max prediction
                        max_detection_value = results[i, j, k]
                        max_detection_loc[0] = i
                        max_detection_loc[1] = j

                        # Remove the rectangle with previous max detection
                        rectangle_max_1.remove()
                        rectangle_max_2.remove()
                        rectangle_max_3.remove()

                        # Add the rectangle in position of max prediction
                        rectangle_max_1 = patches.Rectangle((k, j),
                                                            filter_size[1],
                                                            filter_size[2],
                                                            linewidth=1,
                                                            edgecolor='y',
                                                            facecolor='none')
                        rectangle_max_2 = patches.Rectangle((k, j),
                                                            filter_size[1],
                                                            filter_size[2],
                                                            linewidth=1,
                                                            edgecolor='y',
                                                            facecolor='none')
                        rectangle_max_3 = patches.Rectangle((k, j),
                                                            filter_size[1],
                                                            filter_size[2],
```

```
                                        linewidth=1,
                                        edgecolor='y',
                                        facecolor='none')
            axarr[0][0].add_patch(rectangle_max_1)
            axarr[1][0].add_patch(rectangle_max_2)
            axarr[2][0].add_patch(rectangle_max_3)

        # Save max value
        if results_2[i, j, k] > max_detection_value_2 and use_fp_reduction:
            # Save the max prediction
            max_detection_value_2 = results_2[i, j, k]
            max_detection_loc_2[0] = i
            max_detection_loc_2[1] = j

            # Remove the rectangle with previous max detection
            rectangle_max_1_2.remove()
            rectangle_max_2_2.remove()
            rectangle_max_3_2.remove()

            # Add the rectangle in position of max prediction
            rectangle_max_1_2 = patches.Rectangle((k, j),
                                                  filter_size[1],
                                                  filter_size[2],
                                                  linewidth=1,
                                                  edgecolor='g',
                                                  facecolor='none')
            rectangle_max_2_2 = patches.Rectangle((k, j),
                                                  filter_size[1],
                                                  filter_size[2],
                                                  linewidth=1,
                                                  edgecolor='g',
                                                  facecolor='none')
            rectangle_max_3_2 = patches.Rectangle((k, j),
                                                  filter_size[1],
                                                  filter_size[2],
                                                  linewidth=1,
                                                  edgecolor='g',
                                                  facecolor='none')
            axarr[0][0].add_patch(rectangle_max_1_2)
            axarr[1][0].add_patch(rectangle_max_2_2)
            axarr[2][0].add_patch(rectangle_max_3_2)

        # Create 3 rectangles for correct detection
        rectangle1 = patches.Rectangle((k, j),
                                       filter_size[1],
                                       filter_size[2],
                                       linewidth=1,
                                       edgecolor='r',
                                       facecolor='none')
        rectangle2 = patches.Rectangle((k, j),
                                       filter_size[1],
                                       filter_size[2],
                                       linewidth=1,
                                       edgecolor='r',
                                       facecolor='none')
        rectangle3 = patches.Rectangle((k, j),
                                       filter_size[1],
                                       filter_size[2],
                                       linewidth=1,
                                       edgecolor='r',
                                       facecolor='none')

        # Check if prediction is considered nodule or not
        if results[i, j, k] > detection_threshold:

            # Get bounding cube of detection
            minX = k
            maxX = k + filter_size[2]
            minY = j
            maxY = j + filter_size[1]
            minZ = i
            maxZ = i + filter_size[0]

            detection_cube = np.array([minX, maxX,
                                       minY, maxY,
                                       minZ, maxZ])

            # Create rectangles of detection and sliding window
            rectangle_detected_1 = patches.Rectangle((k, j),
                                                     filter_size[1],
                                                     filter_size[2],
                                                     linewidth=1,
                                                     edgecolor='violet',
                                                     facecolor='none')
            rectangle_detected_2 = patches.Rectangle((k, j),
                                                     filter_size[1],
                                                     filter_size[2],
                                                     linewidth=1,
                                                     edgecolor='violet',
                                                     facecolor='none')
            rectangle_detected_3 = patches.Rectangle((k, j),
                                                     filter_size[1],
                                                     filter_size[2],
```

```
                                                linewidth=1,
                                                edgecolor='violet',
                                                facecolor='none')

                        # False positive reduction detector
                        if use_fp_reduction:

                            if results_2[i, j, k] > detection_threshold_2:

                                print(results_2[i, j, k])

                                # If it is a new nodule, add it to the number of detections
                                # but only if it is a new detection
                                if self.is_new_detection(detection_cube, detected_nodules, threshold=0.5):
                                    # Append new detection
                                    detected_nodules.append(detection_cube)

                                    with graph.as_default():
                                        if use_malignancy:
                                            results_3[i, j, k] = model_3.predict(extracted_reshaped)
                                        if use_spiculation:
                                            results_4[i, j, k] = model_4.predict(extracted_reshaped)
                                        if use_lobulation:
                                            results_5[i, j, k] = model_5.predict(extracted_reshaped)

                                    # Add new detection
                                    num_detections += 1

                                    # Add rectangles of detection
                                    axarr[0][0].add_patch(rectangle_detected_1)
                                    axarr[0][1].title.set_text('Detected - Prediction : {}'.format(results[i, j, k]))
                                    axarr[1][0].add_patch(rectangle_detected_2)
                                    axarr[1][1].title.set_text('Detected - Prediction : {}'.format(results[i, j, k]))
                                    axarr[2][0].add_patch(rectangle_detected_3)
                                    axarr[2][1].title.set_text('Detected - Prediction : {}'.format(results[i, j, k]))

                                    # Compute if the new detection is a false positive
                                    if self.is_false_positive(detection_cube, true_labels_temp, threshold=0.2):

                                        # Check if it is not a true positive that was already detected
                                        if self.is_new_tp(detection_cube, true_pos_detections, threshold=0.125):
                                            false_pos += 1
                                            false_pos_detections.append(detection_cube)

                                            if save_fp:
                                                print('Saving FP {0}'.format(false_pos))
                                                file_name = saving_path_fp + '/FP {0}'.format(false_pos)
                                                np.save(file_name, extracted_reshaped)

                                    # It is a true positive
                                    else:
                                        # Check if it was not detected before
                                        if self.is_new_detection(detection_cube, true_pos_detections):
                                            true_labels_temp = self.update_true_labels(detection_cube, true_labels_temp,
                                                                                       z_coords_temp, threshold=0.125)
                                            true_pos += 1
                                            true_pos_detections.append(detection_cube)

                                            if save_tp:
                                                print('Saving TP {0}'.format(true_pos))
                                                file_name = saving_path_tp + '/TP {0}'.format(true_pos)
                                                np.save(file_name, extracted_reshaped)

                        else:
                            # If it is a new nodule, add it to the number of detections
                            # but only if it is a new detection
                            if self.is_new_detection(detection_cube, detected_nodules, threshold=0.5):
                                # Append new detection
                                detected_nodules.append(detection_cube)

                                with graph.as_default():
                                    if use_malignancy:
                                        results_3[i, j, k] = model_3.predict(extracted_reshaped)
                                    if use_spiculation:
                                        results_4[i, j, k] = model_4.predict(extracted_reshaped)
                                    if use_lobulation:
                                        results_5[i, j, k] = model_5.predict(extracted_reshaped)

                                # Add new detection
                                num_detections += 1

                                # Add rectangles of detection
                                axarr[0][0].add_patch(rectangle_detected_1)
                                axarr[0][1].title.set_text('Detected - Prediction : {}'.format(results[i, j, k]))
                                axarr[1][0].add_patch(rectangle_detected_2)
                                axarr[1][1].title.set_text('Detected - Prediction : {}'.format(results[i, j, k]))
                                axarr[2][0].add_patch(rectangle_detected_3)
                                axarr[2][1].title.set_text('Detected - Prediction : {}'.format(results[i, j, k]))

                                # Compute if the new detection is a false positive
                                if self.is_false_positive(detection_cube, true_labels_temp, threshold=0.2):

                                    # Check if it is not a true positive that was already detected
```

```
                                if self.is_new_tp(detection_cube, true_pos_detections, threshold=0.125):
                                    false_pos += 1
                                    false_pos_detections.append(detection_cube)

                                    if save_fp:
                                        print('Saving FP {0}'.format(false_pos))
                                        file_name = saving_path_fp + '/FP {0}'.format(false_pos)
                                        np.save(file_name, extracted_reshaped)

                            # It is a true positive
                            else:
                                # Check if it was not detected before
                                if self.is_new_detection(detection_cube, true_pos_detections):
                                    true_labels_temp = self.update_true_labels(detection_cube, true_labels_temp,
                                                                               z_coords_temp, threshold=0.125)
                                    true_pos += 1
                                    true_pos_detections.append(detection_cube)

                                    if save_tp:
                                        print('Saving TP {0}'.format(true_pos))
                                        file_name = saving_path_tp + '/TP {0}'.format(true_pos)
                                        np.save(file_name, extracted_reshaped)

                        else:
                            # If not nodule only add rectangle of sliding window
                            axarr[0][1].title.set_text(
                                'Not Detected - Prediction : {0} Prediction {1}'.format(results[i, j, k],
                                                                                        results_2[i, j, k]))
                            axarr[1][1].title.set_text(
                                'Not Detected - Prediction : {0} Prediction {1}'.format(results[i, j, k],
                                                                                        results_2[i, j, k]))
                            axarr[2][1].title.set_text(
                                'Not Detected - Prediction : {0} Prediction {1}'.format(results[i, j, k],
                                                                                        results_2[i, j, k]))

                        axarr[0][0].add_patch(rectangle1)
                        axarr[1][0].add_patch(rectangle2)
                        axarr[2][0].add_patch(rectangle3)

                        # Update figures
                        fig.suptitle((' Num of detections: {}'.format(num_detections))
                                     + ('  -  Max prediction: {}'.format(round(max_detection_value, 3)))
                                     + ('  -  Max prediction 2: {}'.format(round(max_detection_value_2, 3)))
                                     + ('  -  FP: {}'.format(false_pos))
                                     + ('  -  FN: {}'.format(num_nodules - true_pos))
                                     + ('  -  TP: {}'.format(true_pos))
                                     + ('  -  Num of nodules: {}'.format(num_nodules))
                                     )
                        axarr[0][0].title.set_text(
                            'First Slice {} - Location of window: x={} y={}'.format(first_slice, j, k))
                        first_ax_img.set_data(extracted[0, :, :])
                        axarr[1][0].title.set_text(
                            'Middle Slice {} - Location of window: x={} y={}'.format(middle_slice, j, k))
                        middle_ax_img.set_data(extracted[int(extracted.shape[0] / 2), :, :])
                        axarr[2][0].title.set_text(
                            'Last Slice {} - Location of window: x={} y={}'.format(last_slice, j, k))
                        last_ax_img.set_data(extracted[extracted.shape[0] - 1, :, :])

                        # Show result
                        if graphics_show:
                            plt.draw()
                            plt.pause(0.00001)

                        # Remove sliding rectangles
                        rectangle1.remove()
                        rectangle2.remove()
                        rectangle3.remove()
                        print('\rLocation of window: {} {} {}'.format(i, j, k), end='\r', flush=True)

                print('\nSaving figure...')
                save_name = 'Result {}.png'.format(str(i))
                fig.savefig(saving_path_slides + '/' + save_name, dpi=300)
                print('Figure saved...')
                plt.close()

        results_file = saving_path_slides + '/results.txt'

        if not os.path.isdir(saving_path_slides):
            os.makedirs(results_file)

        # Get non zero indices
        index_malignancy = np.nonzero(results_3)
        index_spiculation = np.nonzero(results_4)
        index_lobulation = np.nonzero(results_5)

        f = open(results_file, 'w')
        f.write('METRICS\n')
        f.write('   FP: {}\n'.format(str(false_pos)))
        f.write('   FN: {}\n'.format(str(num_nodules - true_pos)))
        f.write('   TP: {}\n'.format(str(true_pos)))
        f.write('   Total nodules: {}\n'.format(str(num_nodules)))

        if len(results[index_malignancy]) is not 0:
```

```
            f.write('   Max prob malignancy and nodule with {0} detections: {1}\n'.format(false_pos + true_pos, str(
                np.max(results[index_malignancy] * results_3[index_malignancy]))))
        else:
            f.write('   Max prob malignancy and nodule with {0} detections: {1}\n'.format(false_pos + true_pos, 0))

        if len(results[index_spiculation]) is not 0:
            f.write('   Max prob spiculation and nodule with {0} detections:: {1}\n'.format(false_pos + true_pos, str(
                np.max(results[index_spiculation] * results_4[index_spiculation]))))
        else:
            f.write('   Max prob spiculation and nodule with {0} detections:: {1}\n'.format(false_pos + true_pos, 0))

        if len(results[index_lobulation]) is not 0:
            f.write('   Max prob lobulation and nodule with {0} detections: {1}\n'.format(false_pos + true_pos, str(
                np.max(results[index_lobulation] * results_5[index_lobulation]))))
        else:
            f.write('   Max prob lobulation and nodule with {0} detections: {1}\n'.format(false_pos + true_pos, 0))
            f.close()

        return results, results_2, results_3, results_4, results_5, true_pos_detections, false_pos_detections, true_labels
```

Bibliography

[1] Freddie Bray, Jacques Ferlay, Isabelle Soerjomataram, et al. Global cancer statistics 2018: GLOBOCAN estimates of incidence and mortality worldwide for 36 cancers in 185 countries. *CA: A Cancer Journal for Clinicians*, sep 2018.

[2] INEGI. Estadísticas a propósito del día mundial contra el cáncer. Datos Nacionales. *Inegi*, pages 1–14, 2017.

[3] Christopher P. Wild, Bernard W. Stewart, and Chris Wild. World Cancer Report 2014. page 630, 2014.

[4] Richard Wender, Elizabeth Fontham, Ermilo Barrera, et al. American Cancer Society lung cancer screening guidelines. *CA: A Cancer Journal for Clinicians*, 63(2):106–117, 2013.

[5] Gerard A. Silvestri, Anne V. Gonzalez, Michael A. Jantz, et al. Methods for staging non-small cell lung cancer: Diagnosis and management of lung cancer, 3rd ed: American college of chest physicians evidence-based clinical practice guidelines. *Chest*, 143(5 SUPPL):211–250, 2013.

[6] V Moyer. Screening for Lung Cancer: U.S. Preventive Services Task Force Recommendation Statement. *Annals of internal medicine*, 160(5):7, 2014.

[7] Paul Pinsky. Assessing the benefits and harms of low-dose computed tomography screening for lung cancer. *HHS Public Access*, (5):5, 6, 2015.

[8] Paul Pinsky, Naomi S Fineberg, and David S Gierada. Evaluation of Reader Variability in the Interpretation of Follow- up CT Scans at Lung Cancer Purpose : Methods : Results : Conclusion :. *Radiology*, 259(1):263–270, 2011.

[9] Elizabeth A Krupinski, Kevin S Berbaum, Kevin M Schartz, et al. The Impact of Fatigue on Satisfaction of Search in Chest Radiography. *Academic radiology*, 24(9):1058–1063, 2017.

[10] Jennifer Elston Lafata, Janine Simpkins, Lois Lamerato, et al. The economic impact of false-positive cancer screens. *Cancer Epidemiology Biomarkers and Prevention*, 13(12):2126–2132, 2004.

[11] Peter Goldstraw, Kari Chansky, John Crowley, et al. The IASLC lung cancer staging project: Proposals for revision of the TNM stage groupings in the forthcoming (eighth) edition of the TNM Classification for lung cancer. *Journal of Thoracic Oncology*, 11(1):39–51, 2016.

[12] Franco Cavalli. *Cáncer - El gran desafío*. La Habana, first edit edition, 2012.

[13] World Health Organization. Cancer, 2019.

[14] Susan Elmore. Apoptosis: a review of programmed cell death. *Toxicologic pathology*, 35(4):495–516, 2007.

[15] American College of Radiology. Lung CT screening reporting & data system. Technical report, American College of Radiology, 2019.

[16] Chunhua Xu, Keke Hao, Yong Song, Like Yu, Zhibo Hou, and Ping Zhan. Early diagnosis of solitary pulmonary nodules. *Journal of Thoracic Disease*, 5(6):830–840, 2013.

[17] William E. Brant and Clyde A. Helms. *Fundamentals of diagnostic radiology*. Lippincott, Williams & Wilkins, Philadelphia, 4th edition, 2012.

[18] Kyung Soo Lee, John R Mayo, Atul C Mehta, et al. Incidental Pulmonary Nodules Detected on CT Images: Fleischner 2017. *Radiology*, 000(0):1–16, 2017.

[19] G. L. Colice. Chest CT for known or suspected lung cancer. *Chest*, 106(5):1538–1550, 1994.

[20] NLST. Patient and Physician Guide : National Lung Screening Trial (NLST) Study Findings : Low-dose CT versus Chest X-ray screening " Take home " messages. page 800.

[21] Stephan I. Schabel. Historical notes on computerized axial tomography. *Journal of Computed Tomography*, 1(1):75, sep 2007.

[22] L. W. Goldman. Principles of CT and CT Technology. *Journal of Nuclear Medicine Technology*, 35(3):115–128, 2008.

[23] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep Learning in Medical Image Analysis. *Annual review of biomedical engineering*, 19:221–248, 2017.

[24] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[25] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint*, 2015.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778. IEEE, jun 2016.

[27] Klaus Greff, Rupesh K. Srivastava, Jan Koutnik, et al. LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 28(10):2222–2232, 2017.

[28] Diego Ardila, Atilla P Kiraly, Sujeeth Bharadwaj, et al. End-to-end lung cancer screening with three-dimensional deep learning on low-dose chest computed tomography. *Nature Medicine*, 2019.

[29] Yun Liu, Krishna Gadepalli, Mohammad Norouzi, et al. Detecting Cancer Metastases on Gigapixel Pathology Images. pages 1–11, 2017.

[30] Daniel S. Kermany, Michael Goldbaum, Wenjia Cai, et al. Identifying Medical Diagnoses and Treatable Diseases by Image-Based Deep Learning. *Cell*, 172(5):1122–1131.e9, 2018.

[31] Stuart Russell and Peter Norvig. *Artificial Intelligence A Modern Approach Third Edition*. Pearson, New Jersey, 3rd edition, 2010.

[32] A. L. Samuel. Some studies in machine learning using the game of checkers. II-Recent progress. *Annual Review in Automatic Programming*, 6(PART 1):1–36, 1969.

[33] Stephen Lucci and Danny Kopec. *Artificial Intelligence in the 21st century*. Stylus Publishing, LLC, second edition, 2015.

[34] Christopher Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag New York, first edition, 2006.

[35] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer-Verlag New York, New York, second edition, 2009.

[36] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 1943.

[37] Hebb, D. O. Organization of behavior. New York: Wiley, 1949, pp. 335, $4.00. *Journal of Clinical Psychology*, 6(3):307–307, jul 1950.

[38] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, nov 1958.

[39] Marvin Minsky and Seymour Papert. Perceptron: an introduction to computational geometry. *The MIT Press*, 19(88):292, 1969.

[40] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[41] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*, 2:359–366, 1989.

[42] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning Long-Term Dependencies with Gradient Descent is Difficult. *IEEE Transactions on Neural Networks*, 1994.

[43] Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, aug 2002.

[44] Yoshua Bengio and Yann Lecun. Scaling Learning Algorithms towards AI To appear in " Large-Scale Kernel Machines ",. *New York*, (1):1–41, 2007.

[45] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors. pages 1–8. Association for Computing Machinery (ACM), jun 2009.

[46] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research*, 9:249–256, 2010.

[47] Vinod Nair and Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. *Proceedings of the 27th International Conference on Machine Learning*, (3):807–814, 2010.

[48] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. *Journal of Machine Learning Research*, 15:315–323, 2011.

[49] Vincent Y. Ng, Thomas J. Scharschmidt, Joel L. Mayerson, et al. Incidence and survival in sarcoma in the United States: A focus on musculoskeletal lesions. *Anticancer Research*, 33(6):2597–2604, 2013.

[50] SRJ and B. S. Everitt. The Cambridge Dictionary of Statistics. *Journal of the American Statistical Association*, 94(446):657, may 2006.

[51] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint*, 2012.

[52] John B. Bell, A. N. Tikhonov, and V. Y. Arsenin. Solutions of Ill-Posed Problems. *Mathematics of Computation*, 32(144):1320, apr 2006.

[53] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv*, arXiv prep, sep 2016.

[54] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. pages 1–15, 2014.

[55] Ilya Sutskever, James Martens, George E Dahl, and Geoffrey E Hinton. On the importance of initialization and momentum in deep learning. *Jmlr W&Cp*, 28(2010):1139–1147, 2013.

[56] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. *Advances in Neural Information Processing Systems (NIPS)*, pages 2546–2554, 2011.

[57] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13:281–305, 2012.

[58] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, apr 1980.

[59] Jürgen Schmidhuber. Deep Learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[60] L. D. Jackel, R. E. Howard, B. Boser, J. S. Denker, D. Henderson, Y. LeCun, and W. Hubbard. Backpropagation Applied to Handwritten Zip Code Recognition, 1989.

[61] Eric W. Weisstein. Convolution.

[62] Aaron Goodfellow, Ian, Bengio, Yoshua, Courville. Deep Learning. *MIT Press*, 2016.

[63] I Sobel and G Feldman. A 3x3 isotropic gradient operator for image processing. *in Hart, P. E. & Duda R. O. Pattern Classification and Scene Analysis*, pages 271–272, 1973.

[64] J. Wu. Introduction to convolutional neural networks. *National Key Lab for Novel Software Technology*, pages 1–31, 2017.

[65] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[66] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, et al. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[67] Jun Zhang, Mingxia Liu, and Dinggang Shen. Detecting Anatomical Landmarks from Limited Medical Imaging Data Using Two-Stage Task-Oriented Deep Neural Networks. *IEEE Transactions on Image Processing*, 26(10):4753–4764, 2017.

[68] Daniel Hammack. Forecasting Lung Cancer Diagnoses with Deep Learning. pages 1–6, 2017.

[69] Cheng Dai, Bo Xiao, Yun Chen, et al. Automated Detection of Lung Nodules in CT Images with 3D Convolutional Neural Networks. *Proceedings of 2018 6th IEEE International Conference on Network Infrastructure and Digital Content, IC-NIDC 2018*, pages 55–59, 2018.

[70] Shiwen Shen, Simon X. Han, Denise R. Aberle, et al. An Interpretable Deep Hierarchical Semantic Convolutional Neural Network for Lung Nodule Malignancy Classification. 2018.

[71] Hiram Madero Orozco, Osiris O. Vergara Villegas, et al. Automated system for lung nodules classification based on wavelet feature descriptor and support vector machine. *BioMedical Engineering Online*, 14(1):1–20, 2014.

[72] A.A.A. Setio, A. Traverso, T. de Bel, et al. Validation, comparison, and combination of algorithms for automatic detection of pulmonary nodules in computed tomography images: The LUNA16 challenge, 2017.

[73] Kingsley Kuan, Mathieu Ravaut, Gaurav Manek, et al. Deep Learning for Lung Cancer Detection: Tackling the Kaggle Data Science Bowl 2017 Challenge. 2017.

[74] Wenqing Sun, Bin Zheng, and Wei Qian. Automatic feature learning using multichannel ROI based on deep structured algorithms for computerized lung cancer diagnosis. *Computers in Biology and Medicine*, 89(January):530–539, 2017.

[75] M Gomathi and P Thangaraj. Lung Nodule Detection using a Neural Classifier. *International Journal of Engineering and Technology*, 2(3):291–295, 2010.

[76] Darren Baker, Jen Kilpatrick, and Ali Chaudhry. Predicting Lung Cancer Incidence from CT Imagery. page 2017, 2017.

[77] Jinsa Kuruvilla and K. Gunavathi. Lung cancer classification using neural networks for CT images. *Computer Methods and Programs in Biomedicine*, 113(1):202–209, 2014.

[78] Devinder Kumar, Alexander Wong, and David A. Clausi. Lung Nodule Classification Using Deep Features in CT Images. *2015 12th Conference on Computer and Robot Vision*, (January 2016):133–138, 2015.

[79] Qi Dou, Hao Chen, Lequan Yu, et al. Multilevel Contextual 3-D CNNs for False Positive Reduction in Pulmonary Nodule Detection. *IEEE transactions on bio-medical engineering*, 64(7):1558–1567, 2017.

[80] W Li, P Cao, D Zhao, and J Wang. Pulmonary Nodule Classification with Deep Convolutional Neural Networks on Computed Tomography Images. *Computational and Mathematical Methods in Medicine*, 2016, 2016.

[81] Ross Gruetzemacher, Ashish Gupta, and David Paradice. 3D deep learning for detecting pulmonary nodules in CT scans. *Journal of the American Medical Informatics Association*, 25(10):1301–1310, 2018.

[82] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. 2016.

[83] Jia Ding, Aoxue Li, Zhiqiang Hu, and Liwei Wang. Accurate pulmonary nodule detection in computed tomography images using deep convolutional neural networks. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10435 LNCS, pages 559–567. Springer Verlag, 2017.

[84] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, may 2015.

[85] Samuel G Armato, Geoffrey Mclennan, Michael F Mcnitt-gray, et al. The Lung Image Database Consortium LIDC ... and Image Database Resource Initiative IDRI ... : A Completed Reference Database of Lung Nodules on CT Scans. (February):915–931, 2011.

[86] UTHSCSA. Mango.

[87] Abadi Martin, Agarwal Ashish, Barham Paul, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015.

[88] François Chollet. Keras, 2015.

[89] Darcy Mason. Pydicom: An Open Source DICOM Library, 2008.

[90] Travies E Oliphant. *A guide to NumPy*. Trelgol Publishing USA, 2006.

[91] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001.

[92] F Pedregosa, G Varoquaux, A Gramfort, V Michel, et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[93] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51–56, 2010.

[94] J D Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[95] J Bergstra, Daniel L K Yamins, and D D Cox. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures. *ICML*, pages 115–123, 2013.

[96] Dongbao Yang, Nannan Sun, Zhineng Chen, et al. Automated pulmonary nodule detection in CT images using deep convolutional neural networks. *Pattern Recognition*, 85:109–119, 2018.

[97] Joshua Broder. Imaging of Nontraumatic Abdominal Conditions. In *Diagnostic Imaging for the Emergency Physician*, pages 445–577. Elsevier Inc., 2011.

[98] Sanjana Patrick, NPraveen Birur, Keerthi Gurushanth, et al. Comparison of gray values of cone-beam computed tomography with hounsfield units of multislice computed tomography: An in vitro study. *Indian Journal of Dental Research*, 28(1):66, apr 2017.

[99] Massimo Cressoni, Elisabetta Gallazzi, Chiara Chiurazzi, et al. Limits of normality of quantitative thoracic CT analysis. *Critical Care*, 17(3):R93, 2013.

[100] W Jenkins, B Mather, and D Munson. Nearest neighbor and generalized inverse distance interpolation for Fourier domain image reconstruction. In *ICASSP'85. IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 10, pages 1069–1072. IEEE, 1985.

[101] Ian J Goodfellow, Jean Pouget-abadie, Mehdi Mirza, et al. Generative-Adversarial-Nets. *Nips*, pages 1–9, 2014.

[102] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation learning with Deep Convolutional GANs. *International Conference on Learning Representations*, pages 1–16, 2016.

[103] Erik Linder-Norén, Mirantha, et al. jonathand94/Keras-GAN: First Release Keras-GAN, 2019.

[104] James Bergstra, Dan Yamins, and DD Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. *PROCEEDINGS OF THE 12th PYTHON IN SCIENCE CONFERENCE*, (Scipy):13–20, 2013.

[105] Ignacio Enrique Sanchez, Antwerp Belgium, and Marcel Brun. Optimal threshold estimation for binary classifiers using game theory [version 3; referees: 3 approved]. *ISCB Comm JISCB Comm J*, 5(5):1–11, 2016.

[106] Zachary C. Lipton. The mythos of model interpretability. *Communications of the ACM*, 2018.