University "Politehnica" of Bucharest
Faculty of Electronics, Telecommunications and Information Technology

Automatic lung lesion segmentation using deep learning techniques

Diploma thesis

Submitted in partial fulfillment of the requirements
for the degree of *Engineer*
in the domain of *Computer Science and Information Technology*
study program *Information Engineering*

Thesis Advisor(s)
As. Univ. Drd. Ana-Antonia Neacşu,
Prof. Univ. Dr. Corneliu Burileanu

Student
Vlad-Mihai Vasilescu

April 2022

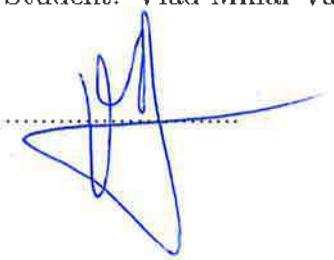
Statement of Academic Honesty

I hereby declare that the thesis *Lung lesion segmentation using deep learning techniques*, submitted to the Faculty of Electronics, Telecommunications and Information Technologies, University POLITEHNICA of Bucharest, in partial fulfillment of the requirements for the degree of *Engineer* in the domain Electronics and Telecommunications, study program *Computers and Information Technology* is written by myself and was never before submitted to any faculty or higher learning institution in Romania or any other country.

I declare that all information sources I used, including the ones I found on the Internet, are properly cited in the thesis as bibliographical references. text fragments cited "as is" or translated from other languages are written between quotes and are referenced to the source. Reformulation using different words of a certain text is also properly referenced. I understand that plagiarism constitutes an offence punishable by law.

I declare that all the results I present as coming from simulations or measurements I performed, together with the procedures used to obtain them, are real and indeed come from respective simulations or measurements. I understand that data faking is an offence punishable according to the University regulations.

Bucharest, June 2021.
Student: Vlad-Mihai Vasilescu



Universitatea "Politehnica" din București
Facultatea de Electronică, Telecomunicații și Tehnologia Informației
Program de studiu **INF**

Anexa 1

TEMA PROIECTULUI DE DIPLOMĂ

a studentului **VASILESCU M. Vlad-Mihai , 442A**

1. Titlul temei: Segmentarea Automată a leziunilor din plămâni folosind tehnici de învățare profundă

2. Descrierea temei și a contribuției personale a studentului (în afara părții de documentare):
Teza urmărește implementarea unui sistem automat de segmentare a leziunilor din tomografii computerizate ale plămânilor folosind algoritmi de învățare profundă (Deep Learning). Importanța unui diagnostic cât mai rapid pentru diferite afecțiuni ale plămânilor, incluzând implicările de sănătate aduse de noul coronavirus COVID-19, motivează elaborarea unei alternative automate de segmentare a zonelor afectate ale plămânilor din imagini biomedicale, precum tomografiile computerizate, eliminând astfel nevoia adnotării manuale de către specialiști. Studentul va implementa un sistem automat de segmentare bazat pe algoritmi de învățare profundă. Sistemul va fi antrenat folosind baze de date accesibile online ce conțin tomografii computerizate de la mai mulți pacienți, împreună cu adnotările leziunilor aferente acestora. Dată fiind importanța caracteristicii de generalizare pentru un astfel de sistem aplicat în domeniul medical, antrenarea sistemului va fi realizată pe un număr redus de date, relativ la setul de date pe care performanța acestuia va fi măsurată. Studentul va implementa diferite metode de obținere a măștilor de segmentare pornind de la secțiuni transversale extrase din volumul inițial reprezentat de tomografia computerizată. Capacitatea de generalizare a sistemului va fi testată prin validarea sistemului pe un set de date obținute diferit, cu un număr mai mare de pacienți.
Pentru realizarea proiectului se va utiliza limbajul de programare Python, precum si librării aferente algoritmilor de învățare profundă, precum Tensorflow.

3. Discipline necesare pt. proiect:
PI, RFIA, AI

4. Data înregistrării temei: 2020-11-24 15:26:46

Conducător(i) lucrare,
As. drd. Ing. Ana-Antonia NEACȘU

Prof. Corneliu Burileanu

Student,
VASILESCU M. Vlad-Mihai

Director departament,
Ș.L. dr. ing Bogdan FLOREA

Decan,
Prof. dr. ing. Mihnea UDREA

Cod Validare: **37d2bbf450**

Table of Contents

List of figures

List of tables

List of abbreviations

CT = Computed Tomography
ANN = Artificial Neural Network
CNN = Convolutional Neural Network
GAN = Generative Adversarial Network
RFB = Receptive Field Block
DAC = Dense Atrous Convolution
RMP = Residual Multi-kernel Pooling

Introduction

Thesis Motivation

Medical Imaging refers to a series of techniques for obtaining interior body images, revealing internal anatomical structures such as bones, organs, or different tissues, making it possible to detect some abnormalities and to assist diagnosis and treatment of patients. Different types of medical imaging techniques have been extensively used, the most common being radiography-based imaging, which uses ionizing/non-ionizing radiation projected towards the object whose internal representation is of interest. The emitted waves pass through the object of interest and are captured on the opposite side by a detector, making assumptions about the type of internal structures the radiation has passed through based on the absorbed energy. Computer tomography (CT) scanning relies on this process, by rotating the source that emits radiation around the subject, moving along a specific axis and producing multiple 2D images that are further combined to result a 3D volume.

CT scans have been extensively used during the last decade for preventive medicine and disease screening, with an approximate number of 75 million[1] CT scans performed all over the world each year, and a predicted increase up to 84 million by 2022. This technique has been introduced to various medical uses, such as head CT scanning – for detecting tumors, bone trauma, haemorrhage, cardiac CT scans – diagnosing coronary artery disease, or lung CT scans – detecting malignant (cancerous) tumors in early stages, or changes in the lung parenchyma. Different methods for extracting relevant information from lung CT scans have been developed during previous years, helping radiologists discover potential abnormalities and/or unusual changes that may occur in the current examination context.

Recent advances in *Deep Learning* have motivated integrating some of the proposed automatic techniques for image processing and analysis into many biomedical imaging tasks. Deep neural networks have been shown to exhibit high representation capacities, which has made them one of the most popular tool for solving any task involving complex multi-dimensional input data, such as medical images. Various architectures of convolutional neural networks have been proposed through the last years for applications involving medical images, such as disease prediction and classification or detection and segmentation of different anatomical objects of interests. The latter two have motivated building much more complex structures that would be able to capture high-level representative features, along with a good generalization for a large number of patients, which is of great interest for this area of research.

[1] https://idataresearch.com/over-75-million-ct-scans-are-performed-each-year-and-growing-despite-radiation-concerns/

Objectives

This thesis aims to implement an automatic lung lesion segmentation framework using deep learning techniques, i.e. different neural network architectures. The final architectures will be able to produce relevant lesion segmentation masks for chest CT scans of different patients, obtained from online available datasets. Thus, the main objectives of this work are:

- Obtaining online available datasets containing CT scans from different patients, along with corresponding annotated lesion masks, and planning the pre-processing workflow which will be applied to each extracted 2D slice.
- Designing neural network architectures which will receive as input 2D slices of CT scans and output the corresponding lesion segmentation mask.
- Building the workflow for training and testing the designed architectures with the obtained datasets.
- Assessing the performance of the trained model and its generalization capabilities by testing it on a different dataset.

The thesis is organized in 4 chapters, as follows:

Chapter 1 will present basic theoretical aspects of Artificial Neural Networks (ANNs) along with some existing methods for biomedical image segmentation tasks. *Chapter 2* will present the datasets used for experimenting different neural network architectures, together with a brief introduction to Computed Tomography (CT) scans and the preprocessing steps that have been applied to all data samples. In *Chapter 3* the proposed architectures will be detailed, as well as performance results and other remarks. Finally, *Chapter 4* is dedicated to final observations and future developments.

Chapter 1
Neural Networks – Theoretical Aspects

1.1 Introductory Concepts

Artificial Neural Networks (ANNs) represent a class of computational models usually designed to represent the mapping function of different input variables to corresponding outputs. They are inspired by biological neural networks which consist of different populations of neurons interconnected through links called synapses. The information which travels between biological neurons is encoded through sequences of *spikes*, the emission of a spike happening when the producing neuron receives sufficient input current such that its membrane potential exceeds a specific threshold value. This threshold is the deciding factor for whether a neuron contributes in the current process by forwarding the information to the next ones in the current population.

A *Dense Neural Network* consists of multiple layers of computational units called *neurons*, connected with previous and next layers by different matrices of *weights*. Each Dense Neural Network contains an input layer, an output layer, and at least one hidden layer. The inputs of each layer (except the input layer, which acts as a buffer containing the input data) consist of weighted sums between previous layer outputs and the weights associated with the connection between these two layers. Each neuron input is the *scalar product* between previous layer outputs and the weights between those outputs and the current neuron (the weights are arranged either as a row or as a column in the corresponding weight matrix, depending on the convention).

Activation functions are functions attached to each layer of the neural network, used to take the input of each neuron and output a value which encodes the *activity* of that respective neuron. When using a linear activation function the output of a dense network could be represented by a linear combination of the input variables, therefore the neural network could be represented by only two layers, input and output, with the weight matrix containing the coefficients of that linear combination, eliminating the need for any extra hidden layers. Since solutions to most problems cannot be sufficiently approximated by using just a linear combination of the input variables, using *non-linear* activation functions has become a popular practice when building ANNs.

Figure 1.1 shows an example of a dense neural network containing 4 layers: input layer with 5 neurons, 2 hidden layers each with 9 neurons, and an output layer with 3 neurons. Considering this example, let $\mathbf{X} \in \mathbb{R}^5$ be a input vector and $\mathbf{W}_1 \in \mathbb{R}^{9\times 5}$ be the weight matrix connecting the input layer to the first hidden layer. Let $\mathbf{Z}_1, \mathbf{Y}_1 \in \mathbb{R}^9$ be the input, respectively the output vector of the first hidden layer. The input vector is calculated as $\mathbf{Z}_1 = \mathbf{W}_1\mathbf{X}$, with a *bias* term eventually added. Let $g_1 : \mathbb{R} \to \mathbb{R}$ be the activation function assigned to each neuron in the first hidden layer. The output of any hidden unit from the first layer is therefore

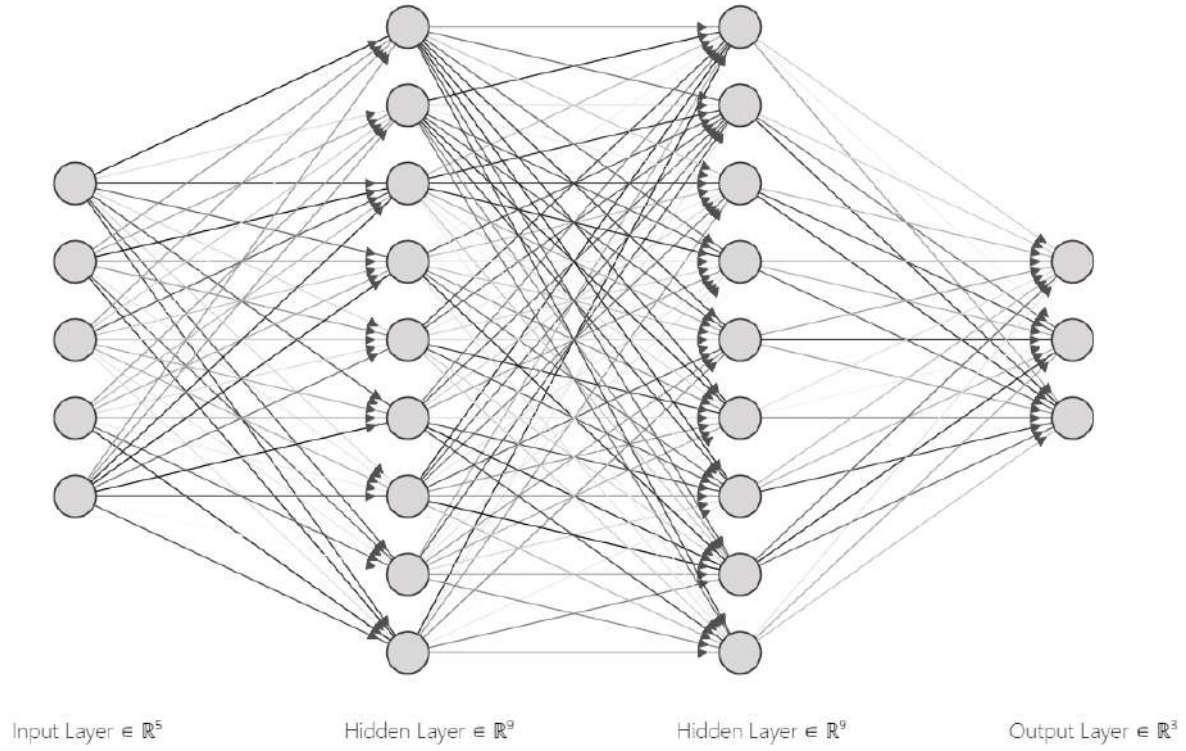


Figure 1.1: Schematic example of a dense neural network

$y_{1i} = g_1(z_{1i})$, for $i = 1...9$, where z_{1i} and y_{1i} are the i^{th} components of vector $\mathbf{Z}_1$, respectively $\mathbf{Y}_1$. These outputs act as inputs for the next hidden layer, therefore similar calculations are needed in order to obtain the final output of our network, given the rest of parameters and activation functions.

The most important characteristic of biological neural networks is their ability to accumulate information to build new representations regarding different scenarios, while building associations between new information and previously memorized patterns. This process is called **learning**. In ANNs learning is the process in which the network changes its internal parameters to achieve certain tasks represented by specific objective functions, i.e. minimizing the error between the representations produced by the neural network and some ideal target representations. Different learning paradigms have been elaborated since the implementation of ANNs for solving certain tasks, the 2 most popular being **supervised learning** and **unsupervised learning**. Supervised learning consists in training the ANN to produce a desired output for each given input. This strategy of learning is suited for *classification* problems, i.e. predicting the class of each input vector given some possible classes. Another task is *regression*, used to predict continuous variables for each input vector, which can be formulated as a function approximation task.

Unsupervised learning deals with input data for which there is no target output. The neural network is forced to learn representations based only thes vectors, such that the outputs (which are not forced to equal any ideal output) will include abstract information used for grouping the input data, thus forming a number of *clusters*. The mechanism for learning such representations includes applying external constraints on the output vectors such that the similarity is preserved after the transformation defined by the network.

In order to approximate the mapping function between the input and output variables, the weights need to be set accordingly. The "correctness" of an approximation has to be quantified through a variable measure which is called the **loss function** that estimates a *distance* between the output variables given by the network and the target output values, in the context of

supervised learning. Let $\mathbf{Y}^{(i)}$ be the desired output vector when the input vector is $\mathbf{X}^{(i)}$. After passing $\mathbf{X}^{(i)}$ to the network, the observed output is $\hat{\mathbf{Y}}^{(i)}$. The loss associated with this example will be denoted $\mathcal{L}^{(i)} = \mathcal{L}(\hat{\mathbf{Y}}^{(i)}, \mathbf{Y}^{(i)})$, where $\mathcal{L}$ denotes the loss function chosen to represent the distance between the target output and the output produced by the neural network. In this case, the loss function associated with the current configuration of the neural network is the mean loss over all examples presented to the neural network, $\mathcal{L}_{total} = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}^{(i)}$, where N is the number of data samples, i.e. the number of tuples $(\mathbf{X}^{(i)}, \mathbf{Y}^{(i)})$. Following the previous example, $\hat{\mathbf{Y}}^{(i)}$ can be calculated by **forward propagation** of the input vector $\mathbf{X}^{(i)}$ through the neural network as follows:

$$\hat{Y}^{(i)} = g_o(W_o \ g_L(W_L \ g_{L-1}(...(W_2 \ g_1(W_1 X^{(i)})))) \tag{1.1}$$

where L is the number of hidden layers, g_o is the activation function for the output layer, $g_k \ \forall k \in \{1 \dots L\}$, are the activation functions attached to each hidden layer. $(W_k)_{1\leq k\leq L}$ is the weight matrix associated with the connections between hidden layer $k-1$ and hidden layer k, and W_o is the weight matrix connecting the final hidden layer to the output layer. In the above notations, it is considered that the activation functions are applied element-wise for each resulting vector. The loss function could be therefore expanded as a function of the neural network parameters $(W_k)_{1\leq k\leq L}$, given the expression of each activation function, by introducing the equation 1.1 in the formula for $\mathcal{L}_{total}$.

Some popular choices for the loss function are presented as follows:

- **Mean Squared Error (MSE)**
  Let $Y^{(i)} \in \mathcal{R}^{N_o}$ be the target output when input $X^{(i)}$ is applied to the network, and $\hat{Y}^{(i)} \in \mathcal{R}^{N_o}$ be the actual obtained output. Then, the MSE is defined by the following equation:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}\sum_{p=1}^{N_o}(y_p^{(i)} - \hat{y}_p^{(i)})^2, \tag{1.2}$$

  where $y_p^{(i)}$ and $\hat{y}_p^{(i)}$ are the p^{th} coordinate values of vector $Y^{(i)}$, and vector $\hat{Y}^{(i)}$ respectively, and N is the dimensionality of the dataset. The total error is thus the mean error over all examples from the dataset, which are in fact the *Euclidean distances* between the predicted vector and the target vector.

- **Mean Absolute Error (MAE)** is defined as follows:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}\sum_{p=1}^{N_o}|y_p^{(i)} - \hat{y}_p^{(i)}| \tag{1.3}$$

  The main difference between MSE and MAE is that MSE is much more sensitive to errors greater than 1, while MAE adds more penalty to errors lower than 1. Taking the partial derivative of MSE with respect to any weight of the network will result in a formula which contains the actual difference between predicted and target values as a factor of the partial derivative, while for MAE the partial derivative does not contain any information about the magnitude of the error itself.

- **Binary Cross-Entropy (BCE)** is used in binary classification tasks (in which only 2 classes are considered) and is defined as follows:

$$BCE = -\frac{1}{N}\sum_{i=1}^{N} y^{(i)} \ log_2(\hat{y}^{(i)}) + (1 - y^{(i)}) \ log_2(1 - \hat{y}^{(i)}) \tag{1.4}$$

In this formulation, it is considered that the output layer of the network contains only one neuron, for which the output is denoted as $\hat{y}^{(i)} \in [0,1]$, and $y^{(i)} \in \{0,1\}$ denotes the class of the i^{th} example from the dataset. The above formulation implies that in order to minimize the BCE loss, the output of the network for examples from class $y = 1$ should be close to 1, while for examples from class $y = 0$ it should be close to 0. If we constrain $\hat{y}^{(i)}$ to take values in the range $[0,1]$ (through the activation function of the final layer) in which case it can be considered that $\hat{y}^{(i)}$ encodes the probability of the i^{th} example to be from class 1, the problem of loss minimization can be thought as maximizing the output probability of membership to class 1 for examples that come from class 1, while minimizing this probability for examples that come from class 0.

- **Categorical Cross-Entropy (CCE)** is the generalization of BCE for a number of classes > 2 and is defined as follows:

$$CCE = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_c^{(i)} \ log_2(\hat{y_c}^{(i)}) \tag{1.5}$$

where C is the number of classes, $y_c^{(i)}$ and $\hat{y_c}^{(i)}$ represent the target probability for the i^{th} example to be from class $c \in \{1 \dots C\}$ and the neural network output probability to be from class c, respectively. The target output vector is represented as $\mathbf{Y}^{(i)} = [y_1^{(i)} y_2^{(i)} ... y_C^{(i)}]$ and should encode the ideal discrete probability density function for the i^{th} example. In almost all cases the target vector $\mathbf{Y}^{(i)}$ contains only one entry equal to 1 and the rest are set to 0. The output of the final layer should be able to encode a discrete probability density function over all C classes through its activation function, like *softmax*. In this case, minimizing the CCE corresponds to maximizing the probability $\hat{y_k}^{(i)}$ for each tuple $(\mathbf{X}^{(i)}, \mathbf{Y}^{(i)})$ with $y_k^{(i)} = 1$ and $y_t^{(i)} = 0$ for $t \in \{1, \dots C\}, t \neq k$.

Training a neural network involves minimizing the total loss through weight optimization. Most weight optimization techniques involve computing the gradient of each weight with respect to the loss function considered for the neural network and updating the weights in the direction of a smaller loss. The **backpropagation** algorithm has been widely used in numerous optimization tasks which consist in training ANNs [1][2]. The algorithm is used for computing the gradient of the loss function with respect to each weight in the neural network, using the *chain rule*, these gradients being further used to update the weights based on a specific rule called the **optimization algorithm**.

Some of the most popular gradient-based optimization algorithms are presented as follows:

- **Gradient Descent** uses the partial derivative of the loss function calculated over the whole dataset to move in the direction of negative gradient, i.e. the direction of the steepest descent, using one *hyperparameter* (non-trainable parameter) called the *learning rate*. The update rule for this algorithm is defined as follows:

$$w_{kj}^{L} := w_{kj}^{L} - \eta \ \frac{\partial \mathcal{L}_{total}}{\partial w_{kj}^{L}} \tag{1.6}$$

where w_{kj}^{L} is the weight connecting the k^{th} neuron from layer $L-1$ and the j^{th} neuron from layer L, η is the learning rate and $\frac{\partial \mathcal{L}_{total}}{\partial w_{kj}^{L}}$ is the partial derivative of the total loss function with respect to the current weight. The disadvantage of this algorithm is that it may result in convergence to local a minimum, i.e. points with zero gradient but which are **not** the global minimum.

- **Stochastic Gradient Descent** uses the partial derivative of the loss function calculated for each example from the dataset, which means the weights are updated a number of times equivalent to the size of the dataset:

$$w_{kj}^L := w_{kj}^L - \eta \ \frac{\partial \mathcal{L}^{(i)}}{\partial w_{kj}^L}, \forall i \in \{1, \ldots, N\}, \tag{1.7}$$

where N is the size of the dataset. Since the network updates its weights after each example from the dataset, the training process will be more chaotic than in the case of Gradient Descent, but it may result in global minimum point due to the high variance of the weights during training. *Mini-batch gradient descent* is the popular trade-off alternative between these two types of gradient descent, and consists in updating the network parameters given the value for the loss function calculated on a mini-batch of examples.

- **Adagrad** is an optimization algorithm which uses a different adaptive learning rate for each parameter of the network, by keeping track of the past gradients:

$$w_{kj}^L(t+1) := w_{kj}^L(t) - \frac{\eta}{\sqrt{\mathcal{U}_{kj}(t) + \epsilon}} \ \frac{\partial \mathcal{L}^{(i)}}{\partial w_{kj}^L(t)}, \forall i \in \{1 \ldots N\} \tag{1.8}$$

$$\mathcal{U}_{kj}(t) = \sum_{p=1}^{t} \sum_{i=1}^{N} \left(\frac{\partial \mathcal{L}^{(i)}}{\partial w_{kj}^L(p)} \right)^2 \tag{1.9}$$

where $w_{kj}^L(t)$ is the weight connecting the k^{th} neuron from layer $L-1$ and the j^{th} neuron from layer L at time step t, $\mathcal{U}_{kj}(t)$ is the sum of squared gradients for the current weight up until time step t and ϵ is a small value to avoid division by 0. This type of update implies that weights which had important updates in the past, i.e. the absolute value of their past derivative was high, will have a lower learning rate in future updates, leading to a faster convergence due to the attenuation of high variance updates for some specific weights. One disadvantage of this algorithm is the summation of **all** past squared derivatives, which can only increase over time, the weights which had many updates in the past coming to a halt, being no longer able to accumulate the information from next derivatives due to the decreasing learning rate. Here the derivative for each weight can be computed for each example of the dataset, or for each mini-batch.

- **Adam** [3] is an optimization algorithm which uses exponential decaying averages for the partial derivatives and the squared partial derivatives with respect to each weight:

$$w_{kj}^L(t+1) := w_{kj}^L(t) - \frac{\eta}{\sqrt{(\hat{m}_2(t) + \epsilon)}} \ \hat{m}_1(t) \tag{1.10}$$

$$\hat{m}_1(t) = \frac{m_1(t)}{1 - \beta_1^t} \quad \hat{m}_2(t) = \frac{m_2(t)}{1 - \beta_2^t} \tag{1.11}$$

$$m_1(t) = \beta_1 \ m_1(t-1) + (1 - \beta_1) \ \frac{\partial \mathcal{L}^{(i)}}{\partial w_{kj}^L(t)} \tag{1.12}$$

$$m_2(t) = \beta_2 \ m_2(t-1) + (1 - \beta_2) \ \left(\frac{\partial \mathcal{L}^{(i)}}{\partial w_{kj}^L(t)} \right)^2, \tag{1.13}$$

where $m_1(t)$, $m_2(t)$ represent the exponential decaying averages for the first order partial

derivatives and squared first order partial derivatives respectively up until time step t, β_1, β_2 are the exponential decay rates for these 2 averages, usually initialized with default values of 0.9 and 0.999, respectively. m_1 and m_2 are initialized with 0 for each weight at the beginning of training. Since this type of initialization will result in a bias towards 0 for all weights, as stated by the authors, this problem is resolved by introducing in the update equations the bias-corrected variants $\hat{m}_1$ and $\hat{m}_2$ in order to account for small exponential averages at the initial time steps of training, since the the denominators of $\hat{m}_1$ and $\hat{m}_2$ will be small subunitary values, resulting in high initial estimates for the actual averages. As the training goes on, the denominator will get closer to 1, since β_1 and β_2 will always be chosen in the range $(0, 1)$, which means the bias-corrected estimates will get closer to the actual exponential moving averages.

During the process of training a NN many issues can arise, such as **overfitting**, **underfitting**, **vanishing gradients** or **exploding gradients**.

Overfitting appears when the neural network perfectly learns the data that it is trained on, performs very well on seen examples, but is unable to generalize for unseen data. This problem can arise from training the network for a long time, forcing it to almost perfectly fit the training data, such that the final model has learned the underlying noise present in the training data, rather than more general characteristics that describes it. Another cause may be the complexity of the NN, a model with a relative high number of parameters being more prone to overfitting than simpler models. Solutions to overcome the problem of overfitting include the adoption of a model with fewer parameters, including weight normalization or weight constraint techniques during the training process in order to account for potentially high weight values that may result in highly confident predictions, or using *early stopping* techniques which imply stopping the training process at the point where the performance on the validation dataset starts to decay.

Opposed to overfitting, underfitting happens when the model is too simple to capture the characteristics of the training dataset, resulting in poor performance on both training and test data. Solutions to this problem include increasing the complexity of the model in order to be able to process and capture the complexity of the task at hand.

Take for example the problem of classifying points based on their x and y coordinates into 2 classes: points inside a circle, and points outside of it, with added noise to both classes, as shown in figure 1.2. Here, the points are coloured according to their class. The background colour signifies the area for which the trained neural network predicts the corresponding class. For all three experiments[1] a network with 2 hidden layers was trained, varying the number of neurons on each layer. For the underfitting experiment the network had 2 neurons per hidden layer, for overfitting there were 16 neurons per hidden layer, and for the balanced model there were 4. It can be seen that for only 2 neurons per hidden layer the model is not capable of extracting the characteristics needed to describe all the green points. On the opposite end, the model with 16 neurons per hidden layer perfectly captures the characteristics of each green point, even the noise that was added, due to its high capacity of memorizing almost every detail. The model with 4 neurons per layer does not perfectly represent each green point, but is able to learn the underlying important features of the green class, that is to represent it as a circle, not focusing on the added noise.

The vanishing gradient problem is encountered in training deep neural networks using gradient-based optimization methods and the backgpropagation algorithm. It consists in the

[1]The experiments were performed using the online simulator at `https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html` using *circle data* setting for which additional points were included

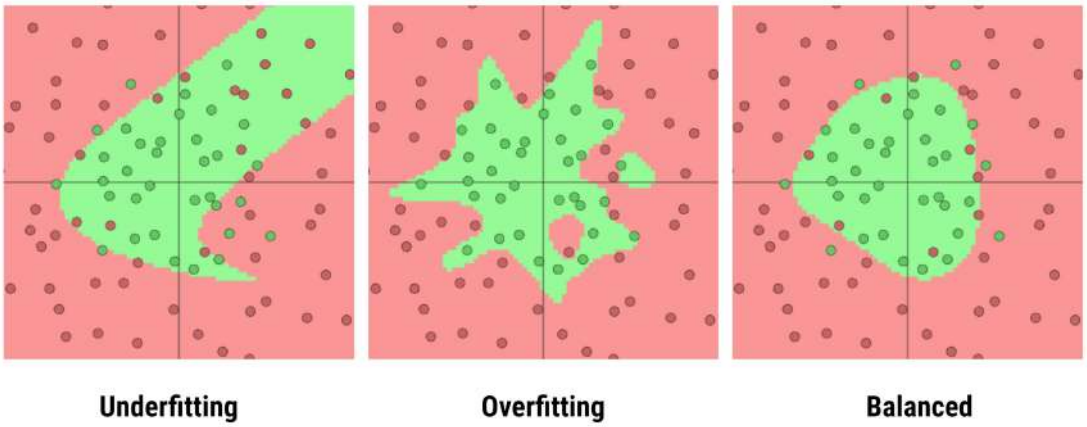


Figure 1.2: Example of an underfitted, overfitted and balanced NN

neural network being unable to propagate information from the output layer back to the initial layers due to exceedingly small gradients of the loss function with respect to the network parameters. The partial derivative of the loss function with respect to the network parameters are expressed as a product using previously computed partial derivatives and activations of next layers, starting from the output. When moving backward towards the input layer, these expression contain more and more such terms, which can cause smaller and smaller partial derivatives for the initial layers of the neural network, meaning that when using a gradient-based optimization algorithm the weights associated with these layers will get significantly smaller updates than layers which are closer to the output layer, if most of the terms are small subunitary values. This problem may arise not necessarily from the high number of hidden layers the network has, but as much as from the choice of activation functions for each layer, since the magnitude of each gradient will depend on it. This problem can also be tackled by using *residual learning* [4] [5] which consists of adding *skip connections* between the input of a layer and the output of next layers, using different techniques to aggregate them. This technique has been proven to ease the training of deep neural networks, since some information from the backpropagated gradients from upper layers is transmitted to earlier layers, along with their own partial derivatives which, as stated above, could be of a much lower magnitude. Opposite to this phenomenon, the exploding gradients problem arises when most of the terms in the expansion of the partial derivatives are supraunitary, resulting in high magnitude updates for weights from the initial layers, which translates in a more chaotic training process due to the high variance of these weights.

As stated above, overfitting may result in potential high weights that may cause the neural network to present a chaotic behaviour when presented with slightly different input data than the one it has been trained on, which translates to a poor generalization capability for unseen data. One of the techniques used to alleviate this problem is through **weight regularization**. Some regularization techniques force the weights of the neural network to have small values by introducing in the expression of the loss function terms which are proportional to the magnitude of weights. Since the optimization algorithm is trying to minimize the loss function, the net result is minimizing the error between the predicted and target outputs while forcing the weights to have smaller values. Some popular regularization techniques are described below:

- **ℓ1 Regularization** introduces in the expression of the loss function a term proportional

to the $\ell 1$-norm of each weight matrix from the NN:

$$\mathcal{L}_{reg} = \mathcal{L} + \lambda \sum_{l=1}^{L} \left\| W^{(l)} \right\|_1 \tag{1.14}$$

$$\left\| W^{(l)} \right\|_1 = \frac{1}{N_l \ M_l} \sum_{i=1}^{N_l} \sum_{j=1}^{M_l} |w_{ij}^{(l)}|, \tag{1.15}$$

where λ is a *penalty hyperparameter* which controls the amount of regularization that is applied, N_l and M_l are the dimensions of the weight matrix W^l. Taking the partial derivative of this new loss with respect to any parameter of the neural network will result in an additional term, for each weight, of magnitude $\frac{\lambda}{N_l \ M_l}$, which means the weights will be penalized by an amount proportional to this term, even if the partial derivative of $\mathcal{L}$ will be close to 0.

- **$\ell 2$ Regularization** introduces in the expression of the loss function a term proportional to the L2-norm of each weight matrix from the NN:

$$\mathcal{L}_{reg} = \mathcal{L} + \lambda \sum_{l=1}^{L} \left\| W^{(l)} \right\|_2 \tag{1.16}$$

$$\left\| W^{(l)} \right\|_2 = \frac{1}{N_l \ M_l} \sum_{i=1}^{N_l} \sum_{j=1}^{M_l} (w_{ij}^{(l)})^2 \tag{1.17}$$

  The main difference between $\ell 1$ and $\ell 2$ regularization is that in the case of using $\ell 2$ regularization the weights will be penalized by an amount that is proportional to their current value, resulting in a much higher penalty for weights with magnitude > 1. As the the weights approach 0, $\ell 2$ regularization effect will decrease since the amount by which it penalizes the weights will also decrease, which is not the case for $\ell 1$ regularization since the magnitude by which they are penalized does not account for the current value of weights.

- **Dropout**[6] is a regularization technique that prevents overfitting by randomly omitting neurons, meaning they are excluded from computing the output vector at some training step. The output vector will thus not depend on any output produced by them (since their output will be automatically set to 0), which will result in no update for the weights associated with these dropped units, at that specific training step. This forces the network not to use its full computational resources to learn the features of each presented example. Using only some of the neurons to learn the current example will also force the network to learn a more shallow description about the current input example, which in turn may result in an increased generalization capability.

As mentioned before, some issues that may arise during the training process can be tackled by choosing the appropriate activation function, like keeping the gradients in a certain range in order to overcome udnerfitting and overfitting. Below are presented some of the popular activation functions extensively used in literature, along with their closed-form expressions:

- **Sigmoid**, also referred to as *logistic function*, has the following analytical form:

$$f : \mathbb{R} \to [0, 1], f(x) = \frac{1}{1 + e^{-x}} \tag{1.18}$$

It is usually used for the output layer of the neural network in the case of binary classification in order to encode the probability that an input example comes from a certain class, by mapping any real number to its correspondent in the range $[0, 1]$. Its main drawback is that as x increases or decreases, the function can be very well approximated by a constant, which means its derivative decreases towards 0. This can result in vanishing gradients, since using this function in the output layer together with the backpropagation algorithm to compute the partial derivatives of the loss function w.r.t. the network parameters will result in insignificant updates, given the output values tend to be in the region for which the sigmoid function is constant. The learning process therefore happens when the input values are in the neighbourhood for which the sigmoid can be approximated by a linear function with non-zero slope.

- **Softmax** is an activation function which extends the sigmoid function to the multi-dimensional case and it is usually used to transform a vector of real values to a vector containing subunitary values which encodes a probability mass function over some discrete set:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N_{out}} e^{x_j}}, \forall i \in \{1, \cdots, N_{out}\} \tag{1.19}$$

  where x_i is the input value for the i^{th} neuron, N_{out} is the number of neurons of the layer the activation is used for, which in most cases is the output layer. In this case, the discrete set representing this probability mass function is given by the classes for which the problem is constructed.

- **ReLU** (Rectified Linear Unit) is another activation function which is defined as follows:

$$f(x) = max(0, x) \tag{1.20}$$

  This activation function has been shown to make the training of NNs more efficient than using the sigmoid function, by increasing the sparsity of representations of the network [7], since it cancels out negative input values. For positive inputs the derivative of the activation function w.r.t. the input will always equal 1, which means for any positive input there will always be a constant update term for the previous weights which produced that input, while for negative inputs the update term will be 0. Since the function does not saturate for large input values it also tackels the problem of vanishing gradients, unlike sigmoid activation function for which high magnitude inputs result in gradients close to 0. However, the vanishing gradient problem may arise at the opposite end, when almost all neurons from a layer using ReLU activation have negative inputs, resulting in mostly 0 gradients, also known as dying ReLU problem[8].

- **Leaky ReLU** is an activation function closely related to ReLU, which is designed to solve the dying ReLU problem, and is defined as follows:

$$f(x) = max(\alpha x, x), \quad \alpha \in (0, 1) \tag{1.21}$$

  For positive inputs the function behaves the same, while for negative inputs the output is a small negative number, proportional to the input, which means a small non-zero positive derivative equal to α. This means that even if all inputs to a layer are negative, gradients still *pass through* when backpropagated, meaning the training process is more likely to continue for earlier layers.

In order to obtain a quantitative measure for the model performance appropriate *metrics* have to be used. In the context of supervised learning these metrics will represent some form

of *distance* between the expected and predicted outputs. One key difference between a loss function, which usually measures the distance between expected and predicted output, is that the function that defines a metric is not constrained to be differentiable, unlike loss functions which need to be minimized through, mostly, gradient-based optimization algorithms. Metrics are also used to monitor the model performance during training in order to spot different abnormalities which may arise at some training steps. Since most real-world problems can be reduced to yes/no (class 1/class 0) decision, some important statistical measures for decision models are constructed for the binary classification task. In this binary setting, there are 4 possible outcomes which are encoded into 4 quantitative measures: *true positives* (TP) – the number of examples classified as class 1 which are from class 1, *false positives* (FP) – the number of examples classified as class 1 which are actually from class 0, *true negatives* (TN) and *false negatives* (FN). The following metrics are defined in terms of these 4 quantitative measures:

- **Accuracy** measures the percentage of the correctly classified examples over both classes:

$$ACC = \frac{TP + TN}{TP + FP + TN + FN} \tag{1.22}$$

  The drawback of using accuracy as the sole measure is that it does not take into account the *class imbalance* problem. Suppose there are 2 classes, the first one with 990 examples and the second one with 10 examples. Always choosing the first class as a prediction regarding the input example yields an accuracy of 0.99, which may not be achievable using a neural network as a prediction model.

- **Sensitivity** (or Recall) measures the proportion of positive examples (class 1) which are correctly classified as positive:

$$Sensitivity(s) = \frac{TP}{TP + FN} \tag{1.23}$$

- **Specificity** measures the proportion of negative examples (class 0) which are correctly classified as negative:

$$Specificity = \frac{TN}{TN + FP} \tag{1.24}$$

  If a classification model would always predict class 1 for any example that is presented, then it would yield a Sensitivity of 1.0, since there would be no false negatives. Similarly, always predicting class 0 would yield in a Specificity of 1.0. Thus, both these measures should be monitored in order to spot different abnormalities during the training process.

- **Precision** measures the proportion of examples predicted as positive that are actually positive:

$$Precision(p) = \frac{TP}{TP + FP} \tag{1.25}$$

  Different from Sensitivity, if the classification model would always predict class 1 then the Precision would be equal to the proportion of examples from the whole dataset which are actually from class 1.

- **F1 score** (*Sørensen–Dice coefficient* or *Dice Similarity coefficient*) is calculated as the harmonic mean between the sensitivity and precision:

$$F1 = 2\,\frac{s\ \ p}{s + p} = \frac{2\ TP}{2\ TP + FN + FP} \tag{1.26}$$

F1 score is usually used for uneven class distributions – the number of negative examples is far bigger than the number of positive examples – achieving a maximum value of 1.0 only when both Sensitivity and Precision are maximum.

1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of ANNs especially designed to process multi-dimensional input data, which have become very popular in computer vision applications like object recognition, object detection, video tracking, image captioning, image segmentation, etc. The core idea of CNNs is the use of *convolution operation* to extract local information from previous layer activations. This reduces the dimension of previous activation maps through different *pooling* techniques in order to transform them to an abstract volume which contains a good description about the type of input the CNN is presented with. Here the term *activation maps* refers to the output volume of the previous layer, which is the result of a convolution operation between the input volume and another same-dimensional array called *filter*, followed by an activation operator. Thus, the 2 core type of layers used for building CNNs are the convolution layers and the pooling layers.

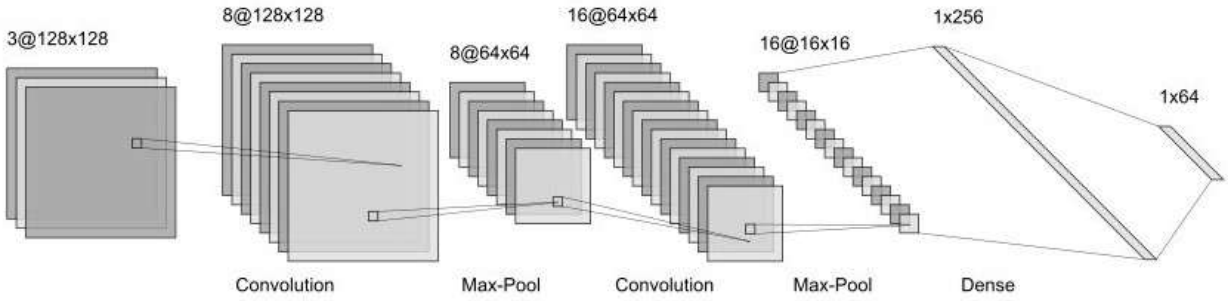


Figure 1.3: Schematic example of a CNN

A **Convolution layer** takes as input an N-dimensional volume and outputs the convolution between its input and another N-dimensional volume called *filter*. The convolution layer contains multiple filters, its final output being the concatenation along some dimension of all convolutions between the input and each filter. Let us consider the case of a 3-dimensional input volume denoted $V_{in} \in \mathbb{R}^{H_{in} \times W_{in} \times D_{in}}$, where H_{in}, W_{in}, D_{in} represent the height, width and depth of the input volume. Usually the filter's first 2 dimensions are equal and much lower than the input's first 2 dimensions, while the filter depth should be equal to the input depth (in the case of a 2D convolution operation – which is performed along 2 dimensions). Thus, we denote the first 2 dimensions of the filter by f, which results in a volume of shape $\mathbb{R}^{f \times f \times D_{in}}$. This means that for computing the next input of each output neuron there needs to be processed a neighbourhood of $f \times f \times D_{in}$ neurons from the input layer, which consists of element-wise product between the filter and that subset. The 2D convolution between the input volume V_{in} can be done using the following formula:

$$V_{out}(x,y) = \sum_{i=-\lfloor \frac{f-1}{2} \rfloor}^{\lfloor \frac{f-1}{2} \rfloor} \sum_{j=-\lfloor \frac{f-1}{2} \rfloor}^{\lfloor \frac{f-1}{2} \rfloor} \sum_{k=1}^{D_{in}} \mathcal{F}_{ijk} \cdot V_{in}(x+i, y+j, k), \forall x = \overline{0, H_{in}-1}, y = \overline{0, W_{in}-1} \quad (1.27)$$

where $\mathcal{F}_{ijk}$ is the value of filter $\mathcal{F}$ which corresponds to a relative position of (i,j,k) in the current neighbourhood. The problem of applying the filter at *margin* neurons (neurons who

do not belong to a full $f \times f$ neighbourhood) can be solved through *padding*, which consists of adding additional rows/columns in order to account for the missing points in the neighbourhood set of these neurons. Usually, padding is done by adding rows/columns of 0, or by repeating margin rows/columns. Let us denote the number of additional rows/columns added in both directions as p. Another aspect of the convolution operation is the *stride* that is used to perform it. The stride controls how the filter convolves on the input volume, meaning how many units it moves after one element-wise product, i.e. how many input neurons it skips before another computation. Let us denote this number of units as s. If the output convolution volume is denoted as $V_{out} \in \mathbb{R}^{H_{out} \times W_{out} \times D_{out}}$, then the following relations hold:

$$H_{out} = \left\lfloor \frac{H_{in} - f + 2p}{s} \right\rfloor + 1 \tag{1.28}$$

$$W_{out} = \left\lfloor \frac{W_{in} - f + 2p}{s} \right\rfloor + 1 \tag{1.29}$$

and D_{out} is equal to the number of filters for the current convolutional layer, considering the final output is obtained by concatenating along the depth axis all convolutions between the input volume and each filter. In Figure 1.3 an example of a standard CNN is shown. Take for example the first convolution operation which takes as input a volume with shape parameters $D_{in} = 3$, $H_{in} = W_{in} = 128$. The output volume has the same width and height, which can be achieved by padding the input with additional rows and columns. The output depth, $D_{out} = 8$, results from the fact that the first convolution layer contains 8 filters. The input region used to compute one feature from the output volume is called the *receptive field*. The size of the receptive field is given by the width and height of the filter used to compute the output 2D array.

A **Pooling layer** consists of reducing the input volume size by a certain factor along certain dimensions, reducing the representation space while keeping most of the input information. Take for example the pooling operations shown in Figure 1.3. The type of pooling used here is called *max pooling* which slides a window of predefined shape over each 2D array from the input volume and gives out the maximum value from that neighbourhood. For example, the first pooling operation takes an input volume with shape parameters $D_{in} = 8$, $H_{in} = W_{in} = 128$ and slides a 2×2 window, resulting in a reduction factor of 2 along width and height axis for each 2D array.

CNNs have been widely used for computer vision applications due to their most important characteristics, which are **local connectivity** and **parameter sharing**.

Local connectivity implies that the output neuron should not depend on all input neurons, but only on a specific group from the input layer. This is an important aspect in computer vision since spatially close pixels are highly correlated, while far away pixels usually do not exhibit any correlation.

Weight sharing means using the same parameters to extract features from all neighbourhoods of neurons in the input volume, which is what convolving the input volume with a filter represents. This idea is backed by the fact that the same features in an image can be found in multiple places, not just some predefined area. Weight sharing also implies a significant reduction of parameters to be tuned, which is one of the reasons deep CNNs have become ubiquitous in computer vision tasks. Take for comparison the task of extracting features from an RGB image using a dense neural network and a CNN, to further be used for classification. If the image has a resolution of 7680×4320 pixels and the first hidden layer of the dense neural network has N_h neurons then the number of weights to be tuned for this layer is $7680 \times 4320 \times 3 \times N_h$, which is a very large number even for small N_h. When using the CNN, the number of weights associated with the first convolutional layer is equal to the number of filters used times the size

of each filter. For example, using 64 filters with a size of 5×5 the total number of weights will be $64 \times 5 \times 5 \times 3$, which is significantly smaller than the number of parameters for a dense network. Another disadvantage for dense network applied to image processing is that the number of parameters for the first layer will increase linearly with image resolution, which is not the case for CNNs, since filter size does not directly depend on image resolution.

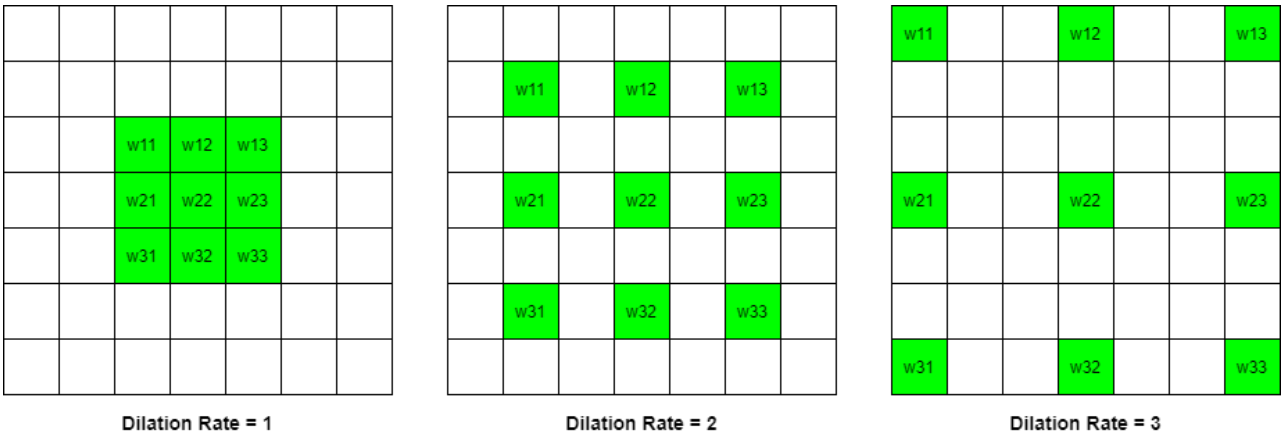


Figure 1.4: 3×3 kernel with different dilation rates

The convolution operations discussed earlier can be more generally referred to as *1-dilated convolutions*, in which the filter is applied to the current input as shown in the leftmost picture from Figure 1.4, considering a 3×3 filter size. Here $w_{i,j}$ represents the filter coefficient at position (i, j) in the 2D filter array. In order to increase the receptive field of a filter, without increasing its size one can use *l-dilated convolutions*, with $l \in \mathbb{N}$, $l > 1$. Figure 1.4 shows how a 2-dilated and 3-dilated convolution is performed using a 3×3 filter. For a 2-dilated convolution the 3×3 filter is expanded (dilated) by introducing one 0 between consecutive filter coefficients, resulting in a neighbourhood of 5×5 in the input array on which the filter is applied. Similarly, for a 3-dilated convolution the filter is expanded by introducing two 0 values between consecutive filter coefficients, resulting in a 7×7 input neighbourhood. It can be seen that the receptive field increases with dilation rate, at no additional cost in terms of number of filter parameters. Dilated convolutions have been used in computer vision tasks such as semantic segmentation [14], [15] in which the authors show that using dilated convolutions in feature extracting modules increases the accuracy of state-of-the-art segmentation architectures. However, using higher dilation rates along with small filter sizes may lead to information loss, since finer details may be omitted from the input on which the convolution is performed.

1.3 Generative Adversarial Neural Networks

Data generation models are a class of machine learning algorithms used to uncover patterns in data distributions. These systems are able to tune their parameters in order to generate new data which is similar to the distribution it is presented with. One important class of such algorithms is represented by **Generative Adversarial Networks (GAN)** which have been extensively used for generating data that could be easily identified as coming from the real distribution of images, text, speech, or any other distribution it has been trained on.

GANs were first introduced by Ian Goodfellow in [9], where the authors present them as a framework containing 2 NNs, a *Generator* and a *Discriminator*, as shown in figure 1.5, which are trained in an alternate manner. The generator model receives as input data noise samples coming from a certain probability distribution. Some following works on GANs also consider inserting additional information along with the input noise vector, e.g. with the

intent to partially control the characteristics of the generated data. The job of the generator model is to produce data samples that are as close as possible to the real data distribution. The discriminator model receives at a given time either data samples coming from the real distribution or data that is generated by the generator model. The discriminator job is to discern between samples coming from the real distribution and samples coming from the generator model. Usually, the discriminator outputs the probability that the input sample comes from the real distribution.

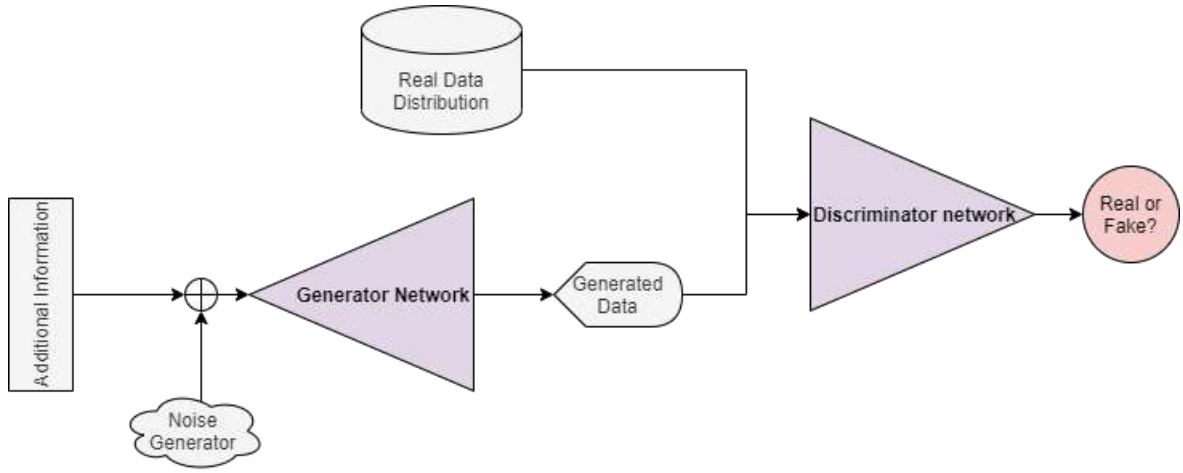


Figure 1.5: Schematic example of a general architecture for GAN

Alternate training is used to train both the generator and the discriminator, which consists in a forward pass for the generator in which fake samples are generated, followed by 2 forward passes for the discriminator, in which first the generated samples are passed and then real samples obtained from real data distribution. This results in an error term associated to the discriminator model. The error term is further used for back-propagation in order to update the parameters for the discriminator, such that the discriminator will maximize the output probability when presented with real samples, and minimize it when presented with fake examples. The generator will update its parameters in order to maximize the error associated with the discriminator, which means it will force its own parameters to change in order to produce realistic data samples, provided that the discriminator has sufficiently learned the differences between real and fake examples in the current training step. As training goes on, the generator will learn to generate more and more realistic data samples, for which the discriminator will be unable to differentiate from real samples. Additionally, apart from maximizing the error associated with the discriminator network, other terms can be added in the expression of the generator's loss function in order to obtain generated samples that match other additional criteria.

Let us consider the output of the generator network to be denoted as $G(z)$, where z is the input noise vector, with some additional deterministic information added. Let $D(c)$ be the probability given by the discriminator, i.e. the probability that input c comes from the real data distribution. If, for example, the discriminator uses the binary cross-entropy loss in order to optimize its parameters, then the losses associated with the two models can be written as:

$$\mathcal{L}_D = \frac{1}{N}\sum_{i=1}^{N} log(D(x^{(i)})) + log(1 - D(G(z^{(i)}))) \tag{1.30}$$

$$\mathcal{L}_G = \frac{1}{N}\sum_{i=1}^{N} log(D(G(z^{(i)}))) + \mathcal{L}_g \tag{1.31}$$

where N is the number of samples used in an epoch/mini-batch, $\mathcal{L}_D, \mathcal{L}_G$ are the losses for the discriminator and generator networks and $\mathcal{L}_g$ is an additional term which imposes some optimization criteria on the generator's parameters. This writing implies that by minimizing $\mathcal{L}_D$ the discriminator will try to minimize the probability that a generated sample is categorized as real, while minimizing $\mathcal{L}_G$ means maximizing that same probability, forcing the generator to produce more realistic samples, by increasing the output probability of the discriminator.

Mode collapse is one of the problems that may arise during the training process of GANs, and consists in the generator network being stuck generating samples which are approximately the same, i.e. the variance between generated samples is almost 0. The goal of each generator is to approximate as good as possible a function which maps one distribution (from which the noise input vectors are generated) to another distribution (the distribution of real data samples). If the generator has come to a point when it produces some very few plausible samples for which the discriminator network fails to recognize as fake data, that may lead the generator to produce *only* those outputs, since the error terms for those samples will be small. Generating the same few data samples for multiple iterations, the discriminator will be able to learn which input will correspond to the generator network. Therefore, the discriminator will backpropagate its error terms in order to update the generator's parameters, but since the generated samples will be very close to each other the mode collapse problem still remains, the generator still producing very similar data samples which will now be modified in order to account for previous error terms. Mode collapse can also happen the other way around, with the discriminator network being stuck in a state which it has not yet learned the differences between real and fake samples, the job for the generator will be easier, since it will have to generate data samples for a weakly trained discriminator.

GANs can be modified such that the generation process is conditioned by some additional information. This information thus has to be added along with the noise to the generator input, and also to the discriminator along with the real and generated data samples. This modification to the classic GAN framework has been introduced as Conditional GAN [10] where it has been applied to generating samples from the MNIST dataset[11] by conditioning the generation process with one of the 10 classes existent in the real data distribution, in an effort to obtain control for the type of data that is being generated.

Deep Convolutional GAN (DCGAN) [12] is the first GAN framework which introduced advances in the image generation process. It implies transposed convolution layers in the generator network which builds up high-level representations from the input noise vector, while eliminating the fully-connected layers. This is done in order to reduce the number of parameters, which leads to an easier training process for deep architectures. They replace the pooling operation with *strided convolution* which allows the discriminator to learn its own down-sampling, instead of using a deterministic down-sampling like *max-pooling* or *average pooling*. Additionally, they use *batch normalization* [13] which normalizes the activation maps with respect to mini-batch statistics in order to stabilize the training process and to prevent mode collapse. Batch normalization is designed to re-center and re-scale the input activation array, by a 2-step process:

1. Transforming the input array $\mathbf{x} \in \mathbb{R}^{N_B \times N_D}$, where N_B is the number of data samples from the current mini-batch and N_D is the number of dimensions for each activation array, into another array with mean 0 and variance 1 along each of the N_D axes:

$$\mu^k = \frac{1}{N_B} \sum_{i=1}^{N_B} x_i^k, \quad \sigma^k = \frac{1}{N_B} \sum_{i=1}^{N_B} (x_i^k - \mu^k)^2, \quad k \in [1, N_D] \tag{1.32}$$

$$\hat{x}_i^k = \frac{x_i^k - \mu^k}{\sqrt{\sigma^k}}, \quad k \in [1, N_D], i \in [1, N_B] \tag{1.33}$$

where $\hat{x}_i^k$ is the k^{th} activation element of the i^{th} activation array from the current mini-batch. Now the activation elements along each of the N_D axes have 0 mean and variance 1, for the current mini-batch.

2. Converting the previously transformed array to another by re-scaling and re-centering using optimized parameters:

$$y_i^k = \gamma^k \ \hat{x}_i^k + \beta^k, \quad k \in [1, N_D] \tag{1.34}$$

where y_i^k is the transformed output for the k^{th} activation element from the i^{th} activation array from the current mini-batch, γ^k and β^k are learned through optimization.

Batch normalization applied to convolutional layers normalizes the activations for each individual feature map (the output obtained after the convolution with one filter, which is a 2D array) using different parameters γ and β for each map. It has been especially used in order to address the problem of *internal covariate shift* which arises due to the randomness present in each mini-batch of data, which forces the parameters of the network to adapt to a new distribution of inputs at each iteration. By re-scaling the activations of each mini-batch in a certain range of values, the parameters should no longer be forced to readjust to new distributions of the input.

GANs have been recently used as **image-to-image translation** architectures in computer vision tasks such as image segmentation, semantic labels to photo, sketch to photo, etc. One of the first GAN architectures constructed for such tasks is *Pix2Pix* [16] in which they frame the generation process as being conditioned not on a noise vector and some class information, but on another image. The generator thus receives as input an image and is trained to produce another image, while the discriminator receives both input and output images, as different pairs for real and fake output images, and tries to determine whether it is a generated sample or not. The authors eliminate the additional noise input vector, which results in a deterministic output for each input image. However, they keep Dropout during the test process, which results in stochastic outputs, since the neurons are randomly eliminated at each prediction step. Also, in order to increase the diversity of the generated images, they use batch normalization at test time, using the statistics obtained on the current test mini-batch, different from the usual protocol where the statistics from the training set are used. An example of an image-to-image translation GAN architecture can be visualized in Figure 1.6, where examples passed to the discriminator network are two concatenated images, i.e. the input image presented to the generator network and the real/generated image.

1.4 State-of-the-art for Biomedical Image Segmentation

Image Segmentation is a computer vision task which consists in classifying each pixel from an image into a certain class. Each image segmentation task can be classified in one of the two categories: *semantic segmentation* and *instance segmentation*. Semantic segmentation consists in assigning a label to each pixel from the input image, not taking into account if there are multiple *objects* of that class present in the image. Instance segmentation assigns different labels for each object from the same class. The difference between semantic and instance segmentation can be visualized on the example task of segmenting different types of geometric figures from a picture, displayed in Figure 1.7. In both middle and right-most picture the hue of each segmented object gives the class, with the difference that in the case of instance segmentation the saturation slightly varies in order to emphasize that each object from that given class is segmented as one individual instance.

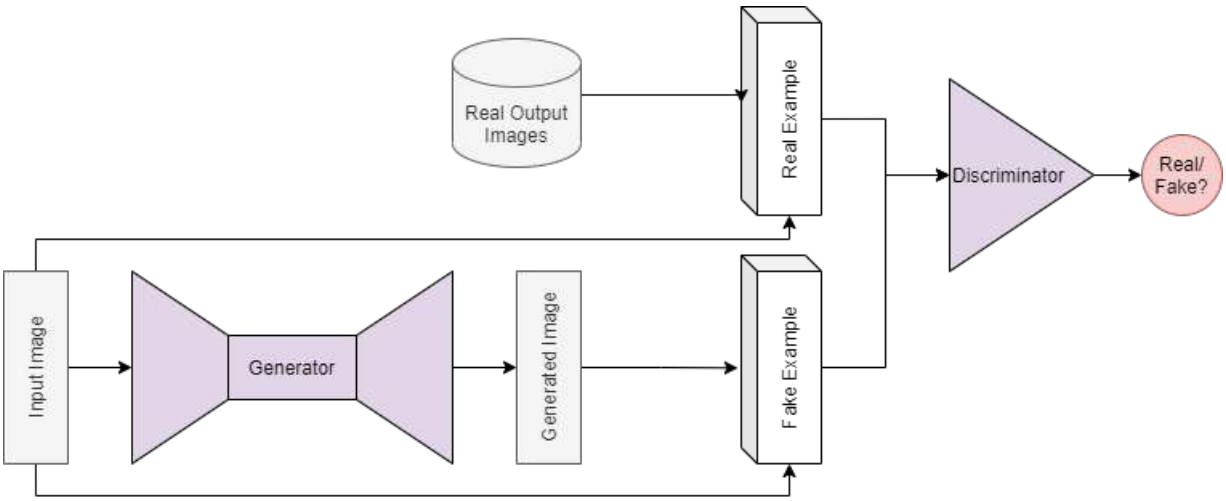


Figure 1.6: Schematic example of an Image-to-Image Translation GAN architecture

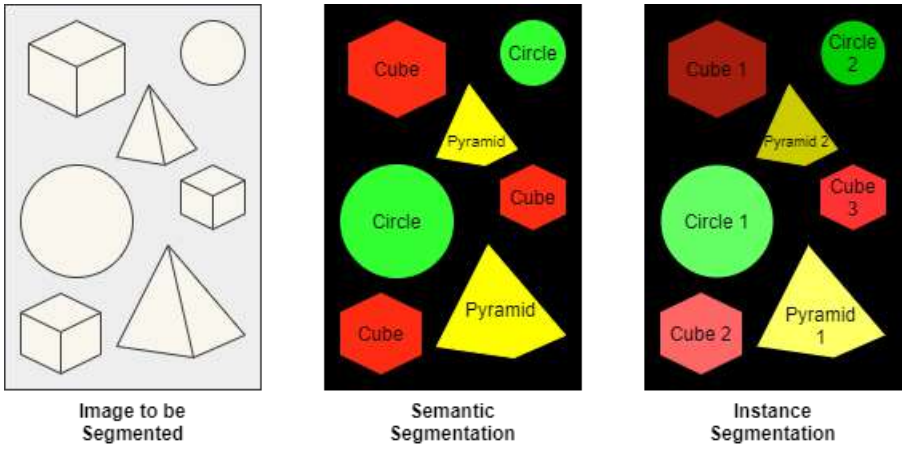


Figure 1.7: Comparison between semantic and instance segmentation

Traditional image segmentation approaches include methods based on *thresholding* and *region growing*. The former one uses the pixel intensity as guidance to segment different objects, considering that these objects present an almost uniform intensity value (their pixel values are concentrated in a small vicinity of intensities). Thus, the problem of segmentation reduces to finding these concentrated areas, by imposing a number of thresholds on the intensity space such that a region between two consecutive thresholds will define a specific class of objects. However, if the considered objects have a very high variability of pixel intensity due to external sources, such as natural light or even due to their intrinsic characteristics the method will fail to recognize it as an object. Region growing works by choosing an initial set of points called *seed points* followed by an expansion process according to their nearby pixels, based on some criteria such as grayscale intensity or colour. If the image histogram is known, then it can be used in order to build a different criteria for neighbourhood expansion, such as determining the maximum allowed difference between a seed and a nearby pixel for them to be considered as part of the same object. This method also assumes that the pixel intensity for all pixels which form an object is approximately the same, implying the same problems as threshold-based methods. *K-means clustering* represents an unsupervised learning algorithm previously used for image segmentation. It consists in forming a predefined number of clusters, in an iterative manner, starting from an initial set of points, called *centroids*, and assigning each other point in the input space to one of them based on some similarity criteria. After assigning each pixel to one centroid, the algorithm recalculates each centroid as the mean of its assigned pixels, and

reassigns each pixel to one of the new centroids. The algorithm continues until the relative movement of each centroid from its last position is sufficiently small. No prior information about what type of objects are presented in the image is needed, however the number of initial centroids will determine the number of objects the segmentation map will contain. Since the criteria by which each pixel is assigned to a centroid is usually intensity-based, not taking into account spatial information of the data, the algorithm suffers from the same problems as the previous two methods. The performance of *K-means*, which is a greedy approach, assigning each pixel to the most similar centroid, may be improved by introducing some degree of randomness, which is what *Fuzzy C-means* does. Fuzzy C-means considers for each pixel in the input space a degree of belonging to each centroid, which are used to calculate the new centroids as weighted sums between all pixels and their degree of belonging to that centroid. Thus, in the calculation of each centroid all pixels will be considered.

Biomedical Image Segmentation consists in segmenting a biomedical image, such as computer tomography, radiography, magnetic resonance imaging, ultrasound etc., in order to represent anatomic objects of interest like organs, lesions, bone structures etc. The lack of substantial annotated datasets for many medical segmentation tasks, along with the high variability between data coming from different patients have been some of the most challenging problems in this domain. Automatic medical object segmentation has been one of the many fields of work where deep NNs have been used to eliminate the necessity of manual annotation, in this case medical objects annotated by medical doctors. Further diagnostics can be drawn using these segmented objects, extracting characteristics such as the area occupied by that specific object, its relative position to another reference object, its shape etc.

One of the most prominent neural network architecture used as a baseline model for various medical segmentation tasks is **U-net**[17]. U-net is a fully convolutional neural network, which consists of an *encoder* and a *decoder* part, as shown in Figure 1.8. The encoder part progressively reduces the input image size using convolution operations followed by pooling layers, to obtain high-level abstract representations. This features are further up-sampled through the decoder part, using either different interpolation techniques or transposed convolution operations to produce the final segmentation map. One of the most important features introduced by U-net architecture is the use of *skip-connections* between output activation maps from the encoding part of the NN, and the up-sampled symmetric activation maps from the decoder part. The motivation behind skip connections comes from the fact that some information is lost in the encoder part due to successive down-sampling, information being passed to upper decoding layers before being contracted to smaller-size representations.

Since the publishing of the U-net segmentation model many biomedical image segmentation NNs have been built upon the underlying ideas of this architecture. UNet++ [18] is an encoder-decoder based architecture which tries to address the issues of the initial U-net, where the low-level feature maps obtained in the decoding layers are *directly* concatenated with the high-level up-sampled feature maps from the decoding layers. This is done by introducing different aggregating steps for low-level feature maps, i.e. using the information from the upper encoding layers to account for the difference between the obtained representations from lower encoding layers, before being passed to the decoding layers. Another feature of the UNet++ architecture is the use of *deep supervision* which forces the model to learn multiple-level representations, apart from the final up-sampled one, at multiple steps during the propagation of the input image. The authors show that these modifications allow the UNet++ to achieve better performance on tasks such as polyp segmentation, liver segmentation and lung nodule segmentation than the U-Net architecture. Another modified U-net architecture is the Attention U-Net [19] which has been used for accurate pancreas segmentation from computer tomography scans. The main modification is the integration of *attention gates* which filter the feature maps propagated from encoding layers to decoding layers in order to filter out irrelevant artifacts,

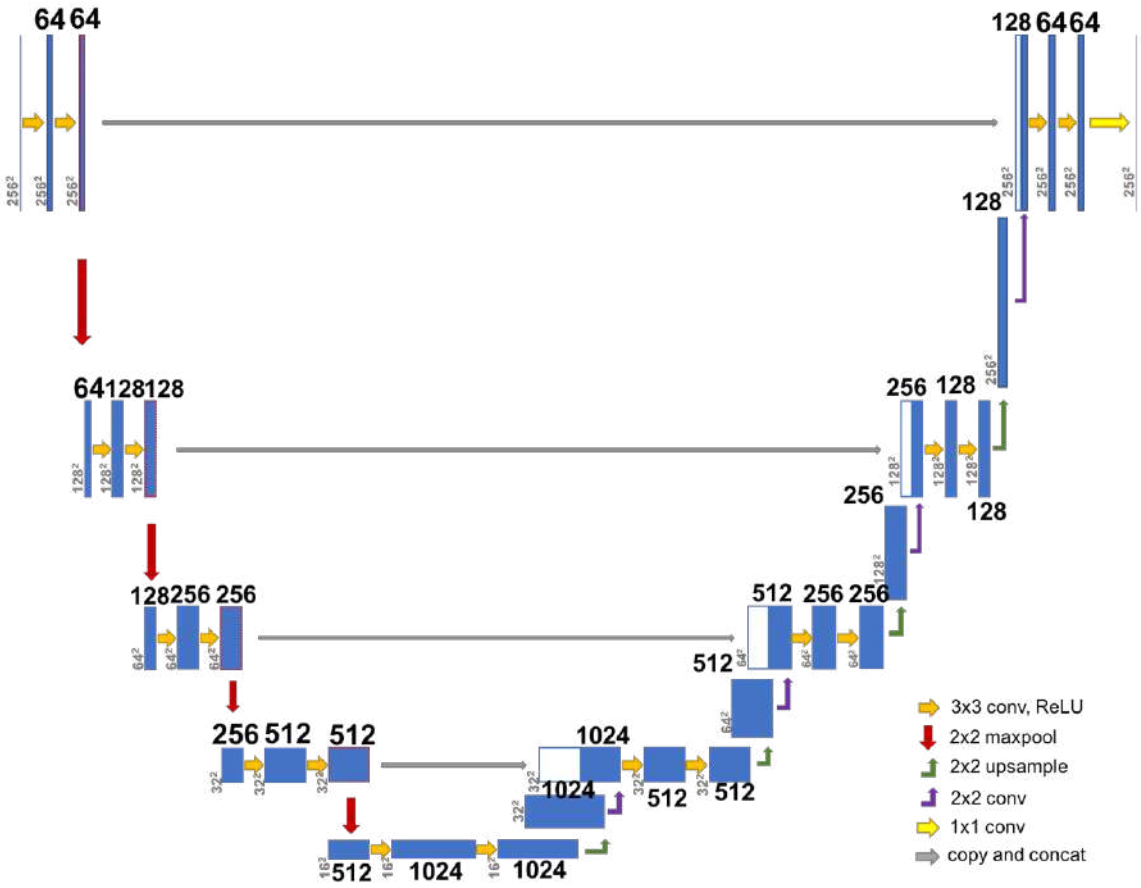


Figure 1.8: U-net architecture[17]

such as excessively high activation outputs in areas that roughly represent any organ tissue, e.g. background. This modification allows the network to gain better performance on tasks such as pancreas segmentation and breast lesion segmentation [20] than the traditional U-net architecture.

The previously presented architectures were designed to segment regions of interest from 2D inputs. However, many biomedical image data comes in a 3D format, which inspired the community to come up with segmentation architectures operating directly on the 3D volume of anatomical data. One such architecture is *V-net* [21], which is very similar to the initial U-net, the main difference being that it uses 3D (instead of 2D) convolution operations, by taking as input a volume of anatomical data and successively down-sampling it using 3D non-overlapping (strided) convolutions instead of usual pooling operations, followed by transposed convolution which build up the 3D segmentation volume.

Another segmentation architecture which was used for accurate lung lesion segmentation and classification is the *Inf-Net* [22]. The architecture uses different computational blocks to build up representations for the final segmentation map, using convolutional blocks to extract 2 sets of low-level features, which serve for predicting the edge map for the considered lesions, followed by 3 sets of high-level features which are used for multiple-level supervision, as shown in Figure 1.9. The 3 high-level feature maps (f_3, f_4, f_5) are combined using a *parallel partial decoder* in order to account for the differences between them, resulting in a final output which is forced through deep supervision to produce the initial segmentation map (S_g). The architecture also uses deep supervision at each of the final 3 convolutional layers, in order to refine the segmentation map produced by the parallel partial decoder module. This is done with the aid of a *reverse attention module* at each of the 3 layers, which takes as input the high-level

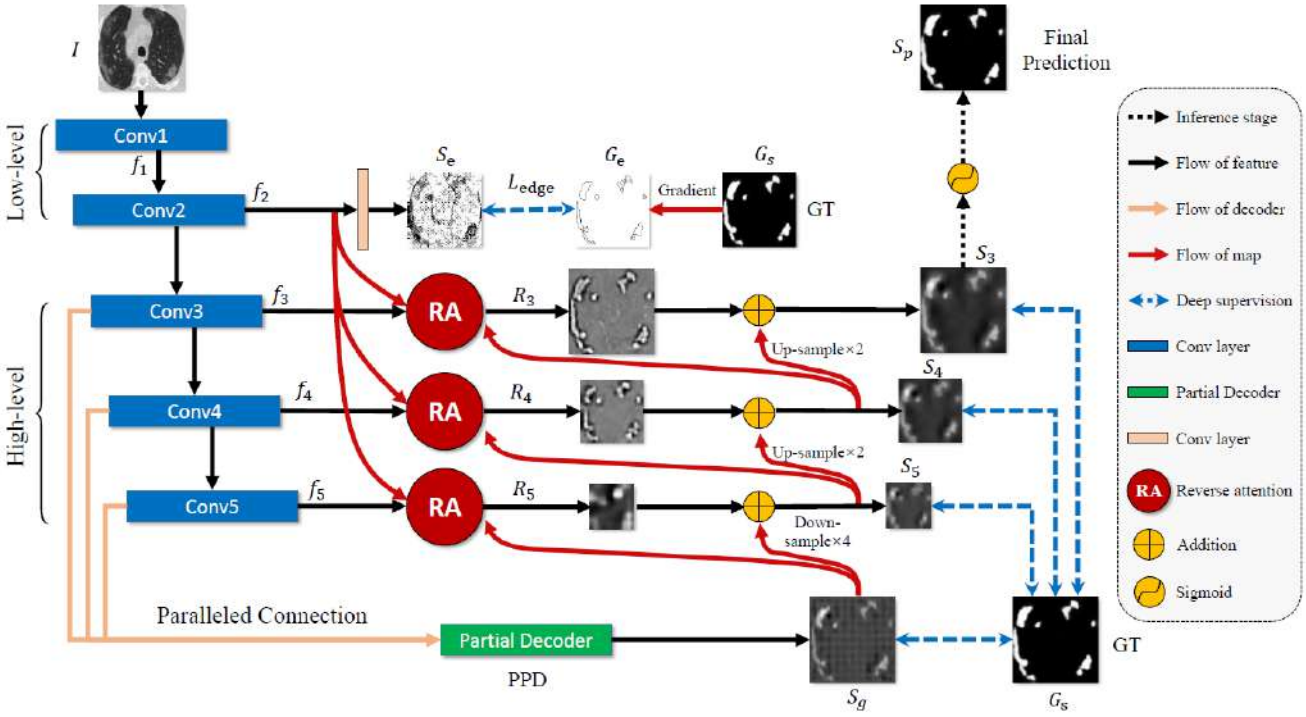


Figure 1.9: Inf-Net architecture[22]

feature map, the segmentation map of the next higher level subject to deep supervision and the low-level feature map (f_2) used to produce the edge map (S_e). This way, the authors claim that the network is able to build up better and better representations of the segmentation map, starting from the initial output map. The main goal is to construct more precise maps by combining lower-level features and edge guidance features. For every input CT image, the final prediction will be the activation map obtained at the third level that implies a reverse attention module. combining all previously extracted information from the upper levels. The architecture is used for lesion segmentation, the results being forwarded, along with the input CT image, to another fully convolutional network (trained separtely) which outputs the class probabilities for each type of lesion, for the segmented areas. The authors trained the segmentation network in a *semi-supervised* manner. The motivation behind using a semi-supervised framework for training comes from the shortage of annotated data. In this context, the model is first trained with labeled data, i.e. CT images with corresponding true segmentation maps, then the model is presented with unlabeled CT images, which produces segmentation outputs further called *pseudo-labels*. These unlabeled examples along with the labeled CT images (which right now is a fully labeled set) are then used as a new training set for refining the model's parameters. This segmentation architecture trained in a semi-supervised manner outperforms the U-Net and its variants on the lesion segmentation task on lung CT images.

Recent works in the field of medical image segmentation have used GANs as image-to-image translation frameworks for the task of segmenting anatomical objects of interest. The majority of these works use a U-Net-like generator with a standard segmentation loss, along with a discriminator network which is assumed that it performs optimally at each time step, thus returning a different error term which forces the generator to modify its parameters accordingly. This way, the framework allows the generator to learn its own loss function, defined by the segmentation loss and the error produced by the optimized discriminator.

One of these frameworks is *SCAN* [23] which has been used for organ segmentation from chest X-rays. It uses a fully convolutional network for generating the segmentation maps, along with a discriminator which receives as input the predicted and actual segmentation maps. Prior

to the actul framework training, they pre-train their generator, claiming that these additional training steps help reduce the mode collapse problem. Th resulted GAN architecture is shown to perform better at organ segmentation than the fully convolutional generator trained without a discriminator.

Another GAN architecture is *SegAN*[24] which does not utilize any additional loss for the generator network; both the generator and discriminator being trained to minimize and maximize, respectively, the same loss function. The discriminator receives as input the original image masked by either the ground truth or the predicted segmentation map. The authors propose a multi-scale $\ell 1$ loss function which employs the feature maps obtained at each level in the discriminator network. The discriminator job is to increase the $\ell 1$-distance between the activation maps obtained for real masked examples and fake masked examples, while the generator tries to minimize the same distance function, forcing itself to build better segmentation maps for each input image. This architecture obtains better performance than the standard U-Net on the task of brain tumor segmentation from MRI images.

Chapter 2
Lung CT Databases

2.1 Lung CT Scans

A *Computed Tomography (CT)* scan is a medical imaging technique which uses an X-ray apparatus to obtain multiple cross-sectional images of the body and advanced computer processing to combine these sections, producing realistic internal volumes. An X-ray generator rotates around the body, with detectors placed on the opposite end to acquire the raw data, which is further reconstructed to produce the final images. This technique provides more details of different tissues, bones and organs than the traditional X-ray techniques. Studies show that CT scans can be used as a preventive medicine option, with applications such as detecting bone trauma, head tumors, thyroid abnormalities or coronary artery disease. Applications of CT scans extend to the industrial domain, in which they are used for inspecting different components for fault analysis. Another important application of CT scans in the industrial domain is explosive material scanning in the field of transport security.

The pixels which constitute each obtained image from a CT scanner are represented on the *Hounsfield scale*, which is a measure for describing relative radiodensity of a material, i.e. the opacity of a material to radio waves and X-rays. The Hounsfield number for a tissue is calculated using the linear attenuation coefficient of that particular tissue, which is obtained through measuring the intensity of the transmitted and received wave. By convention, the Hounsfield unit computed for distilled water at standard pressure ($100kPa$) and temperature ($273.15K$) is zero, while the Hounsfield unit of air for the exact same conditions is -1000. The CT scanners which obey these conventions are thus calibrated using water as a reference, which is the most common standard among medical-grade CT scans. The dynamic range of CT scans is very high, ranging from -1024 to several thousands Hounsfield units. Specific ranges on the Housfield scale are used in order to display these images, truncating values which extend above the predefined limits, depending on the part of the body which has been scanned. These ranges have to be chosen according to the radiodensity of any objects of interest which are intended to be visualized.

Lung CT screening has been used for examining abnormalities that may represent causes of different chest symptoms. Such abnormalities could be represented by changes in lung parenchyma – the substance outside the circulatory system containing pulmonary alveoli and respiratory bronchioles – which are not visible through 2-dimensional X-ray techniques. It was found that analyzing lung CT scans represents a very effective lung cancer diagnosing technique, even in the earliest stages. In [25] the authors state that specific lung abnormalities from CT scans, such as ground-glass opacities, have been reported in a significant number of confirmed

cases of the new coronavirus disease (COVID-19). This study claims that the final diagnostic of patients with moderate to severe symptoms could benefit from chest imaging, even if the real-time reverse transcription–polymerase chain reaction (RT-PCR) test is negative, or it is not available at that moment. Correlating chest CT findings along with epidemiologic history and RT-PCR test results could help produce a better diagnostic and lead to a better estimation of the disease extent.

2.2 Databases Description

In this work, two different online databases containing annotated chest CT lung volumes with different lesions were used.

The first database[1] contains 9 volumetric chest CT scans, along with the corresponding annotations for lungs and lesions (*ground-glass opacities* and *consolidation*). All 9 volumes result in a total of 829 2D slices, out of which 373 have been evaluated as positive (containing lesion areas) and segmented accordingly.

The 9 volumes, lung and lesion annotations are each stored in separate `.nii` type files – one of the most common file format for storing multi-dimensional neuroimaging data. An example of three consecutive slices extracted from the same CT volume, along with the annotation maps for lungs and lesions, can be visualized in Figure 2.1. Here the blue color represents unaffected lung regions, green indicates ground-glass opacities and signifies indicates consolidation.

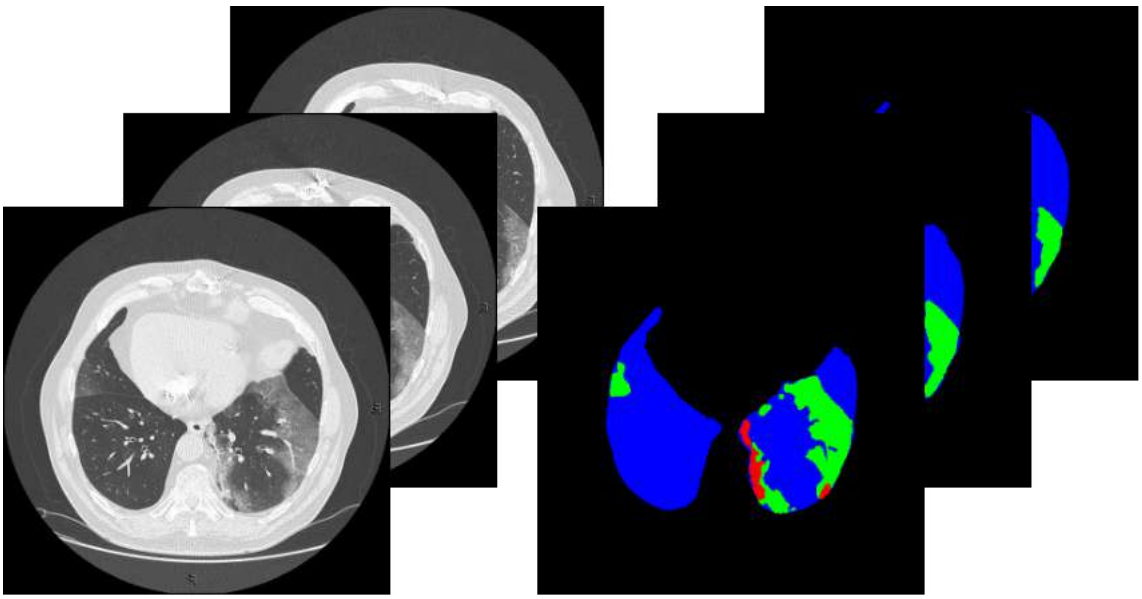


Figure 2.1: Slice examples from the first dataset

This dataset was split into two disjoint groups of patients, one which will be used for tuning the network parameters (training) and the other for validation during the training phase. The training group contains 6 volumes (patients 1, 2, 9, 4, 7, 8), totaling 200 slices, and the validation group contains the other 3 volumes (patients 5, 6, 3) from the dataset, with a total of 173 slices, keeping only slices that contain lesion areas.

The second dataset, detailed in [26], contains 20 labeled chest CT scans obtained from COVID-19 patients, out of which only the 10 volumes from *Coronacases Initiative* were kept. These 10 volumes result in a total of 1351 slices that contain infected regions. These slices will

[1] `https://medicalsegmentation.com/covid19/` – Segmentation dataset no. 2

be used in the testing phase of the final architectures, in order to asses their generalization capabilities when presented with a different distribution of CT volumes.

2.3 Preprocessing

An important requirement for almost any deep learning algorithm is dataset standardization – applying a predetermined set of data processing techniques in order for all input data to meet the same pre-defined constraints. The importance of data standardization is amplified in the context of dealing with data coming from multiple sources (distributions), which in this case are represented by different patients from which CT scans were collected.

The experimental data is represented by the set of slices which contain any lesion areas, extracted from all available CT volumes. Consider $X^{(i)} \in \mathbb{R}^{n \times n}$ to be a 2D slice containing lesion areas, represented on the Hounsfield scale, and $Y^{(i)} \in \{0, 1, \ldots, k, k+1\}^{n \times n \times (k+2)}$ the corresponding segmentation map, where k is the number of distinct lesions, 0 corresponds to background, 1 indicates lung areas and any other number represents a lesion area. Then, the preprocessing steps applied on the pair $(X^{(i)}, Y^{(i)})$ are:

1. Truncate slice $X^{(i)}$ in the range $[-1000, 1000]$:

$$X^{(i)}[l,c] = \begin{cases} 1000, & 1000 < X^{(i)}[l,c] \\ -1000, & X^{(i)}[l,c] < -1000 \\ X^{(i)}[l,c], & -1000 \leq X^{(i)}[l,c] \leq 1000 \end{cases}$$

   $\forall (l,c) \in \{0, 1, \ldots, n-1\}^2$

2. Normalize truncated slice in the range $[0, 1]$:

$$X^{(i)}[l,c] = \frac{X^{(i)}[l,c] - \min_{0 \leq l,c \leq n-1} X^{(i)}}{\max_{0 \leq l,c \leq n-1} X^{(i)} - \min_{0 \leq l,c \leq n-1} X^{(i)}}$$

3. Resize normalized slice to 512×512 pixels using *bilinear interpolation*, which uses linear interpolation along one dimension, followed by another linear interpolation in the second dimension

4. Construct the ground-truth segmentation map $Y_{gt}^{(i)} \in \mathbb{R}^{512 \times 512}$ which will be used for training the lesion segmentation network:

$$Y_{gt}^{(i)}[l,c] = \begin{cases} 1, & Y^{(i)}[l,c] \in \{2, \ldots, k+1\} \\ 0, & Y^{(i)}[l,c] \in \{0, 1\} \end{cases}$$

The choice of truncating the slice in that specific range comes from the fact that bones have an approximate value of 1000 on the Hounsfield scale, which can be only surpassed by the presence of metallic foreign objects. There are also other parts such as dense bones that can surpass 1000 units, which can have an impact on the subsequent normalization, providing that most lesions that appear on lung parenchyma do not surpass this range.

Figure 2.2 presents 6 examples of CT slices before and after truncation and normalization, along with the ground-truth segmentation map. The first 3 columns contains examples from the first dataset, the other 3 coming from the test dataset. First row contains raw slices, for which the grayscale image was produced by normalizing without truncating first. The second row contains slices normalized after they have been truncated, showing the similarity

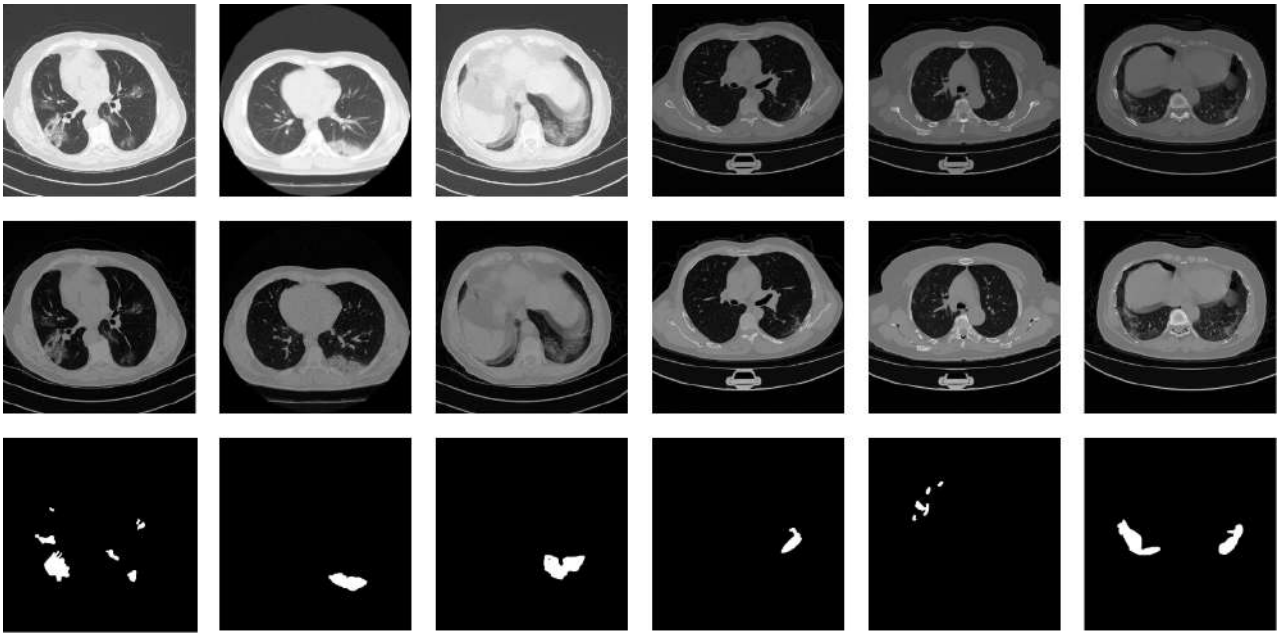


Figure 2.2: Examples of raw and processed CT slices

between preprocessed slices, even for significant differences between raw slices coming from the 2 different datasets. The third row represents the ground-truth segmentation for all lesions in each particular slice, which will be further used as target outputs for the proposed architectures.

Chapter 3
Proposed Solutions

3.1 Architecture 1

The first proposed architecture is presented in Figure 3.1. This architecture has 2 inputs, namely the pre-processed CT scan slice and the same slice on which *histogram equalization* was applied. The highlited areas from the figure – Module 1 and Module 2 – are to be referenced in the Results and Discussion section for comparison studies.

The architecture is organized as follows:

- An *encoder* part with 4 parallel branches that extracts information from the 2 given inputs.
- 2 feature extractor blocks that operate on the feature maps obtained from the encoder part.
- A *decoder* part which up-samples the resulted features to obtain the segmentation map.
- Skip connections between symmetric encoding blocks that receive the unaltered CT slice as input and the corresponding decoding blocks.

Histogram equalization was applied in order to increase the global contrast of the input CT slice, enhancing poorly visible areas of different lesions. Consider the case of a grayscale image $X^{(i)} \in \mathbb{R}^{512\times512}$ which represents the input CT slice, and $P(X = i) = p_i$, $i \in \{0, 1, \ldots, 255\}$ the frequency (probability) of the grayscale level i in the current slice. Histogram equalization is performed as stated below:

1. Calculate the cumulative distribution function of the grayscale image X:

$$CDF_X(k) = \sum_{i=0}^{k} p_i, \quad k \in \{0, 1, \ldots 255\} \tag{3.1}$$

2. Apply the transformation defined by CDF_X on the input CT slice (the input CT slice with grayscale values) to obtain the slice with equalized histogram:

$$X_{eh}^{(i)}[l, c] = CDF_X(X^{(i)}[l, c]), \quad (l, c) \in \{0, 1, \ldots, 255\}^2 \tag{3.2}$$

The obtained $X_{eh}^{(i)}$ will be further used as input, without any additional processing.

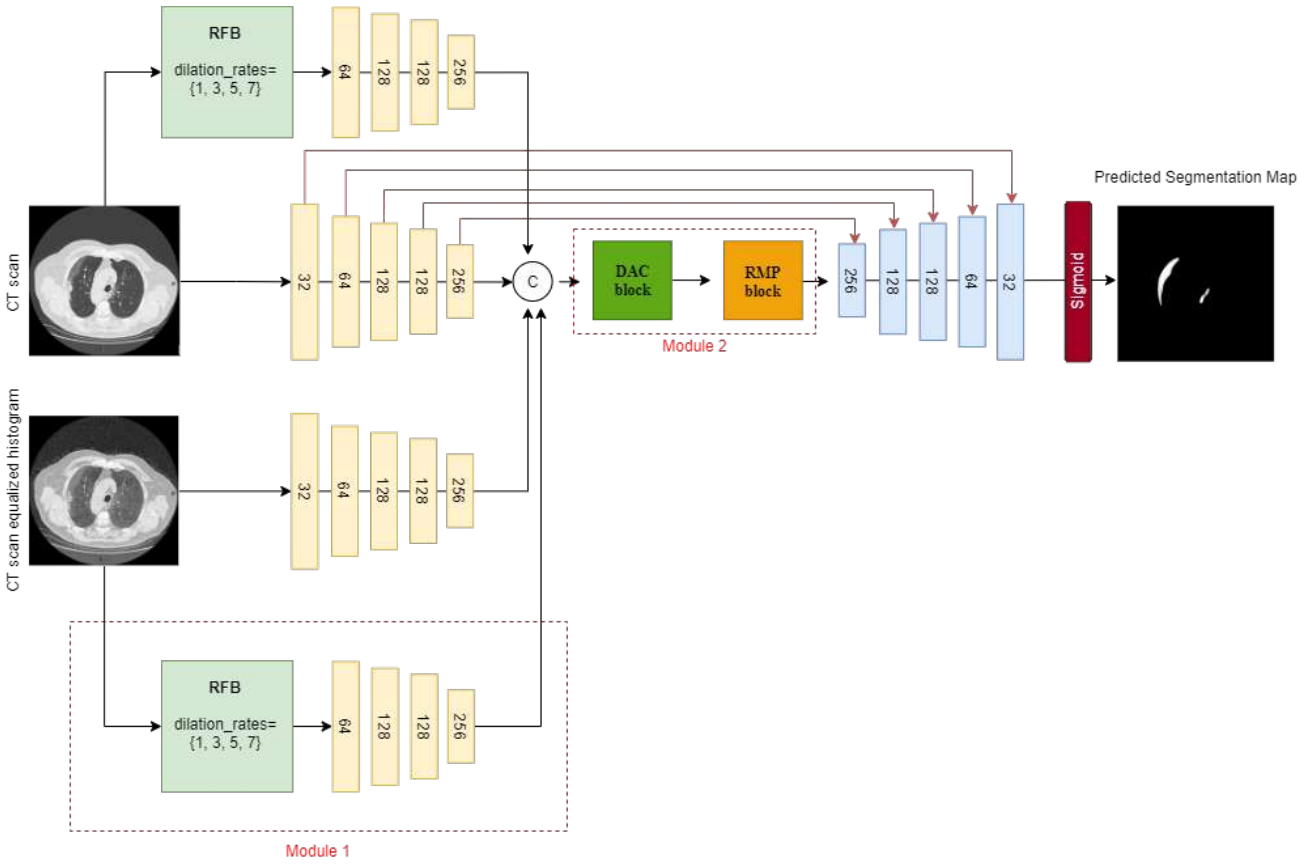


Figure 3.1: First proposed architecture (Architecture 1)

The encoder part of the architecture contains 4 parallel branches that extract information from the 2 inputs using structures of the form Convolution2D-ReLU-BatchNormalization-MaxPooling2D (yellow blocks in Figure 3.1). Each branch contains 5 such structures (with exception the RFB branches, for which the MaxPooling2D operation is performed immediately after), thus resulting in a final feature map of size $(16, 16, 256)$ for each branch. To reduce the number of channels for the concatenated feature maps, an 1×1 convolution with 512 3×3 filters is applied before any further computation. All convolution operations in the encoding blocks use filters of size 3×3, with an increasing number of filters towards the final down-sampling operation. The 4 resulted feature maps are then concatenated along the channel dimension, to be further passed through feature extractor blocks from Module 2.

Inspired by the Receptive Field Block (RFB) architecture in [27], which has been used to obtain fast and powerful object detectors, a similar structure is applied on 2 separate branches from the encoder part, separately on the input CT slice and the equalized histogram version of it. This structure consists of multiple parallel convolutions with different dilation rates which are used to extract information regarding different size areas from the input slices, while maintaining the same number of parameters regarding the area of the receptive field.

The RFB structure from Architecture 1 uses 3 different dilated convolutions, on 3 different branches, with dilation rates 3, 5, and 7. Before applying the dilated convolution operation, for which filters of size 3×3 were used regarding the dilation rate, the authors of [27] perform a 1×1 convolution to obtain the desired channel dimension. This is followed by 2 subsequent convolutions, operating first with filters of size $(1, \textit{dilation\_rate})$, and then with filters of size $(\textit{dilation\_rate}, 1)$. Before each convolution operation, a *zero padding* was used to ensure the same first 2 dimensions on the output feature map were preserved.

Module 2 consists of two succesive feature extractor blocks, namely the Dense Atrous

Convolution block (DAC) and Residual Multi-kernel Pooling block (RMP), introduced in [28] as part of a medical image segmentation architecture. The DAC block consists of 4 parallel branches which process the input feature map using standard 3×3 convolutions, followed by convolutions with different dilation rates which extract information from the high-level concatenated feature-maps regarding different size-objects, as shown in Figure 3.2.

The RMP block contains 3 branches with MaxPooling operation with different pooling sizes (2, 4 and 6) which operate on the feature volume obtained from the DAC block, followed by an 1×1 convolution with 1 filter, as shown in Figure 3.3. Each branch extracts information using different size receptive fields, thus encoding the $(16 \times 16 \times 512)$ input feature volume in 3 different maps with a single channel each, using different encoding scales. These maps are then up-sampled to the initial feature map resolution and concatenated along channel dimension with the input volume.

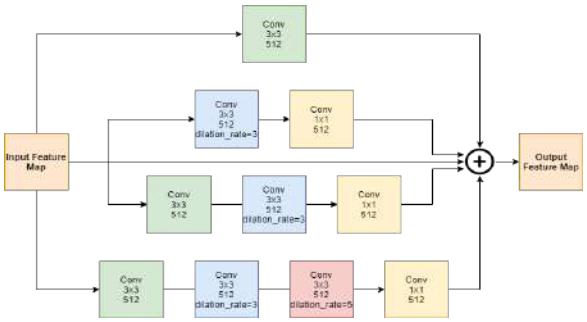


Figure 3.2: Dense Atrous Convolution block[28]

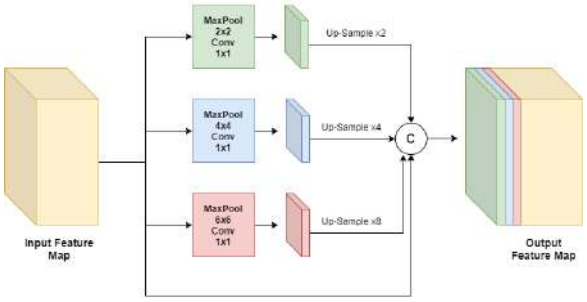


Figure 3.3: Residual Multi-kernel Pooling block

The up-sampling process in the RMP block uses *bilinear interpolation* to achieve the desired resolution. Consider the case of estimating the value of a 2-dimensional function f at some point (x, y) using the value of 4 neighbour points arranged on a square grid: $P_{11}(x_1, y_1)$, $P_{12}(x_1, y_2)$, $P_{21}(x_2, y_1)$, $P_{22}(x_2, y_2)$, with the assumption that $x_1 < x < x_2$ and $y_1 < y < y_2$. The value $f(x, y)$ could be therefore estimated using bilinear interpolation as follows:

1. Estimate the values of $f(x, y_1)$ and $f(x, y_2)$ using linear interpolation along the x-axis:

$$f(x, y_1) \approx \frac{x - x_1}{x_2 - x_1} f(P_{21}) + \frac{x_2 - x}{x_2 - x_1} f(P_{11}) \tag{3.3}$$

$$f(x, y_2) \approx \frac{x - x_1}{x_2 - x_1} f(P_{22}) + \frac{x_2 - x}{x_2 - x_1} f(P_{12}) \tag{3.4}$$

2. Estimate the value of $f(x, y)$ by linear interpolation along the y-dimension using the previously computed values $f(x, y_1)$ and $f(x, y_2)$:

$$f(x, y) \approx \frac{y - y_1}{y_2 - y_1} f(x, y_2) + \frac{y_2 - y}{y_2 - y_1} f(x, y_1) \tag{3.5}$$

The decoder part of Architecture 1 consists of 5 structures of the the form Convolution2D-ReLU-BatchNormalization-UpSampling2D (the blue blocks from Figure 3.1), using bilinear interpolation to up-sample each feature map by a factor of 2 on the first 2 dimensions. Before passing the up-sampled feature map to these structures, *skip connections* are added between these feature maps and the symmetric feature maps obtained in the encoding branch for the original CT scan, which are concatenated along channel dimension. This helps with the information loss during the down-sampling operations in the encoding branch, and further

processing implied by the Module 2 blocks. Thus, combining early features with much more complex ones helps with retrieving some of the information that is lost during the encoding phase. A final 1×1 convolution with 1 filter, followed by a Sigmoid activation is applied to obtain the predicted segmentation map.

In the Results and Discussion section experiments were performed starting from a simplified version of Architecture 1, without Module 1 and Module 2, following successive modifications and finally arriving at the performances on the full architecture, as presented in Figure 3.1. This architecture along with similar experiments were presented in the proceedings of *IEEE* 18^{th} *International Symposium on Biomedical Imaging (ISBI)* as a conference paper [33].

3.2 Architecture 2

The second proposed architecture is shown in Figure 3.4. This architecture represents an image-to-image translation GAN framework, for which two different neural networks are built: a Discriminator and a Generator.

The Generator network consists of a fully convolutional neural network, with 2 inputs – the CT scan and the CT scan with equalized histogram – from which features are extracted using 2 separate branches. Each branch consists of 4 Convolution2D-BatchNormalization-ReLU-MaxPooling2D structures (yellow blocks from Figure 3.4), with a down-size factor of 2 for all MaxPooling operations, and 3×3 filters for all convolution layers. After the final MaxPooling operation, the two resulted feature maps are concatenated along channel dimension, and are further passed to a series of 4 Convolution2D-BatchNormalization-ReLU-UpSampling2D structures – blue blocks from Figure 3.4 – using bilinear interpolation for the up-sampling operation with a factor of 2. Finally, an 1×1 convolution with 1 filter is applied, along with a Sigmoid activation, to obtain the predicted segmentation map. Similar to Architecture 1 (Figure 3.1), skip connections are added from down-sampling blocks to symmetric up-samplig blocks, concatenating the features extracted from the original CT scan with the more complex up-sampled features in order to reduce the information loss during the down-sampling process.

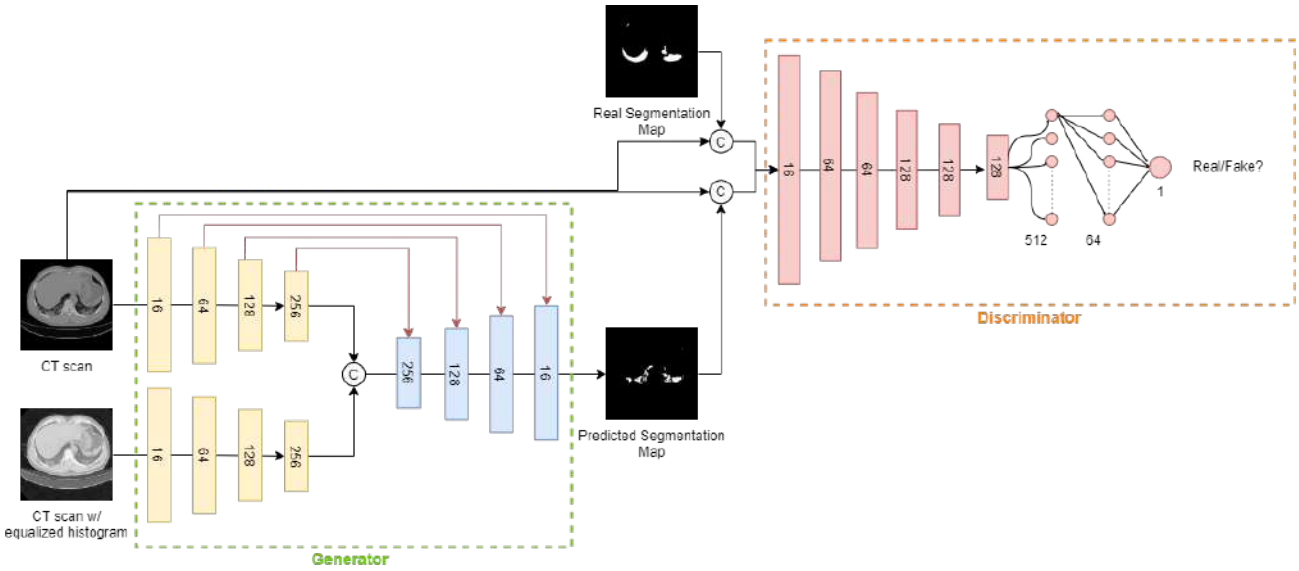


Figure 3.4: Second proposed architecture (Architecture 2)

The Discriminator network receives as input either the CT scan concatenated channel-wise with the real segmentation map, or the CT scan concatenated channel-wise with the predicted segmentation map. The Discriminator is trained such that it can differentiate between these 2 types of input, which will be referred to as "real" and "fake". The Discriminator consists

of 6 Convolution2D-BatchNormalization-ReLU-MaxPooling2D structures, with a down-sizing factor of 2, and 3×3 filter sizes for all convolution layers, followed by two dense hidden layers with 512 and 64 neurons and ReLU activation in-between, and a final layer which contains only 1 neuron with Sigmoid activation. The network is trained in such fashion that "real" input data will generate an output closer to 1 on the final neuron, while "fake" examples will result in outputs closer to 0.

GANs are known for their chaotic behaviour during training, at least in the early epochs. Different constraint techniques such as weight clipping or gradient clipping could help with stability during the training process. In this work, **spectral normalization** was applied for the convolution and dense layers of the Discriminator and Generator in order to impose a maximum singular value of 1 for the kernel defined by each layer. In [29] the authors propose using spectral normalization to stabilize the training of the Discriminator network, concluding that this type of weight normalization results in a system that performs better than with usual weight constraint techniques.

Spectral normalization works by limiting the maximum singular value of a weight matrix to 1. In the case of a dense layer with wight matrix $W \in \mathbb{R}^{m \times n}$ the resulted normalized matrix is computed as $W_{normalized} = \frac{W}{||W||_S}$, where $|| \ W \ ||_S$ represents the maximum singular value of matrix W, also called the *spectral norm*. With this mechanism the weight matrix is re-normalized with the new spectral norm after each update, to ensure a maximum singular value of 1. In the case of Conv2D, for which the weights are organized in a volume of shape $(k, k, ch_{in}, ch_{out})$, where $k \times k$ is the filter size, ch_{in} is the number of channels for the input volume, and ch_{out} is the number of resulted output channels, a.k.a the number of filters in the current convolution layer, the matrix W for which the spectral norm is calculated is the weight volume reshaped to $(k \times k \times ch_{in}, ch_{out})$. Calculating the exact value of spectral norm for all layers would be inefficient, resulting in a significant increase in the training time, which is why an iterative algorithm called *power iteration* is more commonly used for estimating the maximum singular value. To be noted that for a $M \times N$ matrix A the singular values are the square roots of the eigenvalues of the matrix $B = A^*A$, where A^* is the transpose conjugate matrix of A.

For this work, spectral normalization was performed using layers implemented in *deel-lip* library [30].

3.3 Experiments and Discussion

In this section, experiments with the two proposed architectures are conducted, along with some post-processing proposals for the results obtained from the second architecture.

Architecture 1 (Figure 3.1) was trained using *Adam* optimizer (described in 1.1) in order to minimize the *Dice coefficient loss*. If the real segmentation map is denoted by $Y^{(i)} \in \{0, 1\}^{512 \times 512}$ and the predicted segmentation map by $\hat{Y}^{(i)} \in [0, 1]^{512 \times 512}$ the loss function is defined as follows:

$$\mathcal{L}_{Dice} = \frac{1}{N_b} \sum_{i=1}^{N_b} 1 - DSC(Y^{(i)}, \hat{Y}^{(i)}) \tag{3.6}$$

$$DSC(Y^{(i)}, \hat{Y}^{(i)}) = \frac{2|Y^{(i)} \cdot \hat{Y}^{(i)}|}{|(Y^{(i)})^2| + |(\hat{Y}^{(i)})^2| + \epsilon} \tag{3.7}$$

Here the $(\cdot)^2$ operation is applied element-wise, the $|\cdot|$ denotes the reduction sum operator and ϵ is a small constant to prevent division by 0. The DSC has a maximum value of 1 only if the two arguments are equal, i.e. $Y^{(i)}[l, c] = \hat{Y}^{(i)}[l, c], \quad 0 \leq l, c \leq 511$. Thus, the network optimizes its parameters in order to produce segmentation maps similar to the real ones, relative to the

Dice similarity score.

For Architecture 1 comparative studies were conducted by first adding Module 1 to the base architecture, followed by Module 2 and skip connections between encoding blocks and decoding blocks.

In Table 3.1 the performances of 3 variations of Architecture 1 on the validation set are presented. It can be seen that adding the RFB branch for the CT scan with equalized histogram (Module 1) significantly increases the mean Dice coefficient score and the mean Sensitivity. Adding further skip connections and the 2 feature extractor blocks (Module 2) results in the highest Dice score, and also the highest mean Precision on the validation set. Additionally, two other previously introduced architectures were trained and evaluated on the same training and validation sets for comparison, namely the U-Net [17] and Inf-Net [22]. The architectures were trained using Adam optimizer to minimize the Dice coefficient loss (Equation 3.6). It can be seen from Table 3.1 that the final proposed architecture obtains a higher mean Dice score and a higher mean Sensitivity on the validation set.

	DSC	**Sensitivity**	**Specificity**	**Precision**
U-Net	0.492	0.456	0.996	0.657
Inf-Net	0.664	0.674	0.997	0.789
A1	0.515	0.495	0.995	0.564
A1+M1	0.665	0.802	0.996	0.608
A1+M1+M2+skip.con	0.707	0.803	0.996	0.671

Table 3.1: Performance of Architecture 1 (A1) on the validation set; M1 - Module 1; M2 - Module 2

The architecture that achieved the best Dice similarity score on the validation set was further used for the testing phase, along with both the pre-trained U-Net and Inf-Net architectures. Table 3.2 illustrates that the proposed architecture achieves the best Dice similarity score and Sensitivity on the test set, leading to a good generalization capability, even when performing on much larger datasets, relative to the training set.

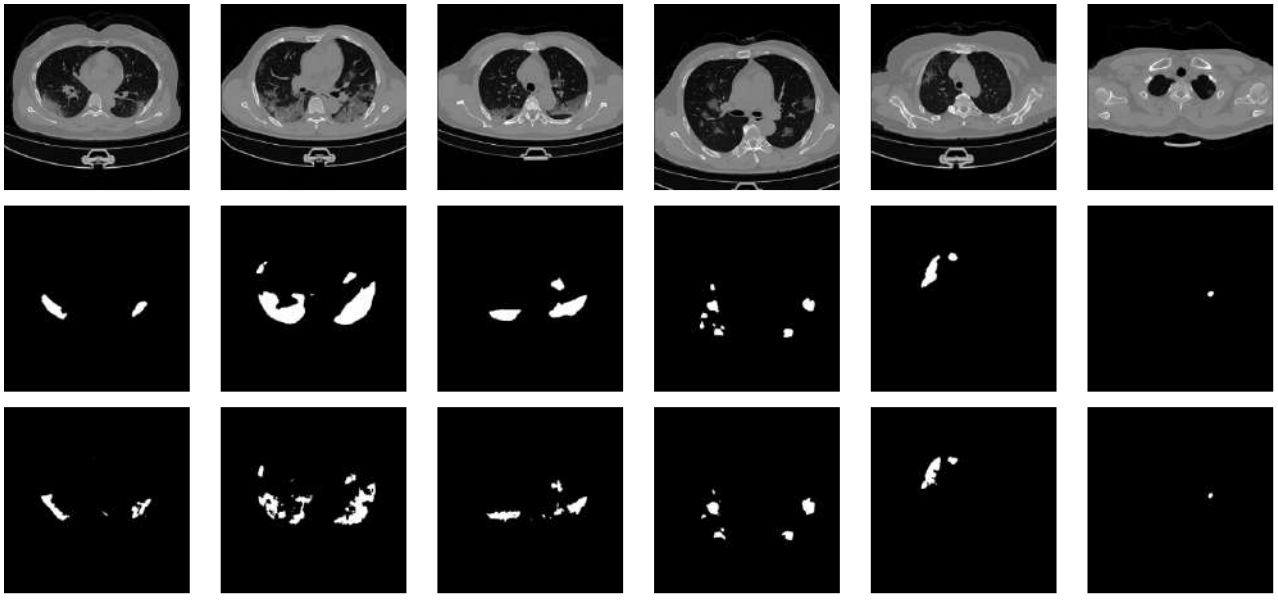


Figure 3.5: Experimental results on the test set for Architecture 1; First row – 2D CT slices; Second row – ground-truth segmentation maps; Third row – predicted segmentation masks

In Figure 3.5 a few experimental results from the test set are presented, in which can be

seen that the trained architecture can capture most of the lesion areas, for various 2D CT slice structures, even for masks that only contain a few pixels of lesions.

	DSC	**Sensitivity**	**Specificity**	**Precision**
U-Net	0.446	0.416	0.998	0.696
Inf-Net	0.428	0.431	0.997	0.634
A1+M1+M2+skip.con	0.515	0.525	0.996	0.685

Table 3.2: Performance comparison on test set

Architecture 2 (Figure 3.4) was trained using *AdaGrad* optimizer (described in 1.1), using a different optimizer instance for each of the two networks, the Discriminator and the Generator.

The Discriminator was trained such that it could differentiate between the two input classes: the one that contained a real segmentation mask vs. the one with a generated mask. The Generator was trained such that the predicted segmentation mask would be as close as possible to the real segmentation mask and that it would also fool the Discriminator network in thinking that this predicted mask would represent a real one when applied at its input, thus maximizing the error of the Discriminator. Initial experiments showed that the Discriminator was able to learn its task much easier than the Generator network, thus resulting in a big error term for the Generator – which has led to a chaotic behaviour due to the error signal backpropagated to Generators parameters – caused by the overconfidence developed in the Discriminator network. Since it is of interest to balance the two networks such that they do not overpower each other, and thus a stable training for the generator, several techniques are applied in order to balance the training process:

- *Label smoothing* was applied in order to reduce the confidence of the Discriminator network regarding which type of data it is presented with (for a detailed discussion please refer to [32]). This technique implies using *soft targets* as guiding signals for the Discriminator output, of the form:

$$y' = (1 - \alpha)y + \frac{\alpha}{K} \tag{3.8}$$

  where $\alpha \in [0, 1]$ is the smoothing parameter and K is the number of classes, which in this case is equal to 2 (real vs. generated). For all further experiments α was set to 0.4, which resulted in the soft labels $y_{real} = 0.8$ (instead of 1) and $y_{fake} = 0.2$ (instead of 0).

- *Spectral normalization* was applied for all convolution and dense layers from the Generator and Discriminator. Adding this type of weight constraint significantly increased the stability of the Generator network during the training process.

- Some works on GANs advise for using more discriminator updates per batch, followed by a single update for the generator, in order to keep the system balanced. Initial experiments showed that for a single update for each network resulted in a discriminator that would learn to differentiate between the two types of input much better than the Generator was able to produce relevant segmentation maps. Starting from this, and also from the fact that the Generators mean Dice on the validation and training set was relative low, it was assumed that the Generator could not produce segmentation masks good enough (in terms of Dice score between predicted and real masks) to fool the Discriminator, thus resulting in a big error term that the Generator was focused on during the optimization process (instead of a balanced focusing on the Dice loss and the error term from the Discriminator). This issue motivated using more Generator updates per batch of data.

Subsequent experiments showed that for 3 Generator updates, after a single Discriminator update resulted in a relative stable system during the training process. This strategy was adopted for all further experiments.

As presented earlier, the Discriminator is trained to classify between the two types of input, while the Generator is trained to maximize the error of the Discriminator when presented with a generated example, and also to minimize a local term, in this case the Dice similarity loss between the predicted and the real segmentation mask. Consider the functions defined by the two networks (at some point during the training process), $G(x) : [0,1]^{512\times512\times2} \rightarrow [0,1]^{512\times512}$ – the Generator – and $D(x) : [0,1]^{512\times512\times2} \rightarrow [0,1]$ – the Discriminator. Each input $X^{(i)} \in [0,1]^{512\times512\times2}$ of the Generator is represented by the CT scan slice concatenated along channel dimension with its equalized histogram version, while the input for the Discriminator is represented by the CT scan slice concatenated with either the real ($X^{(i)}_{real}$) or generated segmentation map. Considering this, the loss functions associated to these two networks are defined as follows:

$$\mathcal{L}_D = \frac{1}{B}\sum_{i=1}^{B}(0.8 - D(X^{(i)}_{real}))^2 + (0.2 - D(G(X^{(i)})))^2 \tag{3.9}$$

$$\mathcal{L}_G = \frac{1}{B}\sum_{i=1}^{B}\underbrace{(1 - DSC(Y^{(i)}, G(X^{(i)})))}_{\text{local error}} + \underbrace{(0.8 - D(G(X^{(i)})))^2}_{\text{discriminator error maximization}} \tag{3.10}$$

where DSC is defined in Equation 3.7 and B is the batch size on which each update is performed. Minimizing the loss in Equation 3.9 reduces to minimizing each squared term, that is producing outputs closer to 0.8 for real inputs, and close to 0.2 for generated inputs. Equation 3.10 presents the loss used for the Generator network, for which the minimization process consists of minimizing the local error term (i.e. maximizing the Dice similarity score) and minimizing the second term, which consists in maximizing the Discriminator update when it is presented with a generated example.

Following the training process, the Generator network was taken separately for the main task of producing the segmentation mask, in this case images of size 512×512 with values in the range $[0, 1]$, due to the final *Sigmoid* activation. Since the real segmentation masks were represented by binary images – with values of only 0 and 1 – a *segmentation threshold* needs to be applied on the predicted mask, in order to further compare these 2 masks. This threshold was set to 0.5 for all further experiments.

A first experiment was conducted to see if the performance of this architecture reaches better performance than the same Generator trained alone, outside the adversarial setting. For this purpose, the same Generator architecture was trained alone, using Dice similarity loss (Equation 3.6) and *AdaGrad* optimizer with a learning rate of 10^{-2}. The performances on the validation and test set are presented in Table 3.3. It can be seen that the Generator architecture trained in an adversarial setting produces much higher mean Dice scores on both validation and test set, along with a significant increase in Sensitivity. The evolution of Dice similarity score and the loss function during the training process for these 2 architectures can be visualized in Figure 3.6, for a number of 2000 epochs. It can be seen that the Generator trained alone exhibits a more chaotic behaviour, especially during the initial epochs. The DSC for both architectures converges after 2000 epochs, however the loss function for the proposed GAN architecture still decreases, which may signify that the Generator continues to optimise its loss through the minimization of the second term from Eqaution 3.10 (maximization of the Discriminator error).

However, the Precision decreases, which implies that besides most of the lesions presented

	DSC		Sensitivity		Specificity		Precision	
	Val	Test	Val	Test	Val	Test	Val	Test
Generator	0.338	0.497	0.354	0.463	0.996	0.996	0.456	0.662
Generator + Discriminator (A2)	0.531	0.538	0.685	0.661	0.994	0.993	0.472	0.511

Table 3.3: Performance comparison for A2 and the Generator trained separately

in the real image, Architecture 2 additionally mispredicts other lesion spots, more than the Generator alone does. This issue is further discussed in the next paragraph.

Thresholding the predicted masks resulted in multiple small regions of maximum value, besides the important bigger objects in the image, which resemble a real mask on which impulsive noise of maximum value was applied. Additionally, some of the convex regions that represent lesion areas contained more or less discontinuities, after observing side-by-side the predicted and real segmentation mask. With the objective of mitigating some of these issues, two types of post-processing techniques are applied, as follows:

- Adaptive smoothing using a **Bilateral filter**:

$$A_f(x,y) = \frac{1}{F_n} \sum_{(i,j)\in V_{xy}} A(x+i, y+j) exp\left(-\frac{i^2+j^2}{2\sigma_d^2} - \frac{(A(x,y) - A(x+i,y+j))^2}{2\sigma_i^2}\right) \tag{3.11}$$

$$F_n = \sum_{(i,j)\in V_{xy}} exp\left(-\frac{i^2+j^2}{2\sigma_d^2} - \frac{(A(x,y) - A(x+i,y+j))^2}{2\sigma_i^2}\right) \tag{3.12}$$

  where A, A_f are the input image, filtered image respectively, and V_{xy} represents the set of indices relative to a center point, forming its neighbourhood – on which the filtering process is performed. This type of smoothing takes into account the Euclidean distance between the center pixel and each neighbour, along with the difference between their intensities to calculate the smoothing weight for each point. This type of filtering results in small weights for neighbours which are more distanced from the center of the filtering window and/or more different in terms of pixel intensity than the center pixel. The F_n term (Equation 3.12) is used as a normalising value, to ensure that the coefficients for each filtering window sum to 1, a necessary condition for any smoothing filter. σ_d and σ_i are the only parameters that control the behaviour of this filter, specifically the tolerance for higher distances and bigger intensity differences. Increasing the value of this parameters would result in bigger coefficient terms for pixels that would normally have lower contributions in the filtering process due to either high distances or different intensities.

- **Mathematical Morphology** operations, which consists of comparing in different ways an image to be processed with a much smaller one called *structuring element*. A detailed discussion regarding this type of operations can be found in [31]. Following, the basic morphologic operations are presented, where Z^2 denotes the structuring element and Z^1 denotes the image to be processed:

  - *Erosion* : $Z^1 \ominus Z^2 = \{\mathbf{z} | Z^2_{\mathbf{z}} \subseteq Z^1\}$ – $Z^2_{\mathbf{z}}$ represents the translation of the structuring element Z^2 at the position of subset $\mathbf{z}$ in the image Z^1. The above definition implies that the processed image will contain pixels with a value of 1 for which if the structuring element is placed on it will be fully included in the objects of the initial image. The main effect of this operation is a dimension reduction for the initial objects. An example of erosion can be visualized in Figure 3.7, where a

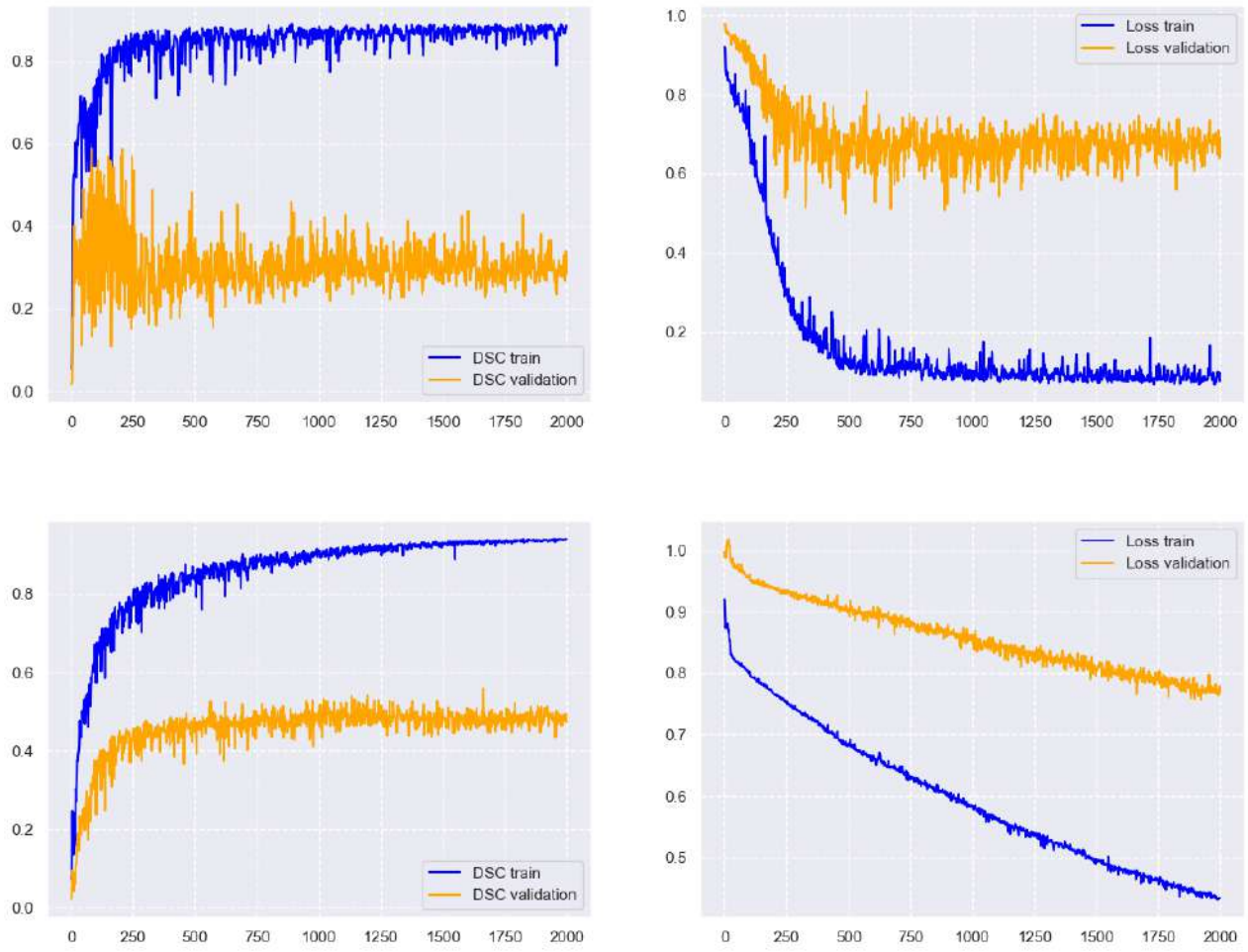


Figure 3.6: The evolution of Dice similarity score (DSC) and loss value of Generator during the training, on both training and validation set; First row – evolution for Generator trained alone; Second row – evolution for the proposed GAN architecture

structuring element of size 3×3 was used, resulting in the disappearance of all pixels outside the bigger object, along with the accentuation of empty spots present in the object.

- *Dilation* : $Z^1 \oplus Z^2 = \{\mathbf{z} | Z^2_{\mathbf{z}} \cap Z^1 \neq \emptyset\}$ – the processed image will contain pixels with a value of 1 for which if the structuring element is placed on it will have at least 1 common pixel with the objects in the initial image. The main effect of this operation is a dimension increase for the initial objects. An example of dilation can be visualized in Figure 3.7 where the added impulsive noise is amplified, along with filling the empty spots present in the object.
- *Opening*: $(Z^1 \ominus Z^2) \oplus Z^2$ – this operation can be used if the image to be processed is affected by impulsive noise (with a maximum value of 1), thus using the first erosion operation eliminates most of the small objects (comparable in size with the structuring element Z^2) which are considered noise, and the following dilation operation acts as a corrector by increasing the size of the bigger objects (which are of interest) which have been affected by the first erosion.
- *Closing*: $(Z^1 \oplus Z^2) \ominus Z^2$ – this operation can be used if the objects present in the image to be processed contain empty spots (due to unwanted sources), thus using

the first dilation operation fills most of the empty spots present (comparable in size with the structuring element Z^2), and the following erosion resizes the dilated objects to approximately their original size and reduces the unwanted effects of the first dilation operation, that is the amplification of possible noise spots.

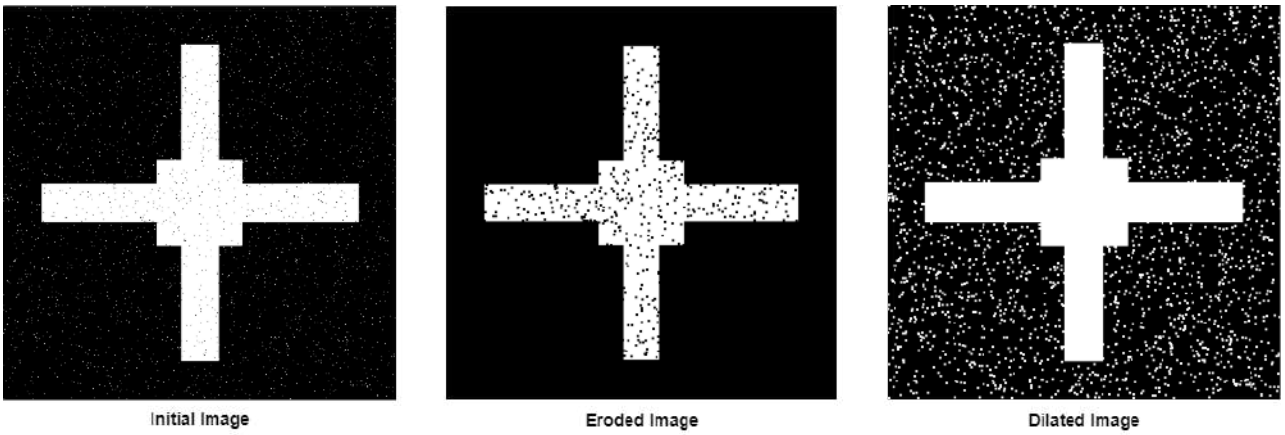


Figure 3.7: Example of Dilation and Erosion with a structuring element of size 3×3

The performances for this proposed architecture along with the post-processing operations discussed above are presented in Table 3.4. The operation of smoothing using the Bilateral filter was applied on the activation map after the final Sigmoid activation, with pixel intensities in the range $[0, 1]$, for which a filtering window of size 9×9 was used, along with parameters $\sigma_d = \sigma_i = 8$, after which the same threshold was applied to obtain a binary image. The morphological operations were applied on the thresholded segmentation map, using square structuring elements. From Table 3.4 it can be seen that this architecture along with an opening operation, followed by a closing operation with a structuring element of size 7×7 achieves the highest Dice similarity score on the validation set, and also a significant increase in the Precision, compared with the raw predicted masks. Also, using the smoothing with Bilateral filter achieves the highest Sensitivity on the validation set. In Figure 3.8 a few experimental results obtained from the test set can be visualized, along with the effectiveness of using the described post-processing operation.

	DSC	**Sensitivity**	**Specificity**	**Precision**
A2	0.531	0.685	0.994	0.472
A2+B.F.	0.553	0.722	0.993	0.486
A2+Op_9	0.561	0.605	0.996	0.592
A2+Op_7	0.567	0.639	0.995	0.565
A2+Op_7+Cls_7	0.570	0.649	0.996	0.566

Table 3.4: Performance results for Architecture 2 (A2) and post-processing techniques, on the validation set; B.F. - bilateral filter; Op_N/Cls_N - morphological opening/closing using a N $\times$ N structuring element

The third row in Figure 3.8 contains the predicted segmentation maps, while the fourth presents the predicted mask on which morphological opening followed by morphological closing was applied, using a structuring element of size 7×7. It can be seen that most of the noisy areas resulted after applying the segmentation threshold disappeared (the areas highlighted with red).

Table 3.5 presents the overall performance measures for the presented architectures, on both validation and test set. The first architecture (containing all previously introduced modules –

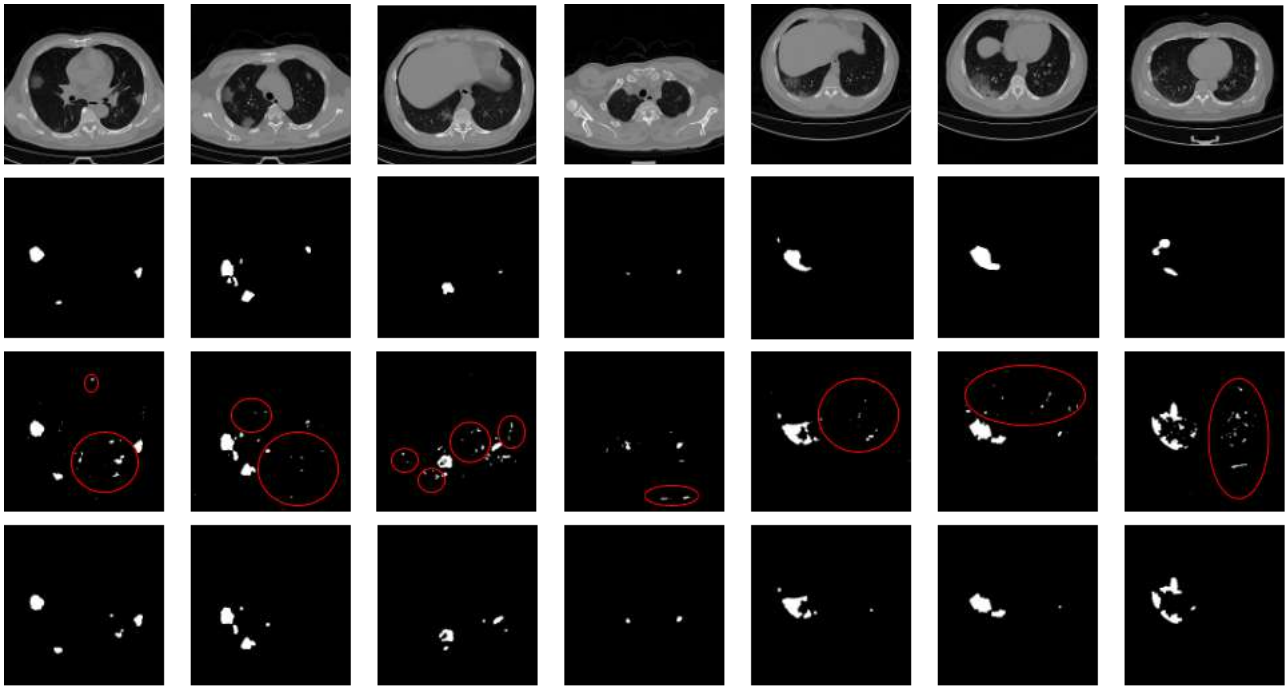


Figure 3.8: Experimental results on the test set for Architecture 2; First row – 2D CT slices; Second row – ground-truth segmentation maps; Third row – predicted segmentation masks, on which areas with noise were Highlighted; Fourth row – post-processed masks with morphological opening, followed by closing;

Figure 3.1) obtains the maximum Dice similarity score and the maximum Sensitivity on the validation set, along with second best Precision score on both validation and test set. The second architecture (presented in Figure 3.4) along with the post-processing operation using morphological opening followed by morphological closing obtains the highest Dice similarity score on the test set, the second best being atributed to the same architecture along with Bilateral filter post-processing, which additionally obtains the best Sensitivity on the test set. The Precision score, however, is higher for the two well-known architectures, U-Net and Inf-Net. This fact alone implies that the 2 architectures produce segmentation maps with lesion areas most likely contained in the same regions as in the ground-truth segmentation maps, but adding the low Sensitivity indicates that a relatively high number of lesion spots are omitted.

	DSC		**Sensitivity**		**Specificity**		**Precision**	
	Val	Test	Val	Test	Val	Test	Val	Test
U-Net	0.492	0.446	0.456	0.416	0.996	0.998	0.657	**0.696**
Inf-Net	**0.664**	0.428	0.674	0.431	0.997	0.997	**0.789**	0.634
A1+M1+M2+skip.conn	**0.707**	0.516	**0.803**	0.525	0.996	0.998	**0.671**	**0.685**
Generator	0.338	0.497	0.354	0.463	0.996	0.996	0.456	0.662
Generator + Discriminator (A2)	0.531	0.538	0.685	**0.661**	0.994	0.993	0.472	0.511
A2 + B.F.	0.553	**0.556**	**0.722**	**0.693**	0.993	0.993	0.486	0.523
A2 + Op_7 + Cls_7	0.570	**0.562**	0.649	0.593	0.995	0.995	0.565	0.618

Table 3.5: Performance comparison on the validation and test set for all proposed architectures

Conclusions

Final Remarks

In this project, two different approaches for automatic lung lesion segmentation were presented. First one implies a neural network architecture comprised of an encoder-decoder structure and different feature extracting blocks. The second one uses a much simpler architecture for the actual structure that produces the desired outputs, but is integrated in an adversarial setting by using another neural network which acts as a discriminator during the training process, providing useful feedback for the generator structure.

One of the main limitations regarding medical data in the context of training a neural network is the lack of annotated samples, in this case a shortage of annotated segmentation masks. This limitation motivated a training setup that implied using a relative small training set, compared to the data that the model was tested on afterwards. Thus, the generalization capability of the final models was one of the main objectives for this project.

Following the discussion in Chapter 3.3, both approaches achieved satisfactory results when tested on a much bigger dataset, proving good generalization capabilities. Furthermore, some post-processing techniques were also applied on the outputs produced by the second architecture, smoothing the predicted segmentation masks and resulting in a higher performance on both validation and test set.

Personal Contributions

The personal contributions to this project can be summarized as follows:

- Pre-processing the raw CT volumes, as described in Chapter 2.3, using Python programming along with some scientific packages such as `Numpy`, `Scipy`, `OpenCV`
- Research on existing methods for lung lesion segmentation neural networks and different deep learning paradigms for medical image segmentation tasks
- Designing and implementing the proposed architectures, as described in Chapter 3, using Tensorflow 2.0 and `deel-lip` library [30]
- Setting up the training process for the proposed architectures: choosing the corresponding loss functions, observing the models performance over epochs, reiterating between training and hyperparameter tuning (especially during training of the second proposed architecture, which exhibited a much more chaotic behaviour in the initial phases), assessing the performance of trained models on the pre-processed datasets.

- Implementing and training two existing architectures for image segmentation, using the same training setup for performance comparison
- Choosing and implementing the right post-processing techniques for the second models predicted masks, with the objective of eliminating different sources of noise related to image thresholding, obtaining better results in terms of Dice score and Sensitivity

The source code for model architectures along with other functionalities can be found at `https://github.com/Vladimirescu/Lung_lesion_segmentation`.

Future Work

Several emerging areas related to medical imaging, including automatic segmentation of anatomical regions of interest, have been recently taken on by the advances of artificial neural networks, mostly due to their high representation capability for complex input data. A lot of research areas related to lung lesion segmentation can be addressed following this project, including, but not limited to, the following:

- Integrating the architectures in a semi-supervised setting, for which a small set of labeled samples are combined with a much larger set of unlabeled examples, providing a solution for the shortage of annotated medical data
- Training and testing different GAN architectures
- Designing new methods for training Generative Adversarial Networks such as to increase the stability during the training process, without exhibiting too much underfitting behaviour
- Designing architectures that act directly on the 3D volume of CT scan, instead of extracted 2D sections
- Correlating the results between predicted masks for CT slices obtained from the same patient, such as the combined masks form a relevant 3D volume of lesions
- Design methods for classifying the predicted lesion areas into multiple types of lesions, either by using two models – one for segmentation and one for classification, either by combining both tasks into one model

References

[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[3] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic gradient descent," in *ICLR: International Conference on Learning Representations*, pp. 1–15, 2015.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[5] Y. Tai, J. Yang, and X. Liu, "Image super-resolution via deep recursive residual network," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3147–3155, 2017.

[6] 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 arXiv:1207.0580*, 2012.

[7] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323, JMLR Workshop and Conference Proceedings, 2011.

[8] L. Lu, Y. Shin, Y. Su, and G. E. Karniadakis, "Dying relu and initialization: Theory and numerical examples," *arXiv preprint arXiv:1903.06733*, 2019.

[9] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *arXiv preprint arXiv:1406.2661*, 2014.

[10] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[11] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[12] Y. Yu, Z. Gong, P. Zhong, and J. Shan, "Unsupervised representation learning with deep convolutional neural network for remote sensing images," in *International Conference on Image and Graphics*, pp. 97–108, Springer, 2017.

[13] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*, pp. 448–456, PMLR, 2015.

[14] H. Wu, J. Zhang, K. Huang, K. Liang, and Y. Yu, "Fastfcn: Rethinking dilated convolution in the backbone for semantic segmentation," *arXiv preprint arXiv:1903.11816*, 2019.

[15] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," *arXiv preprint arXiv:1511.07122*, 2015.

[16] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1125–1134, 2017.

[17] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241, Springer, 2015.

[18] Z. Zhou, M. M. R. Siddiquee, N. Tajbakhsh, and J. Liang, "Unet++: A nested u-net architecture for medical image segmentation," in *Deep learning in medical image analysis and multimodal learning for clinical decision support*, pp. 3–11, Springer, 2018.

[19] O. Oktay, J. Schlemper, L. L. Folgoc, M. Lee, M. Heinrich, K. Misawa, K. Mori, S. McDonagh, N. Y. Hammerla, B. Kainz, *et al.*, "Attention u-net: Learning where to look for the pancreas," *arXiv preprint arXiv:1804.03999*, 2018.

[20] N. Abraham and N. M. Khan, "A novel focal tversky loss function with improved attention u-net for lesion segmentation," in *2019 IEEE 16th International Symposium on Biomedical Imaging (ISBI 2019)*, pp. 683–687, IEEE, 2019.

[21] F. Milletari, N. Navab, and S.-A. Ahmadi, "V-net: Fully convolutional neural networks for volumetric medical image segmentation," in *2016 fourth international conference on 3D vision (3DV)*, pp. 565–571, IEEE, 2016.

[22] D.-P. Fan, T. Zhou, G.-P. Ji, Y. Zhou, G. Chen, H. Fu, J. Shen, and L. Shao, "Inf-net: Automatic covid-19 lung infection segmentation from ct images," *IEEE Transactions on Medical Imaging*, vol. 39, no. 8, pp. 2626–2637, 2020.

[23] W. Dai, N. Dong, Z. Wang, X. Liang, H. Zhang, and E. P. Xing, "Scan: Structure correcting adversarial network for organ segmentation in chest x-rays," in *Deep learning in medical image analysis and multimodal learning for clinical decision support*, pp. 263–273, Springer, 2018.

[24] Y. Xue, T. Xu, H. Zhang, L. R. Long, and X. Huang, "Segan: Adversarial network with multi-scale l 1 loss for medical image segmentation," *Neuroinformatics*, vol. 16, no. 3, pp. 383–392, 2018.

[25] T. C. Kwee and R. M. Kwee, "Chest ct in covid-19: what the radiologist needs to know," *RadioGraphics*, vol. 40, no. 7, pp. 1848–1865, 2020.

[26] M. Jun, G. Cheng, W. Yixin, A. Xingle, G. Jiantao, Y. Ziqi, Z. Minqing, L. Xin, D. Xueyuan, C. Shucheng, W. Hao, M. Sen, Y. Xiaoyu, N. Ziwei, L. Chen, T. Lu, Z. Yuntao, Z. Qiongjie, D. Guoqiang, and H. Jian, "COVID-19 CT Lung and Infection Segmentation Dataset," Apr. 2020.

[27] S. Liu, D. Huang, *et al.*, "Receptive field block net for accurate and fast object detection," in *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 385–400, 2018.

[28] Z. Gu, J. Cheng, H. Fu, K. Zhou, H. Hao, Y. Zhao, T. Zhang, S. Gao, and J. Liu, "Ce-net: Context encoder network for 2d medical image segmentation," *IEEE transactions on medical imaging*, vol. 38, no. 10, pp. 2281–2292, 2019.

[29] T. Miyato, T. Kataoka, M. Koyama, and Y. Yoshida, "Spectral normalization for generative adversarial networks," in *International Conference on Learning Representations*, 2018.

[30] M. Serrurier, F. Mamalet, A. González-Sanz, T. Boissin, J.-M. Loubes, and E. del Barrio, "Achieving robustness in classification using optimal transport with hinge regularization," 2020.

[31] C. Vertan, *Prelucrarea si analiza imaginilor*, pp. 72–89. Printech, 1999.

[32] R. Müller, S. Kornblith, and G. Hinton, "When does label smoothing help?," *arXiv preprint arXiv:1906.02629*, 2019.

[33] V. Vasilescu, A. Neacşu, E. Chouzenoux, J.-C. Pesquet, and C. Burileanu, "A deep learning approach for improved segmentation of lesions related to covid-19 chest ct scans," in *2021 IEEE 18th International Symposium on Biomedical Imaging (ISBI)*, pp. 635–639, IEEE, 2021.