Medical Heart Image Analysis With Machine Learning Techniques And Deep Learning Neural Networks

by

Periklis Bouzanis

Submitted
in partial fulfilment of the requirements for the degree of

Master of Artificial Intelligence

at the

UNIVERSITY OF PIRAEUS

July 2022

University of Piraeus, NCSR “Demokritos”. All rights reserved.

Author...Periklis Bouzanis

II-MSc “Artificial Intelligence”

Month 07, 2022

Certified by Michalis Filippakis, Associate Professor

Thesis Supervisor

Certified by Ilias Maglogiannis, Professor

Member of Examination Committee

Certified by Maria Halkidi, Associate Professor

Member of Examination Committee

Medical Heart Image Analysis With Machine Learning Techniques And Deep Learning Neural Networks

By

Periklis Bouzanis

Submitted to the II-MSc "Artificial Intelligence" on July 2022, in partial fulfillment of the requirements for the MSc degree

Abstract

Human heart is considered one of the most import organs of the human body, since its job is to provide the body with blood [23]. One of the methods that clinicians utilize, to examine the heart and its internal structure condition, is the TransThoracic Echocardiogram (TTE), which is the most used, agile, and cost-effective cardiac imaging modality. Machine Learning techniques and Deep learning neural networks, implemented in TTE images, can deliver highly accurate and automated interpretation of heart's clinical condition, which can greatly assist cardiologists in their evaluation of heart's abnormality or not [1][2].

In the current master thesis, a deep learning algorithm will be examined in various dataset resolutions and a comparison of its performance on the task of classification of the enlargement of the left atrium of the human heart, with the use of TTE images from patients of a Greek Hospital, will be studied. The basic algorithm is a combination of a Unet and a Convolutional Neural Network (CNN). Unet will segment the A4C TTE images over the cardiac Left atria (LA) and CNN will classify the segmented images for normal or abnormal size of the LA. Addittionaly a Semi-supervised GAN will be trained and evaluated in classifying the cardiac LA as normal or abnormal.

Thesis Supervisor: Mr Michalis Filippakis
Title: Associate Professor

Acknowledgments

I would like to thank Associate Professor Michalis Filippakis of the Dept. of Digital Systems of University of Piraeus for supervising my research and providing the guidance needed for completing this thesis. Also, I would like to thank all the teaching faculty of this Master in AI from University of Piraeus and the NCSR "Demokritos", as well as clinical Cardiologist Mr. Anastasios Papaspyropoulos for the knowledge i acquired under his guidance, over the subjects of the human heart and the echocardiograms. Last, but not least, I would like to thank my family and friends for their patience and support.

Table of Contents

TABLE OF CONTENTS 3

LIST OF FIGURES 6

LIST OF TABLES 11

ABBREVIATIONS 13

1 INTRODUCTION 14

1.1 BRIEF DESCRIPTION 14

1.2 CHAPTERS DESCRIPTION 15

2 MACHINE LEARNING 17

2.1 SUPERVISED LEARNING 17

2.1.1 Classification Problems 19

2.2 UNSUPERVISED LEARNING 20

2.3 SEMI-SUPERVISED LEARNING 21

2.4 ARTIFICIAL NEURAL NETWORKS 22

2.4.1 Perceptron algorithm 23

2.4.2 Training process 24

3 CONVOLUTIONAL NEURAL NETWORKS (CNN) 27

3.1 CONVOLUTIONAL LAYER 27

3.2 ACTIVATION FUNCTIONS 32

3.3 POOLING LAYER 35

3.4 BATCH NORMALIZATION 37

3.5 REGULARIZATION 39

3.5.1 Dropout 39

3.5.2 L2 Regularization 40

4 SEGMENTATION WITH UNET 43

4.1 SEMANTIC SEGMENTATION 43

4.2 U-NET ARCHITECTURE 46
5 GENERATIVE ADVERSARIAL NETWORK (GAN) 49
5.1 BASIC GAN 49
5.2 SEMI-SUPERVISED LEARNING WITH GAN 51
6 HUMAN HEART AND TTE 53
6.1 HEART ANATOMY 53
6.2 CARDIAC LEFT ATRIAL (LA) ENLARGEMENT 55
6.3 2D TRANSTHORACIC ECHOCARDIOGRAPHY 56
6.3.1 Basic theory 56
6.4 2D TTE STANDARD TOMOGRAPHIC VIEWS 58
6.4.1 Left Atria check with TTE 59
7 EXPERIMENTS 63
7.1 DATASET 63
7.1.1 Dataset statistics 64
7.2 DATASET PREPARATION 64
7.3 METHODS 67
7.4 RESULTS 72
7.4.1 Unet 73
7.4.2 CNN Results 78
7.4.3 Unet - CNN Pipeline Results 80
7.4.4 SGAN 82
7.5 CONCLUSION 83
REFERENCES 84
APPENDIX 86
1.1 IMPORT LIBRARIES CODE 86
1.2 DATASET CREATION CODE 87
1.3 UNET CODE 100
1.3.1 Code for Training the Unet Models 100
1.3.2 Code for Evaluating Unet the Models 114
1.4 CNN CODE 127
1.4.1 Code For Training CNN Models 127

1.4.2 Code for Evaluating CNN Models *142*
1.5 CNN-UNET PIPELINE EVALUATION CODE 154
1.6 GAN CODE 164
1.6.1 Code for Train GAN *164*
1.6.1 Code for Evaluating GAN's supervised Discriminator *174*
1.7 DATASET STATISTICS 181

List of Figures

Figure 2.1, Supervised Learning Flow (source: [1])

Figure 2.2, Unsupervised Learning Flow, (source :[3])

Figure 2.3, clustering people according to theit height and weight (source :[4])

Figure 2.4, Semi-Supervised Learning Flow (source :[3])

Figure 2.5, single perceptron, ANN with one hidden layer, MLP network with two hidden layers and one output

Figure 2.6, (a) perceptron algorithm output for three features as input, using a differentiable activation function h(.)

Figure 2.7, Cost function minimization through training of MLP with a dataset of two target classes

Figure 3.1, Basic CNN architecture (source: www.medium.com/techiepedia/binary-image-classifier-cnn-using-tensorflow-a3f5d6746697)

Figure 3.2, a convolutional layer may have more than one feature maps (source: www.towardsdatascience.com/applied-deep-learning-part-4-convolutional-neural-networks-584bc134c1e2)

Figure 3.3, Image represented as a matrix (source [9])

Figure 3.4, Image in grayscale mode with three vertical lines of different color.

Figure 3.5, Application of convolution with the receptive field in a local area of the input image, results in the representative pixel of the local area, in the hidden layer of the CNN (source [9]).

Figure 3.6, Sliding the receptive field over the whole input image, results in the creation of the feature map of the input image in the hidden layer of CNN (source [9]).

Figure 3.7, Implementation of convolution on a 9 x 9 input image with a 3 x 3 filter/receptive field, which results to the 1st hidden layer's 7 x 7 feature map.

Figure 3.8, Implementation of convolution on a 9 x 9 input image with a 3 x 3 filter/receptive field and a stride of 2, which results to the 1st hidden layer's 4 x 4 feature map.

Figure 3.9, Implementation of convolution on a 9 x 9 input image with a 3 x 3 filter/receptive field , padding of 1 and a stride of 1, which results to the 1st hidden layer's 9 x 9 feature map.

Figure 3.10, ReLU activation function curve (source [8])

Figure 3.11, Sigmoid activation function curve (source [8])

Figure 3.12, Softmax activation function curve (source [8])

Figure 3.13, Implementation of Max pooling 2 x 2 with stride 1 on a 7 x 7 feature map, which results to the convolution layer output of size 6 x 6.

Figure 3.14, Implementation of average pooling 3 x 3 with stride 1 on a 7 x 7 feature map, which results to the convolution layer output of size 5 x 5.

Figure 3.15, (a) Fully connected layers, (b) After implementing Dropout with 0,5 random probability, which zeroes the values corresponding to the connections of the randomly chosen neurons (source: [9])

Figure 3.16, Network with a p dropout probability during training and the participation of dropped out weights with a factor of p, during testing.(source :[9])

Figure 4.1, Semantic segmentation on a TTE cardiac image: (a) Apical 4 chamber (A4C) view, (b) Precise Semantic segmentation mask over the cardiac Left atria, (c) Loosen semantic segmentation mask of the same area.

Figure 4.2, Semantic segmentation on an image with objects of different classes

(source:www.researchgate.net/figure/Example-of-2D-semantic-segmentation-Top-input-image-Bottom-prediction_fig3_326875064)

Figure 4.3, Semantic segmentation with different architectures. Output image is a representation of the masks of the different objects from the initial scene (source: www.researchgate.net/figure/Semantic-segmentation-CNN-architectures_fig2_321124704)

Figure 4.4, A characteristic U-net architecture (source: [18])

Figure 5.1, GAN basic semantic (https: // www.itrelease.com / 2020/ 06/advantages-and-disadvantages-of-generative-adversarial-networks-gan/)

Figure 5.2, GAN goal is, through training, the samples distribution pg produced by the Generator (green solid line) to resample with the data distribution samples pdata (black dotted line) ((a) through (d)), while Discriminator discovers the distribution (blue dotted line) that discriminates fake from real samples (source [19])

Figure 6.1, Heart's anatomy (source: www.texasheart.org/heart-health/heart-information-center/topics/heart-anatomy/)

Figure 6.2, Diagram of an ultrasound wave with its properties. (source [24])

Figure 6.3, Interaction between ultrasound wave and body tissue. In cardiac tissues reflection is the useful interaction that is used in 2d TTE (source [24])

Figure 6.4, 2D TTE standard tomographic views on the diastolic phase: (a) PLAX , (b) PSAX , (c) A4C , (d) A2C , (e) Apical long-axis , (f) S4C, (g) Suprasternal (source [25])

Figure 6.5, Biplane Method of disk summation technique (source [27])

Figure 7.1, A4C view images. Left images are the echo machine's caption during the diastolic phase. Right images are the cardiologist's assessment with the marked left cardiac atrial.

Figure 7.2, Top images: produced from the TTE software. Bottom images: cropped in 4:3 aspect ratio and downsized to 800 x 600 resolution

Figure 7.3, Annotation of the areas to produce the corresponding masks with the VIA software: (a) Extended mask, (b) Original mask from cardiologist assessment image

Figure 7.4, The final masks (middle images) and their filtering over the original image (right images)(a) Extended mask, (b) Original mask

Figure 7.5, Implemented U-Net architecture (source:[2])

Figure 7.6, Implemented CNN architecture (source:[2])

Figure 7.7, Unet - CNN pipeline for the extended masking models

Figure 7.8, Unet - CNN pipeline for the precise masking models

Figure 7.9, Generator of the Semi-supervised GAN (source:[2])

Figure 7.10, Discriminator of the Semi-supervised GAN (source:[2])

Figure 7.11, Segmentation results of 160 x 120 test set images from Unet with no data augmentation (extended masking)

Figure 7.12, Segmentation results of 160 x 120 test set images from Unet with data augmentation (extended masking)

Figure 7.13, Unet's per pixel normalized matrices with DA of test set images (extended masking): (a) 240 x 320, (b) 120 160

Figure 7.14, Segmentation results of 120 x 160 test set images from Unet without data augmentation (precise masking)

Figure 7.15, Segmentation results of 120 x 160 test set images from Unet with data augmentation (precise masking)

Figure 7.16, Unet's per pixel normalized matrices with DA of test set images (extended masking): (a) 240 x 320, (b) 120 160

Figure 7.17, CNN's normalized matrices with DA of test set images (extended masking): (a) 240 x 320, (b) 120 160

Figure 7.18, CNN's normalized matrices with DA of test set images (precise masking): (a) 240 x 320, (b) 120 160

Figure 7.19, Segmentation results of 120 x 160 test set images from Unet with data augmentation (extended masking 120 x 160)

Figure 7.20, Unet-CNN pipeline's normalized matrices on test set : (a) extended masking (120 x 160), (b) precise masking (120 x 160)

List of Tables

Table 6.1, Dataset Statistics

Table 6.2, U-Net Extending Masking Results

Table 6.3, U-Net Precise Masking Results

Table 6.3, CNN Extending Masking Results

Table 6.4, CNN Precise Masking Results

Table 6.5, Unet - CNN Pipeline models

Abbreviations

ANN = Artificial Neural Network

CNN = Convolutional Neural Network

DA = Data Augmentation

DL = Deep Learning

DNN = Deep Neural Network

LA = Left atria

ML = Machine Learning

MLP = Multiple Layer Perceptron

TTE = Transthoracic Echocardiogram

1 Introduction

1.1 Brief Description

Human heart is considered one of the most import organs of the human body, since its job is to provide the body with blood [23]. One of the methods that clinicians utilize, to examine the heart and its internal structure condition, is the TransThoracic Echocardiogram (TTE), which is the most used, agile, and cost-effective cardiac imaging modality. A TTE consists of videos, images and doppler measurements from different cross-sections of heart [24]. Machine Learning (ML) techniques and Deep learning (DL) neural networks (DNN), implemented in TTE images, can deliver highly accurate and automated interpretation of heart's clinical condition, which can greatly assist cardiologists in their evaluation of heart's abnormality or not [1][2].

In the current master thesis, a deep learning algorithms will be examined with various and a comparison of their implementation and performance on the task of classification of the enlargement of the Left Atrial(LA) of the human heart, with the use of Apical 4 champer (A4C) TTE images from patients of a Greek Hospital, will be studied. In particular, a U-Net will be trained on the aforementioned images in the task of image segmentation over heart's LA area and its output will be fed as an input to a Convolution Neural Network (CNN), which will be trained to classify the heart's LA size as normal or pathological.

The two networks will be trained on two resolutions of the image inputs, the 120 x 160 and the 240 x 320. Additionally, we are going to use two kinds of image masking. An extended masking over the LA'a area and the precise one, over cardiologist's assessment area, while the networks will be trained with no data augmentation (DA) and with data augmentation (DA) techniques , in order to evaluate their contribution on small sized medical datasets.

Addittionaly a Semi-supervised GAN will be trained and evaluated in classifying the cardiac LA as normal or abnormal with the use of an unamsked image dataset of A4c images.

1.2 Chapters Description

In Chapter 2, the basic terms of the Machine Learning are described, which are common for the Deep Learning. According to this, we analyze the terms of supervised , unsupervised and semi-supervised learning. For the supervised learning, the subcategory of classification problems is presented, since our case of categorizing the cardiac left atrial size as abnormal or normal, belongs to this category. Additionally, a description of Artificial Neural Networks (ANN) is given and the perceptron algorithm is analyzed thoroughly, along with the training process of an ANN, which includes the processes of forward feeding - calculation of predictions – backpropagation and ANN's weight updates through the gradient descent technique.

In Chapter 3, the basic architecture of a CNN is analyzed, such as the convolutional and pooling layers, the sigmoid and softmax activation function. Also, various techniques are presented to improve a CNN's performance. For this, Batch normalization and the regularization tools of dropout and L2 regularization are discussed.

In Chapter 4, we deal with the concept of the image segmentation. We describe the basics of image segmentation and present the Jaccard Index and power Jaccard loss as a suitable metric and a loss function respectively, for the segmentation tasks. Afterwards, we analyze the U-net as a CNN, which has been developed mainly for segmentation and precise localization on biomedical images.

In Chapter 5, we describe the concept of the Generative Adveraial Network (GAN), its basic theory for training and producing real like images from an image distribution. Then we describe a specialized use of the GAN in semi-supervised learning task, where its Discriminator part can be trained, additionaly and as a classifier.

In Chapter 6, we make a description of the human heart anatomy and its basic functions. We give the theory on which the TTE relies on, present the basic view images a cardiologist can produce with the TTE and the internal heart's structure that can be examined with each view. Finally we present the state of the LA enlargement and how it is traced through the TTE and its normal values.

In chapter 7, we present our experiment results over the classification task of LA size as normal or abnormal. We present the dataset that was used, its population statistics and its preparation for each of our models. Afterwards, we describe our U-net implementation for the segmentation of the LA on A4C view images of TTE and our CNN and Seimi-supervised GAN implementation for the classification. Finally, we analyze the results of training the above Deep Networks over our dataset.

2 Machine Learning

Machine Learning (ML) is a process that a computer system applies in order to solve a given problem, according to previous examples of the same domain, taking into account the probabilistic distribution of these examples. A definition of the Machine Learning is given by T. Mitchell (1997) [3]: "A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*". This definition reveals the basic features of a machine learning algorithm [3]:

- The class of tasks (*T*): the exact problem the algorithm must become capable of solving.
- The measure of performance (*P*): The evaluation metric of the algorithm's capability of accomplishing the tasks.
- The source of experience (*E*): the task's domain previous cases (training examples), which the algorithm uses to produce a cased based reasoning process.

The ML algorithms are divided into three major categories, with respect to the use or not of the examples category labels/continuous values/similarities, during algorithm's training process:

- Supervised
- Unsupervised
- Semi – supervised

2.1 Supervised Learning

In supervised learning the source of the experience is provided as a matrix $X_{m,n}$ along with a label vector Y_m . Matrix $X_{m,n}$ stores the *m* examples of the examined case, where a row belongs to a single example, while each of the *n* columns stores the value of a specific feature of this example. Vector *Y* associates each example with a category/class or a continuous value (ground truth), with respect to the problem that the machine learning algorithm is

intended to solve. Domain's experts of the examining problem have assigned the ground truth. Transferring this to the current problem of the cardiac LA enlargement classification, each row of the sample matrix X describes a TTE image, while columns encapsulates a 2D matrix, whose elements correspond to the image's pixels values (example's features). An element in vector $\boldsymbol{Y}$ associates the normal or the abnormal size of the cardiac LA of the respected image in the matrix X**.**

During the algorithm's training procedure, the algorithm computes an output for each example. According to the supervised learning paradigm, the main goal of the algorithm is to optimize its parameters, in order to minimize the difference between the assigned target output of vector Y (the ground truth) and its computed output [3], thus minimizing the misclassification or the error between the target and the computed continuous value. The final evaluation of the ML algorithm is accomplished by implementing it in a test set with examples from the same distribution as the training set, that have not been used in the training set, and measuring test set's error predictions. The prepose of the, unseen before, test examples, is to make the algorithm capable to generalize beyond the training set.

In order to achieve this, a supervised learning algorithm follows a procedure of standard steps (Fig. *2.1)*:

- Forward propagation, where algorithm computes an output for the training examples of X
- Gradient Descent, where the error between the target and the computed outputs, is calculated.
- Backpropagation, where the algorithm's parameters are updated according to their partial derivatives from the gradient descent.

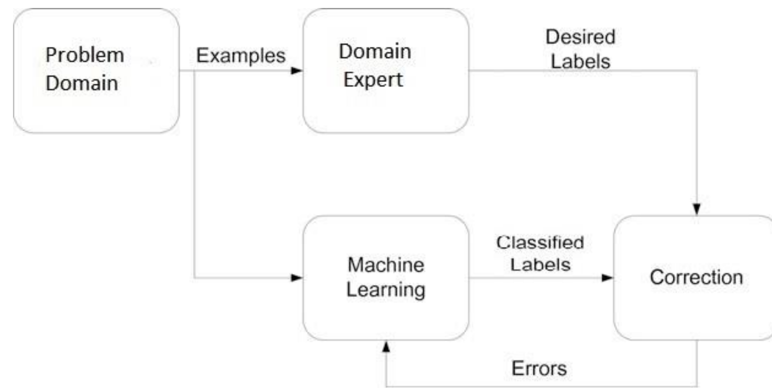


Figure 2.2, Supervised Learning Flow

2.1.1 Classification Problems

One of the main implementations of the supervised learning is on classification problems. In the classification problems, the algorithm's goal is to discover a mapping from an input example x ∈ X *to an output* $y \in Y = \{1, \ldots, C\}$, where C denotes the number of classes of the specific problem. If $C = 2$, then the problem is called a binary classification problem and the most common output labelling is $Y = \{0, 1\}$, where "*0*" means the negative meaning of a characteristic for the example x, while "*1*" denotes the positive characteristic for the example x (e.g., "0" for abnormal size of cardiac LA and "1" for the normal size, "0" for a spam email and "1" for a non-spam etc.). If C > 2, then the problem is called multiclass classification problem, where an example is assigned with a specific class label (e.g. types of iris flowers: setosa, versicolor and virginica). Additionally, the classification problem, where an example can belong in more than one class (a man can be tall and overweighted), is called a multi-label classification problem.

The formalization of a classification problem is made by a function approximation. The supervised learning algorithm must discover, with its training, a function *f*, which maps each example x with an output class y. The algorithm, in order to compute an output $\hat{y}$ for a specific example x, uses a probabilistic activation function $\hat{f}$. This function, for a binary classification problem, is usually the sigmoid function, which outputs values between 0 and 1. If $\hat{f}(x) \geq 0.5$ then the computed output $\hat{y}$ is classified to the class "1", otherwise it belongs to class "0". If the examined problem is a multiclass problem, then the most usually function $\hat{f}$ is the softmax function, which output, for an example x,

is the probabilities the example to belong to each class. Then the computed output $\hat{y}$ is classified to the class with the highest probability.

2.2 Unsupervised Learning

Unsupervised learning is the ML paradigm, where the algorithm is provided only with input examples $x \in X$ and have no knowledge about which class or what continuous value these examples belong to or are assigned with, according to their features θ, since no ground truth output Y is provided or is ad hoc known. Algorithm's goal is to discover common structures (namely clusters) that characterize the examples and assign its example to one of these structures (clusters), taking into account their features (*Fig. 2.2, 2.3*). This procedure leads in achieving the optimization of the cluster prototypes, from the similarity of their respective examples. In this learning paradigm, if k is the number of chosen clusters, we are interesting in inferring the cluster, in which an example x belongs to, by computing the unconditional density estimation p(k|X), where the task is to build models of the form p =(x_i|θ) [3][4].

An implementation of unsupervised learning is in e-commerce, where users/clients are assigned to a specific behavior cluster, according to their purchasing and web surfing habits, so as a more focused advertising strategy to be applied to each group, in respect to each groups discovered characteristics.

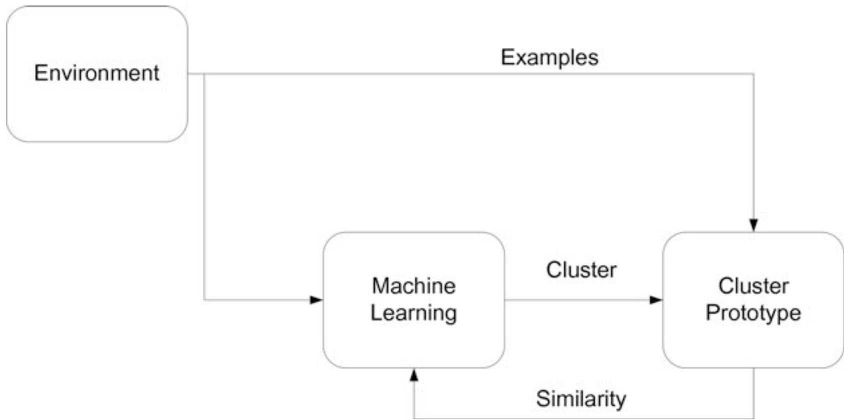
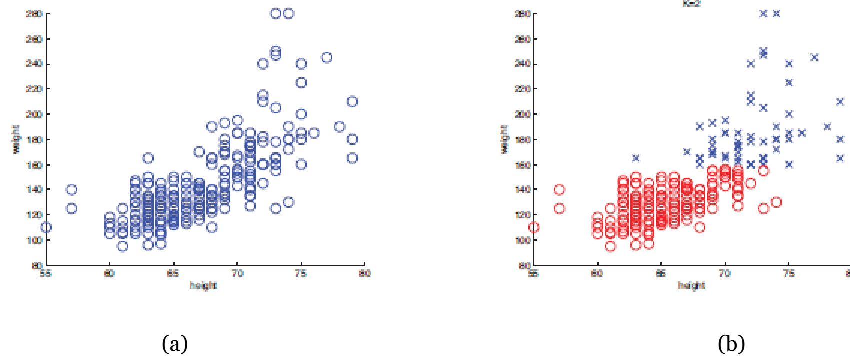


Figure 2.2, Unsupervised Learning Flow



(a) (b)

Figure 2.3, (a) Unlabeled data of height and weight of people, (b) Clustering of people into K = 2 clusters according to similarities in their height and weight

2.3 Semi-Supervised Learning

Semi-Supervised learning is a mixture/combination of supervised and unsupervised learning. In this mML paradigm, examples $x \in X$ with their ground truth Y are provided, along with examples $x' \in X'$ without ground truth in an effort to overperform a relative supervised algorithm (*Fig. 2.4)*. Semi-supervised learning tries to solve the problem of the expensive and difficult to find labeled data x from some domains, with the use of the unlabeled data x', which are available in large amount, easy to obtain and cheaper as they do not need annotation from an expert.

In order to achieve this, the algorithm uses an unsupervised part to categorize the unlabeled data into clusters by discovering their common structures. The number of clusters are defined by the number of classes of labeled data. Thus, the unlabeled data are eventually categorized according to those clusters and they added to the labeled data. Then, the supervised learning part of the algorithm uses an increased training dataset by using both labeled and clustered (previously unlabeled) data in its training process, thus achieving a better generalization over the examined problem [3][5].

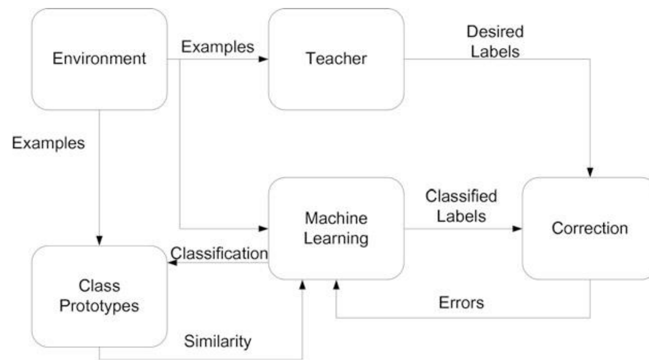


Figure 2.4, Semi-Supervised Learning Flow

2.4 Artificial Neural Networks

Artificial Neural Networks (ANN) were developed by Frank Rosenblatt in 1960's. An ANN is a computational system, which goal is to discover mathematical representations of information processing by mimicking the human brain's neurons operations, thus recognizing the deeper relationships of the features of a set of data [6]. The main part of the ANN is the perceptron (or neuron), which receives a sample's features as input and calculates an output. The combination of multiple perceptrons in a layer, builds an ANN, while multiple layers of perceptrons consist a Multiple Layer Perceptron (MLP) network (*Fig. 2.5*).

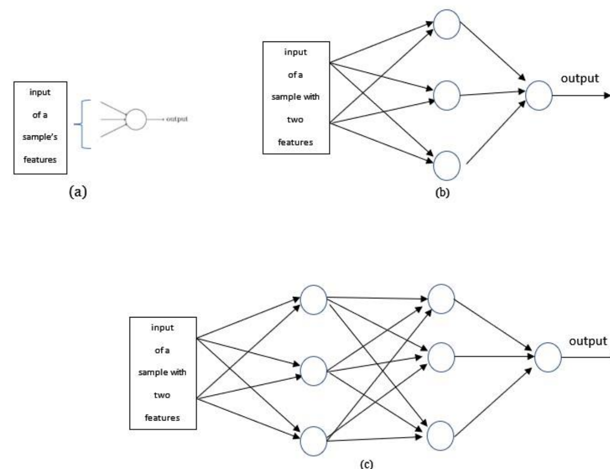


Figure 2.5, (a) *single perceptron,* **(b)** *an Artificial Neural Network (ANN) with one hidden layer,* **(c)** *a Multiple Layer Perceptron (MLP) network with two hidden layers and one output*

2.4.1 Perceptron algorithm

A Neuron uses the perceptron algorithm to compute the output, which will lead to a sample's predicted value/class. The perceptron receives as an input, the D features of a sample, it takes the sum of products between each feature x_i and a respective coefficient w_i (weight) and finally adds a bias coefficient b of the neuron to calculate its *output* z:

$$z = b + \sum_{i=1}^{D} w_i x_i \qquad \textbf{(1)}$$

The weights are used to evaluate the magnitude of each feature's contribution to the output z. Because output z is a linear function over the features x and coefficients w and may take high values, which can cause computational issues, the output z of the neuron is transformed with the use of a nonlinear, differentiable *activation function h(.)*. The application of an activation function to neuron's output z returns the final output of a neuron (*Fig. 2.6*):

$$\alpha = h(z) = h(b + \sum_{i=1}^{D} w_i x_i) \qquad \textbf{(2)}$$

Activation function must have some properties in order to be helpful in the perceptron's algorithm. It must be zero centered and differentiable, so as to add the non-linearity to the perceptron's algorithm. Non-linearity is an import aspect of an ANN, as it permits ANN to approximate any function to describe a dataset.

If we combine multiple neurons to construct an ANN with one hidden layer (like this in *Fig. 2.7(b))* of M neurons, then for each hidden new neuron j , the equations (1) and (2) are becoming:

$$z_j^{(1)} = b_{j0}^{(1)} + \sum_{i=1}^{D} w_{ji}^{(1)} x_i \qquad \textbf{(3)}$$

$$a_j^{(1)} = h(z_j^{(1)}) = h\left(b_{j0}^{(1)} + \sum_{i=1}^{D} w_{ji}^{(1)} x_i\right) \qquad \textbf{(4)}$$

The (1) on the equations denotes the number of layer's depth in the network. Then, on the output layer of the ANN, with *K* output neurons, since each output neuron receives as input the outputs of the hidden layer, for the *kth* output neuron (k =1,...,K), the equations will give [4]:

$$z_k^{(2)} = b_{k0}^{(2)} + \sum_{j=1}^{M} w_{kj}^{(2)} \alpha_j^{(1)} \quad \textbf{(5)}$$

$$\alpha_k^{(2)} = h(z_k^{(2)}) = h\left(b_{k0}^{(2)} + \sum_{j=1}^{M} w_{kj}^{(2)} \alpha_j^{(1)}\right) \quad \textbf{(6)}$$

Eventually the equations (5) and (6) can be used for any MLP of *L layers* total depth. So, on any layer *l* (*l=1,...,L)*, a neuron *n* of this layer, with *N* outputs from the previous layer (*l-1),* the equations will give:

$$z_n^{(l)} = b_{n0}^{(l)} + \sum_{p=1}^{N} w_{np}^{(l)} \alpha_p^{(l-1)} \quad \textbf{(7)}$$

$$\alpha_n^{(l)} = h(z_n^{(l)}) = h\left(b_{n0}^{(l)} + \sum_{p=1}^{N} w_{np}^{(l)} \alpha_p^{(l-1)}\right) \quad \textbf{(8)}$$

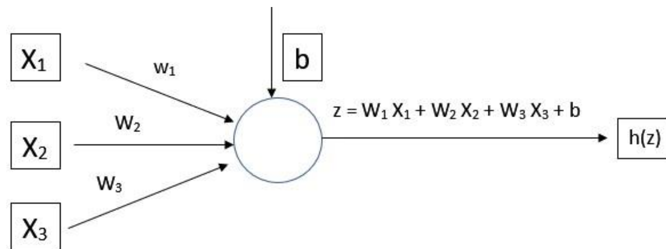


Figure 2.6, (a) *perceptron algorithm output for three features as input, using a differentiable activation function h (·)*

2.4.2 Training process

The goal of applying a training process in an MLP network is to enforce the network to seek and find this function that describes better a dataset of a certain distribution and generalizes well over the same data. This task is accomplished by learning the values of MLP's weights and biases that describe this function

and enforce a cost function between the calculated outputs and the ground truth to find a total minimum (*Fig. 2.7*). The training process is implemented through a repeated steps (epochs) of two successive subprocesses, the Feed forward process and the Backpropagation process, using the Gradient descent technique.

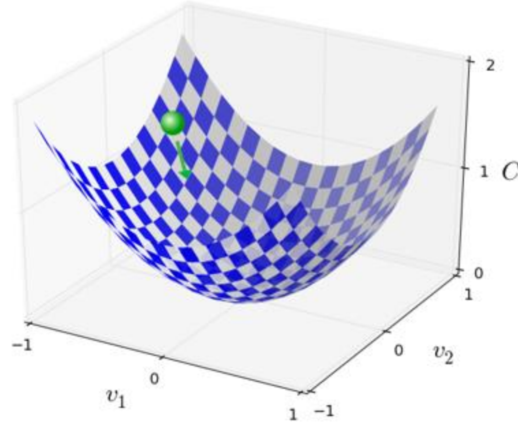


Figure 2.7, *Cost function minimization through training of MLP with a dataset of two target classes.*

Before training, network's weights and biases are usually initialized randomly with values from a chosen probabilistic distribution. During the *Feed forward* process the network is feed with a training dataset (input) X = {x_1, x_2,…,x_N} of *N* independent sample vectors *x* from the same distribution. On each hidden layer, for each neuron, activations and the outputs of activation function are calculated according to the equations (7) and (8). The outputs of the last layer's activation functions of the MLP are the network's predictions $\hat{y}$ over the training samples x_i. The algorithm in order to determine its efficiency, calculates the averaged, over all samples N, *Cost* function *J,* with the use of the *Loss* function *L*:

$$J(W) = \frac{1}{2N}\sum_{i=1}^{N} L(\hat{y}_i, y_i) = \frac{1}{2N}\sum_{i=1}^{N} L(h(x_i;\, W), y_i) \qquad \textbf{(9)}$$

where *W* corresponds to the weight vector of the last layer *l*, of the MLP.

The feedforward process is complete with the computation of the Gradients of the weights through the calculation of the first class derivatives of the *Cost function* over the weights W*:*

$$dJ(W^{(l)}) = \frac{\vartheta J(W^{(l)})}{\vartheta W^{(l)}} = \frac{1}{2N} \sum_{i=1}^{N} \frac{\vartheta\left(L\left(h\left(x_i; W^{(l)}\right), y_i\right)\right)}{\vartheta W^{(l)}} \quad \textbf{(10)}$$

Backpropagation process begins with the use of the Gradient Descent technique with which the weights of the last layer are updating their values by subtracting from them their respective previous derivatives multiplied by a learning rate η:

$$W^{(l)} = W^{(l)} - \eta \frac{\vartheta J(W^{(l)})}{\vartheta W^{(l)}} \quad \textbf{(11)}$$

Then, the calculation of the weights on every layer of the MPP is accomplished through the application of the chain rule of calculus, in order to backpropagate the weight update from output to input:

$$dJ(W^{(l-1)}) = \frac{\vartheta J(W^{(l)})}{\vartheta \hat{y}} \cdot \frac{\vartheta J(\hat{y})}{\vartheta W^{(l-1)}} = \frac{\vartheta J(W^{(l)})}{\vartheta \hat{y}} \cdot \frac{\vartheta \hat{y}}{\vartheta a^{(l-1)}} \cdot \frac{\vartheta a^{(l-1)}}{\vartheta W^{(l-1)}} \quad \textbf{(12)}$$

The above process of feedforward – predictions $\hat{y}$ – cost calculation – backpropagation for the derivatives - weight update through gradient descent, when it is applied one time to the whole dataset, corresponds to one epoch of the MLP's training. The goal of the training is achieved, when through many epochs, the MLP finds the parameter values (weights) that minimize the cost function.

3 Convolutional Neural Networks (CNN)

Convolutional neural networks (CNNs), are α type of artificial neural networks (ANN), which are used in deep learning and are implemented in processing input data that has a known form of single or multiple dimension array. The main domains of CNN's application are the processing of images, video, audio, and speech, where the data can be represented in matrix like form. The main parts of the CNNs are the convolution layer, which is responsible for the implementation of the convolution operation and encompasses the input's produced features, the pooling layer, which function is to encapsulate the most important and interesting parts of a convolutional layer and the fully connected layer(s), which is responsible for inferring the function that describes best the features of the input. A typical full CNN consists of repeated sequences of the couple convolution – pooling layer and may have as last layer one or more fully connected layers (*Fig. 3.1)*.

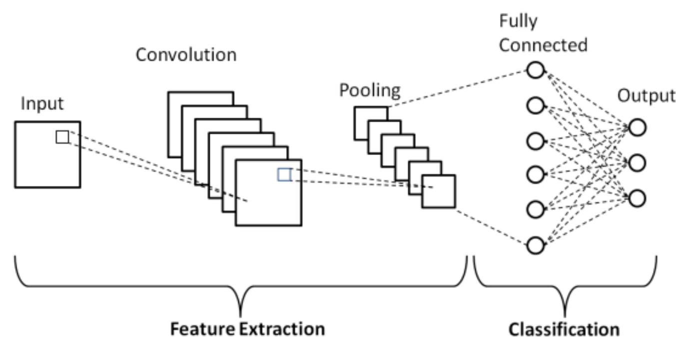


Figure 3.1, *Basic CNN architecture*

3.1 Convolutional Layer

The main part of a CNN is the convolution layer. As its name indicates, it applies a mathematical, convolution like, linear operation, to the output of the previous layer of the CNN, which replaces the matrix multiplication process. The convolution linear operation is implemented through a local receptive field or linear filter. The results of the convolution operation are then fed to a non-linear activation function σ. This leads to the convolutional layer's unit output, which

is referred to as feature map. A single convolutional layer consists of more than one feature maps (*Fig. 3.2)*. Each feature map describes specific characteristics of the previous layer's output, which are closely correlated to a specific filter that led to the exact feature map [7]. During the training of a CNN, one of the goals is to learn those receptive fields (linear filters), which lead to a better representation of the input, that is, to compute the filters, which discover better the different features of the input data.

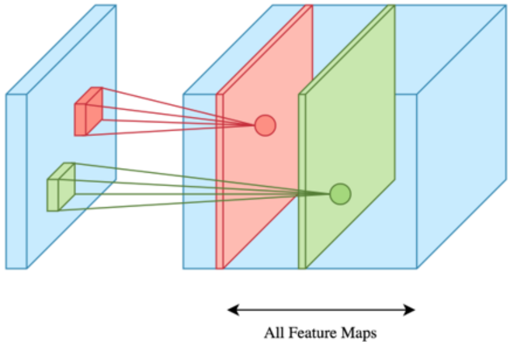


***Figure 3.2**, a convolutional layer may have more than one feature maps*

The operation of convolution is described as the inner product between every local area of the input matrix, of size equal to a receptive field (linear filter). Using as example, an input image as a matrix of size 28 x 28, where each cell represents an image's pixel value, usually between 0 and 255 in RGB representable mode in grayscale mode (*Fig. 3.3)*. Each group of neighboring pixels in this image can encapsulate different kind of image's features, while a specific group of neighboring pixels can encompass a specific image feature, which may appear in different areas in the image, although the values in the pixels may different (*Fig. 3.4*).

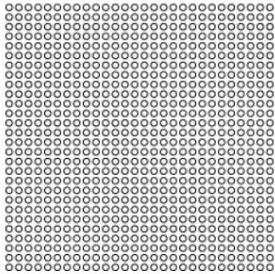


Figure 3.3: Image represented as a matrix

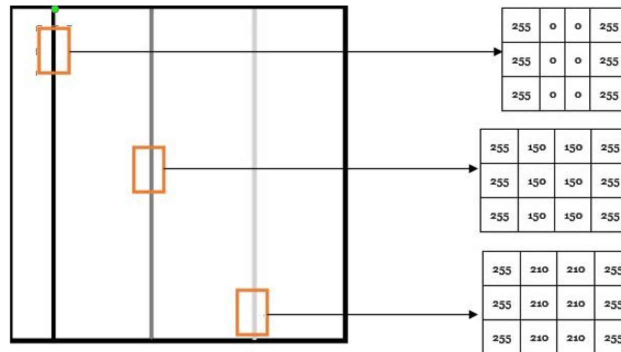


Figure 3.4, Image in grayscale mode with three vertical lines of different color. The same vertical line feature is appearing in different areas of the image, while their respective pixel values are different, since their color is different.

The convolution is implemented by applying the receptive field to a local area of the image, where a linear transformation is taking place between the values of the receptive field and the pixel values of this area. This transformation leads to an output weighted sum, which describes the image's local area. In order to accomplish the convolution function, the center value of the receptive field is placed on the input image's pixel of interest. Thus, the receptive field will overlap the neighboring pixels of the input image's pixel of interest. Then, the values of the output are calculated by multiplying each receptive field's value by the corresponding input image pixel values and summing the multiplication results. The sum result of applying the receptive field to every local area of the input by sliding it over the whole image, is the corresponding feature map (*Fig. 3.5, 3.6*), which constitutes the first hidden layer of the CNN.

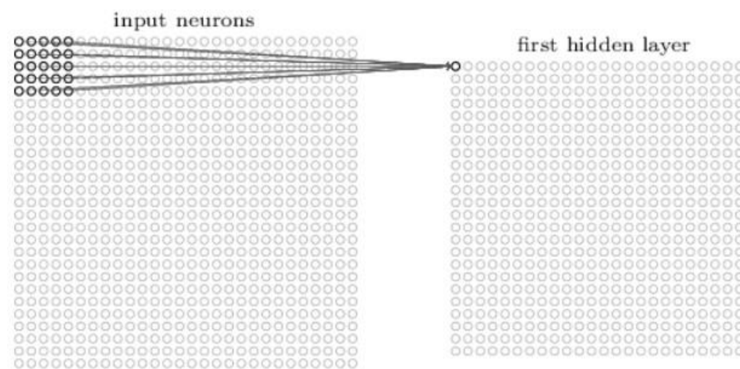


***Figure 3.5**, Application of convolution with the receptive field in a local rea of the input image, results in the representative pixel of the local area, in the hidden layer of the CNN*

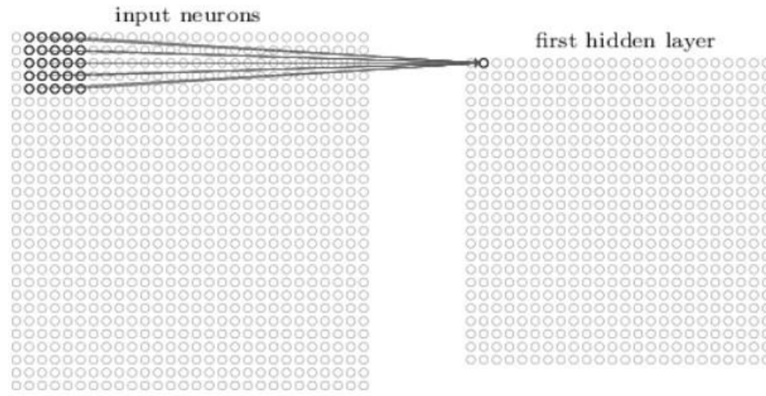
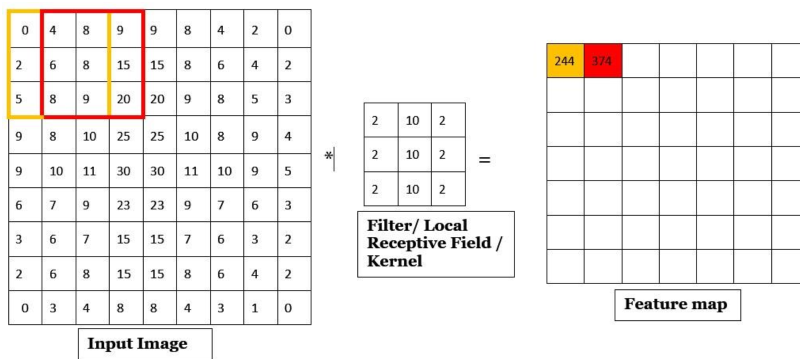


***Figure 3.6**, Sliding the receptive field over the whole input image, results in the creation of the feature map of the input image in the hidden layer of CNN*

If the input is a matrix $X_{m,n}$ and the local receptive field (filter) is of size *f x f*, then the value *z* of each hidden neuron on *i,j* position of the feature map is [9]:

$$z_{i,j} = b + \sum_{l=0}^{f} \sum_{m=0}^{f} w_{l,m} \cdot a_{i+l,j+m} \quad \textbf{(13)}$$

where b is the shared bias value across all the hidden neurons of the filter, $w_{l.m}$ is an *f x f* array representation of the filter, which implements the filter's *shared weights* to be calculated throw the backpropagation process and $a_{i+l,j+m}$ is the input's pixel value when the filter is projected to the input image (*Fig. 3.7*).



Hidden neuron at position 1,1 of feature map:

0*2 + 4*10 + 8*2 + 2*2 + 6*10 + 8*2 + 5*2 + 8*10 + 9*2 = 244

Hidden neuron at position 1,2 of feature map:

4*2 + 8*10 + 9*2 + 6*2 + 8*10 + 15*2 + 8*2 + 9*10 + 20*2 = 374

***Figure 3.7**, Implementation of convolution on a 9 x 9 input image with a 3 x 3 filter/receptive field, which results to the 1st hidden layer's 7 x 7 feature map.*

The result of implementation of the convolution process in an input image is the 1st hidden layer's feature map, which neurons share the same weights and bias. This enables the feature map to detect a specific same feature throw out the input image. If we apply more than one filter on the same hidden layer, then from each filter, a different feature map will be produced, enabling the detection of different features of the input image. Feature of an image can be a vertical or horizontal line, a curved line, or any other type of shape.

The size of the feature map is depended from two factors, during the convolution process, apart from the size of the input image and the applied filters at each hidden layer of the CNN. The first one is the *stride*, which corresponds to the horizontal and vertical step of the filter projection over the input image (or the previous layer's extracted feature map). Choosing a different stride of value 1, allows CNN to search different kinds of features and at the same time to reduce the computational effort, since the resulted feature map becomes smaller (*Fig. 3.8*). The second factor is *padding*, which is a technic of adding extra pixels with zero values around the input image's margin (or the previous layer's extracted feature map). Padding, allows the CNN to give more attention to the border pixels of the input at each CNN layer, while at the same time reduces the degradation of the size of the resulted feature map and thus enables it to lose less information (*Fig.3.9*).

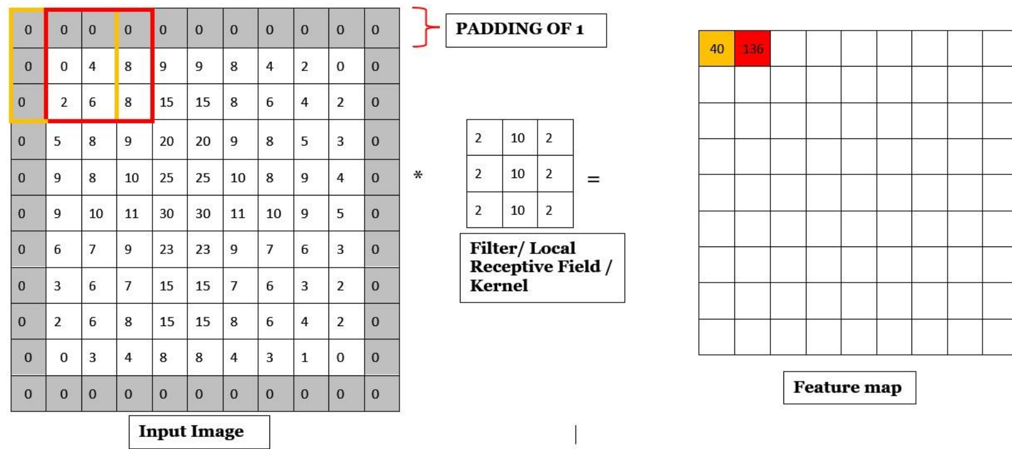
Hidden neuron at position 1,1 of feature map:

0*2 + 4*10 + 8*2 + 2*2 + 6*10 + 8*2 + 5*2 + 8*10 + 9*2 = 244

Hidden neuron at position 1,2 of feature map:

8*2 + 9*10 + 9*2 + 8*2 + 15*10 + 15*2 + 9*2 + 20*10 + 20*2= 578

Figure 3.8*, Implementation of convolution on a 9 x 9 input image with a 3 x 3 filter/receptive field and a stride of 2, which results to the 1st hidden layer's 4 x 4 feature map.*



Hidden neuron at position 1,1 of feature map:

0*2 + 0*10 + 0*2 + 0*2 + 0*10 + 4*2 + 0*2 + 2*10 + 6*2 = 40

Hidden neuron at position 1,2 of feature map:

0*2 + 0*10 + 0*2 + 0*2 + 4*10 + 8*2 + 2*2 + 6*10 + 8*2 = 136

Figure 3.9*, Implementation of convolution on a 9 x 9 input image with a 3 x 3 filter/receptive field, padding of 1 and a stride of 1, which results to the 1st hidden layer's 9 x 9 feature map.*

If we denote as $\boldsymbol{M_l}$ the size of the feature map of the $\boldsymbol{l}$ ***th*** layer of a CNN, $\boldsymbol{f}$ the filter size, p the padding and ***s*** the stride, then the $\boldsymbol{M_l}$ size of the feature map is calculated from the equation [9]:

$$M_l = \frac{M_{l-1} + 2p - f}{s} + 1 \qquad \textbf{(14)}$$

3.2 Activation Functions

An application of an activation function follows the output of a convolutional layer. The output of the convolutional layer is the product of a linear transformation, as it is demonstrated in equation (3). This linearity, during the error backpropagation process, for the update of the weights, the applied differentiation will cause the zeroing of the weights of the model and the model will not be capable of improving itself during its training process. The

introduction of a nonlinear activation function, prevents this behavior and transforms the model from a linear to a non linear one, allowing it to learn more complex decision functions, apart from the linearly separable ones. The output after the application of the activation function for the neuron of equation (3) is:

$$\alpha_{i,j} = \sigma(z_{i,j}) = \sigma\left(b + \sum_{l=0}^{f} \sum_{m=0}^{f} w_{l,m} \cdot a_{i+l,j+m}\right) \quad \textbf{(15)}$$

The most used activation function for the inner layers of a CNN is the *Rectifier Linear Unit (RelU)*. RelU returns the convolutional layer's output if is a positive value, otherwise it returns zero (*Fig. 3.10*):

$$RelU(z) = \begin{cases} z, & z > 0 \\ 0, & z \leq 0 \end{cases} \quad \textbf{(16)}$$

RelU's linear behavior for positive inputs, allows the model to be trained easier and achieve better performance. Its possible large positive outcomes, prevents the derivatives of inner layers to vanish during the error backpropagation process. The gradient $\frac{\vartheta z}{\vartheta x} = \frac{\vartheta z}{\vartheta y}\frac{\vartheta y}{\theta x}$, although it will have smaller magnitude than $\frac{\vartheta z}{\vartheta y}$, it will allow the derivatives to be propagate in deeper hidden layers of a CNN model, than the sigmoid function [8].

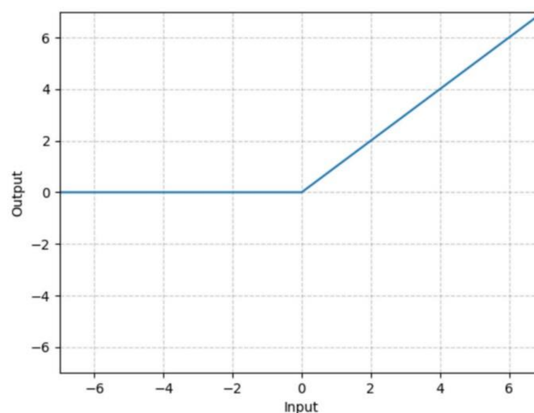


***Figure 3.10**, RelU activation function curve*

Sigmoid activation function is mainly used on the last layer of a CNN, after the fully connected layers, when the CNN must learn to solve a binary classification problem. For a given input *z*, sigmoid outcome $y = \sigma(z)$ takes values in the space (0,1) (Fig. 3.11):

$$\text{sigmoid}(z) = \frac{1}{1+e^{-z}} \qquad \textbf{(17)}$$

If the sigmoid outcome of the CNN for a sample of the dataset is under 0.5 then the sample is categorized in the class 0 (absence of a characteristic), otherwise in class 1. The fact that sigmoid function returns values in the space (0,1), prevents it from be used on the hidden layers of a CNN. The gradient $\frac{\vartheta z}{\vartheta x} = \frac{\vartheta z}{\vartheta y}\frac{\vartheta y}{\theta x}$ will have much smaller magnitude than $\frac{\vartheta z}{\vartheta y}$ and in deeper CNN layers will eventually vanish, making the gradient-based learning practical impossible.

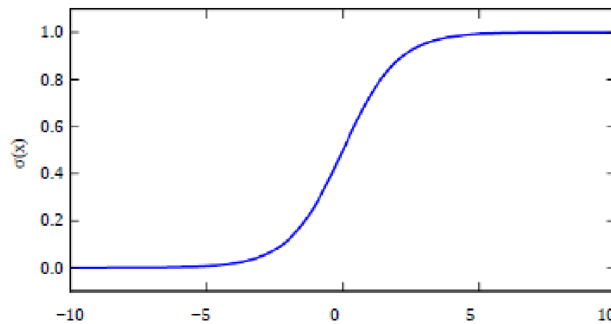


***Figure 3.11**, Sigmoid activation function curve*

Another, widely used, activation function is the *softmax*. It is used on the last layer of a CNN model, when the CNN must learn to solve a multiclass problem. It calculates, from the output logits of the last fully connected layer, the probability distribution of a sample, over predicted output classes C (*Fig. 3.12*). It is known as the normalized exponential and is calculated by the equation [10]:

$$\hat{f}(x) = P(y = c|x \in X) = \frac{e^{-w^T x}}{\sum_{c=1}^{C} e^{-W^T X}} \quad \textbf{(18)}$$

$$\hat{y} = argmax_{c=1}^{C} \hat{f}(x) \quad \textbf{(19)}$$

For the sample i, the computed output $\hat{y}$ is classified to the class with the highest probability.

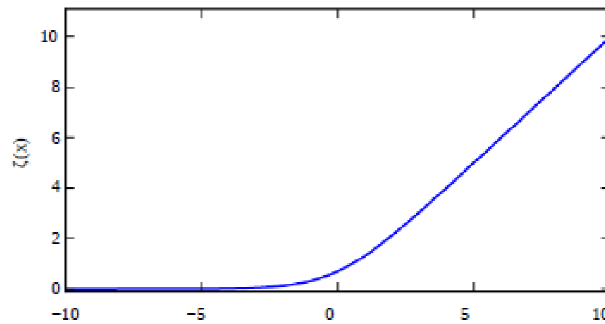


***Figure 3.12**, Softmax activation function curve (source [8])*

3.3 Pooling Layer

The third main part of a CNN layer is the *pooling layer*. It is applied after the activation function, direct on the current layer's feature map. Its goal is to replace the values at each certain location of a feature map with a summary statistic of the nearby values, reducing at the same time the size of the feature map by a factor promotional to the pooling layer's size.

Thus, pooling operation allows the CNN's representation to become less invariant to small alterations of the input [7], meaning that if small changes take place to the pixel's values of the input image (or to the values of a feature map for inner layers of a CNN), the values of the respective pooling layer outputs will mostly remain unchanged. This helps each convolution layer and the total CNN to learn a function that is invariant to small input's translations and consequently to be able to generalize better. Additionally, the reduction of the feature map's size improves the computational efficiency, since the next convolutional layer will have to apply less computational effort over the

diminished input [7]. Moreover, the pooling operation does not make use of any share weights and so is not participating in the backpropagation process, during the training process, reducing the use of memory usage and thus reducing more the computational effort of the CNN's algorithm.

The most common pooling operations on CNNs are the *Max pooling* and the *Average pooling*. *Max pooling* is applied when from each certain location of the feature map with size equal to pooling's size, we keep the maximum value, which becomes the representative value for this location. During the application of the *Average pooling*, the represented value of its location on the feature map is the average value of the location's corresponding values (*Fig. 3.13, 3.14*).

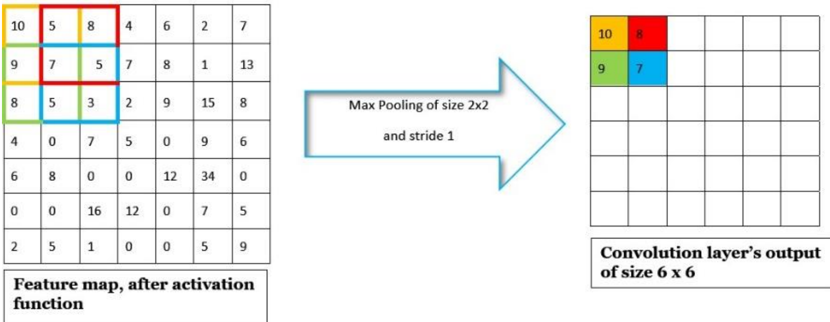


Figure 3.13, *Implementation of Max pooling 2 x 2 with stride 1 on a 7 x 7 feature map, which results to the convolution layer output of size 6 x 6.*

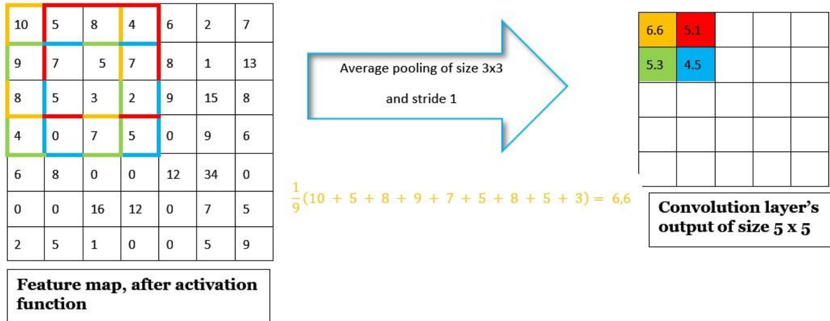


Figure 3.14, *Implementation of average pooling 3 x 3 with stride 1 on a 7 x 7 feature map, which results to the convolution layer output of size 5 x 5.*

3.4 Batch Normalization

During the training process of a CNN, the output of each convolutional block changes, as its trainable parameters (weights, biases) change as well. These changes affect the distribution of each next layer's inputs, causing each next layer to experience internal covariance shift. Covariance shift leads to a more complicated training process of the CNN, requiring smaller learning rates, adequate parameter initialization and more epochs for the filters to learn a function that generalizes well. This effect is amplified on the deeper layers of the CNN and eventually the rate of convergence to a proper function will decrease significantly [12].

The internal covariance shift of a CNN can be reduced with the implementation of the *Batch Normalization* process to the input vectors of each convolutional block. Batch Normalization is applied to a layer's input, when each value of the input is replaced by a value belonging to a distribution of the total input's values with zero center and unit variance. This new value is calculated on two steps. Initially, the mean value of a layer's d-dimensional (d batches) input x = ($x^{(1)}$ ... $x^{(d)}$) with each dimension's size of *m examples,* is subtracted by each x_i value of the input and then the result is divided by the squared variance of the input values:

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - mean(x^{(k)})}{\sqrt{Var(x^{(k)})^2 + \varepsilon}} \quad , \qquad \textbf{(20)}$$

$$mean(x^{(k)}) = \frac{1}{m}\sum_{i=1}^{m} \hat{x}_i^{(k)} \, ,$$

$$Var(x^{(k)}) = \sqrt{\frac{1}{m}\sum_{i=1}^{m} \left(\hat{x}_i^{(k)} - mean(\hat{x}_i^{(k)})\right)^2} \, ,$$

ε: arbitrarily small constant for numerical stability

Afterwards, the final input value $y^{(k)}$ to a layer is calculated with the use of the two parameters $\gamma^{(k)}$ and $\beta^{(k)}$, for each activation $x^{(k)}$:

$$y_i^{(k)} = \gamma^{(k)}\, x_i^{(k)} + \beta^{(k)} \qquad \textbf{(21)}$$

The second equation ensures that the representation of a layer does not change after its input's normalization process and adds two extra parameters to be learned, along with the existing model's parameters. The original layer's input can be obtained by setting $\gamma^{(k)} = Var(x^{(k)})$ and $\beta^{(k)} = mean(x^{(k)})$.

The Batch normalization transformation is a differentiable transformation. Therefore, it can be used in the backpropagation process, in order to back propagate the gradient of the loss function l and to calculate its trainable extra parameters γ and β *[12]*:

$$\frac{\vartheta l}{\vartheta \hat{x}_i} = \frac{\vartheta l}{\vartheta \hat{y}_i} \cdot \gamma \qquad \textbf{(22)},$$

$$\frac{\vartheta l}{\vartheta Var(x^{(k)})^2} = \sum_{i=1}^{m} \frac{\vartheta l}{\vartheta \hat{x}_i} \cdot (x_i^{(k)} - mean(x^{(k)}) \cdot$$

$$\cdot \frac{-1}{2} (Var(x^{(k)})^2 + \varepsilon)^{-3/2} \qquad \textbf{(23)}$$

$$\frac{\vartheta l}{mean(x^{(k)})} = \sum_{i=1}^{m} \frac{\vartheta l}{\vartheta \hat{x}_i} \cdot \frac{1}{\sqrt{Var(x^{(k)})^2 + \varepsilon}} \qquad \textbf{(24)}$$

$$\frac{\vartheta l}{\vartheta x_i} = \frac{\vartheta l}{\vartheta \hat{x}_i} \cdot \frac{1}{\sqrt{Var(x^{(k)})^2 + \varepsilon}} + \frac{\vartheta l}{\vartheta Var(x^{(k)})^2} \cdot$$

$$\cdot \frac{2\left(x_i^{(k)} - mean(x^{(k)}\right)}{m} + \frac{\vartheta l}{mean(x^{(k)})} \cdot \frac{\vartheta l}{\vartheta y_i} \qquad \textbf{(25)}$$

The benefits of the Batch normalization application to a CNN are the ability to use higher training rates, reduced need for Dropout regularization and the lower number of training steps needed for model's convergence.

3.5 Regularization

Regularization is a general term in ML and DL fields and is the implementation of various techniques that prevent from learning complex models and avoid overfitting phenomenon. These techniques can be applied to various stages of a CNN's training process, which allow for better results in CNN's generalization.

3.5.1 Dropout

Dropout is a regularization technique that is mainly used after the Fully connected layers of a CNN, and less after its pooling layers. Its goal is to create image noise augmentation, allowing the CNN to search a greater area of possible models. Dropout is implemented by randomly zeroing some of the input values of a layer (*Fig. 3.15*) with a probability p of a Bernoulli distribution. Actually, on each mini-batch, It excludes some neurons and their respective incoming and outgoing connections from the network's layers it is applied, during the training process. This procedure has as a result a smaller network with the same number of layers but with less neurons [13][14].

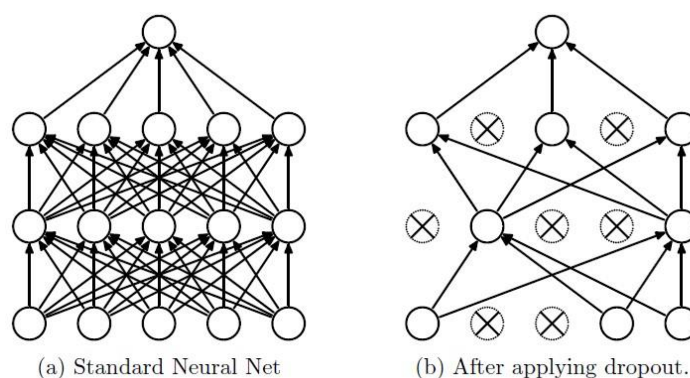


Figure 3.15*, (a) Fully connected layers, (b) After implementing Dropout with 0,5 random probability, which zeroes the values corresponding to the connections of the randomly chosen neurons*

Thus, dropout technique allows the exploration of a bigger number of different smaller (over the neurons/units total number) networks. These smaller networks, at test time, are combined in one network with smaller weights, where the weights from excluded connections participate with a factor of p (Fig. 3.16) [13][14].

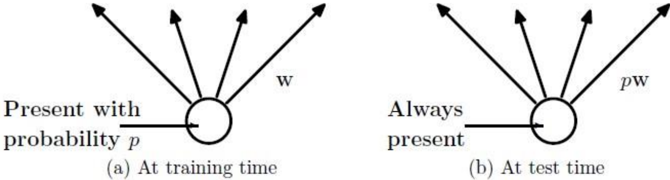


***Figure 3.16**, Network with a p dropout probability during training and the participation of dropped out weights with a factor of p, during testing*

3.5.2 L2 Regularization

L2 regularization [8][9] is a technique that is implemented at the cost function of the output of a CNN layer. It applies a weight decay by diminishing the layer's weights and preventing their excessive growth. In order to accomplish this task, it adds to the cost function J_o of the training set of size n, the sum of squared weights w, multiplied by a regularization parameter lamda (λ):

$$J = J_o + \frac{\lambda}{2n}\sum_w w^2 \qquad \textbf{(26)}$$

If the equation (16) is used in the backpropagation process, taking the partial derivatives, and applying the learning rule with a learning rate α, the new weights can be calculated:

$$\frac{\vartheta J}{\vartheta w} = \frac{\vartheta J_0}{\vartheta w} + \frac{\lambda}{n} w \qquad \textbf{(27)}$$

$$w = w - \alpha \frac{\vartheta J}{\vartheta w} = w - \alpha \frac{\vartheta J_0}{\vartheta w} - \alpha \frac{\lambda}{n} w =$$

$$= \left(1 - \alpha\frac{\lambda}{n}\right)w - \alpha\frac{\vartheta J_0}{\vartheta w} \quad \textbf{(28)}$$

The last equation reviles the fact that the larger a weight is, the more it is penalized and it is led towards zero without becoming ever zero. *Lamda* regularization parameter determines the excess of the regularization application and it takes positive values. If λ = 0, then no L2 regularization is applied. If λ is too large, then the weight decay will increase significantly and the model will lead to underfitting. Choosing the right value of λ will determine the efficiency of the L2 regularization. Eventually L2 regularization helps the network to generalize better and reduces its overfitting over the training set.

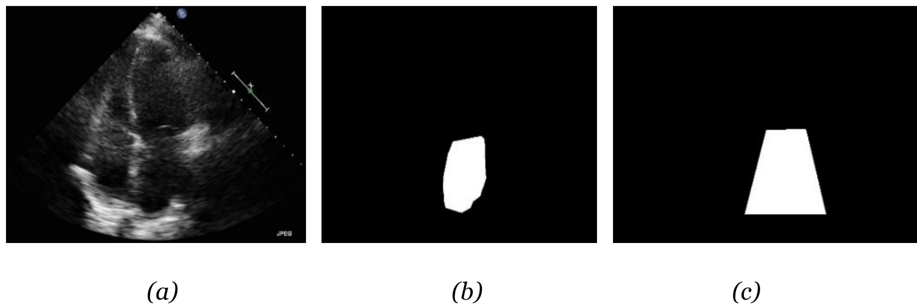
4 Segmentation with Unet

Segmentation and object detection are two tasks of computer vision, which goal is to find and precisely allocate the presence of different objects in an image, by trying to discover the exact boundaries of these objects in the image. Since DL CNNs are used to extract import features of an image, they have been also used for the specific feature recognition and allocation of different objects in images. Unet belongs to this family of the CNNs and has been initially developed for Biomedical Image Segmentation [17] .

4.1 Semantic segmentation

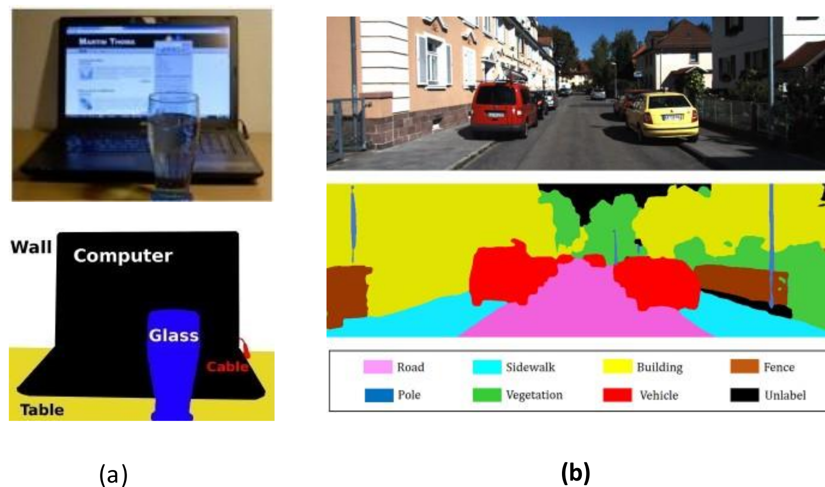
Semantic segmentation in computer vision and machine learning is the task that classifies each pixel of an image and assigns it to a certain semantic label of a category. It recognizes the existence of an object category inside the image and separates the object area of this category from the background, by precisely delineate the contours of this area with a different color [16] (*Fig. 4.1*).

Since semantic segmentation categorizes each pixel to a specific class, it is a classification supervised problem and the different classes that are used, define its design decisions. It can be a binary classification problem when the task is the separation of an object from its background like the segmentation of skin signs from the rest of the skin for skin cancer detection or the separation of the Left atria on a cardiac TTE image for helping its better enlargement assessment. Multi class segmentation can also be implemented, separating the different classes in an image, as scene understanding in self-driving car (*Fig. 4.2*). The number of classes affects the choice of the last activation function of a CNN. Usually, sigmoid function is used for binary semantic segmentation and softmax for multiclass segmentation.



(a) *(b)* *(c)*

***Figure 4.1**, Semantic segmentation on a TTE cardiac image using Unet: (a) Apical 4 chamber (A4C) view, (b) Precise Semantic segmentation mask over the cardiac LA, (c) Loosen semantic segmentation mask of the same area.*



(a) **(b)**

***Figure 4.2**, Semantic segmentation on an image with objects of different classes (a) objects on a table*

One characteristic design of CNNs, that implement the sematic segmentation task is their final layers. Pixel by pixel classification on the original image input, in order to separate the desired objects by background (and from each other for different class objects), dictates an alteration on CNN's architecture. The final fully connected layers are replaced by convolutional layers and the last layer is also a convolutional one, where its output feature map corresponds to an image representing the same scene as the input image (*Fig. 4.3*). Main difference of the output image, is that the output represents the masks of the areas of objects of different classes that separates them form the rest background of the scene.

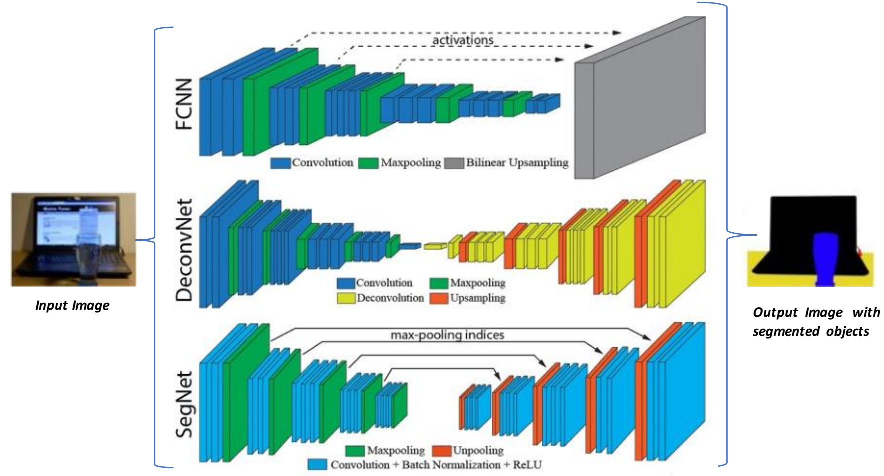


***Figure 4.3**, Semantic segmentation with different architectures. Output image is a representation of the masks of the different objects from the initial scene*

Performance evaluation of a semantic segmentation system is an import part, as it is for every machine and deep learning problem. If for L number of classes, we denote by $\boldsymbol{C}_{ij}$ the number of pixels with ground truth class i and predicted class j, $\boldsymbol{P}_j = \sum_i \boldsymbol{C}_{ij}$ the total number of pixels which were predicted in class j and $\boldsymbol{G}_j = \sum_{i=1}^{L} \boldsymbol{C}_{ij}$ the total number of pixels labelled with class i , the following metrics can be measured [16]:

a) Overall Pixel (OP) accuracy: OP = $\frac{\sum_{i=1}^{L} \boldsymbol{C}_{ii}}{\sum_{i=1}^{L} \boldsymbol{G}_i}$

b) Per-Class (PC) accuracy: PC = $\frac{1}{L} \sum_{i=1}^{L} \frac{\boldsymbol{C}_{ii}}{\boldsymbol{G}_i}$

c) Jaccard Index (JI), or mean Intersection over Union: JI = $\frac{1}{L} \sum_{i=1}^{L} \frac{\boldsymbol{C}_{ii}}{\boldsymbol{G}_i + \boldsymbol{P}_i - \boldsymbol{C}_{ii}}$ **(29)**

Jaccard Index calculates the average of the intersection over the union of the labelled segments. Thus, it is considered a standard metric for performance evaluation of a semantic segmentation system, as it evaluates the false alarms and the missed values per class, at the same time [15][16].

On the contrary, Overall Pixel accuracy is not a good evaluation metric, since its measure of the proportion of the correctly labelled pixels, in image inputs with

very imbalanced classes (e.g., Fig. 4.1), limits its objective results. Such inputs are usually those with a large background class, which may occupy over 70-75% of the total number of pixels. The Per-Class accuracy calculates the proportion of correctly labelled pixels for each class separately and then averages over the classes. In that case, false alarms are included in the background class. Thus, this metric can compensate and become capable of use in semantic segmentation problems where there is a small or no background class [16] (e.g., Fig. 4.2).

Another important aspect in a segmentation system is the choice of its loss function. Common loss functions for semantic segmentation is Focal loss, cross-entropy and Jaccard loss. Nevertheless, the latter losses, as is stated in Duque-Arias D. et al (2021) [17], were found not to perform well in some architectures, like Unet and SegNet. In the same paper, it has been proposed and tested a generalization of the Jaccard loss, the *Power Jaccard Loss*, which is able to penalize wrong predicted labels by increasing the weight of wrong predictions during training, thus improving the performance [17]:

$$J_p(y,\hat{y},p) = 1 - \frac{y \cdot \hat{y} + \varepsilon}{(y^p + \hat{y}^p - y \cdot \hat{y}) + \varepsilon} \quad \textbf{(30)}$$

Where p is the factor that determines the degree of penalization of the wrong predicted labels and constant ε prevents the zero division. For p = 1, Power Jaccard Loss is becoming the Jaccard Loss. For binary segmentation problems, where distinction between background and a subject is the goal, a suitable value for the power p has been tested to be the 1.75 [17].

4.2 U-net Architecture

U-net is a specific architecture of a fully CNN (*Fig. 4.4*), which has been developed mainly for segmentation and precise localization on biomedical images. Data augmentation on small annotated datasets, is a characteristic strategy of this CNN, achieving efficient and fast results. The last property makes U-net a good tool for image segmentation, since the access to such kind of annotated biomedical images is very restricted, due mainly to personal data

restriction laws. It consists of two parts, the contracting path and a symmetric expanding path [18].

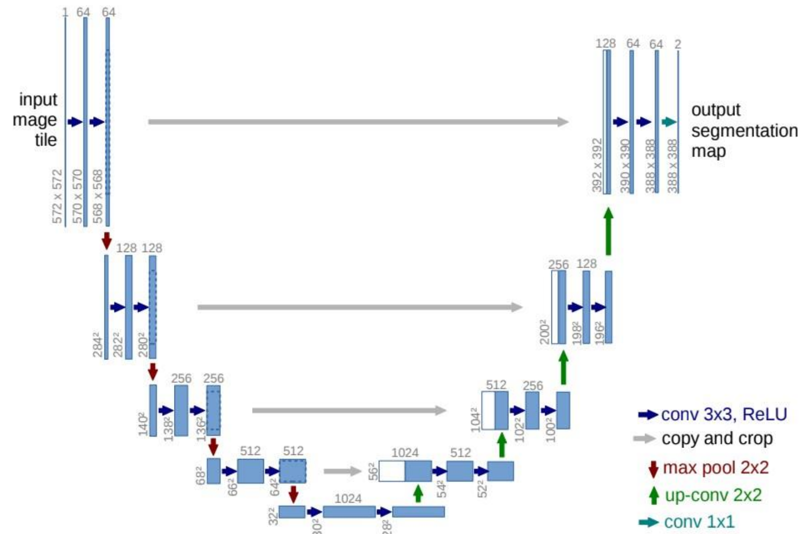


***Figure 4.4**, A characteristic U-net architecture*

The contracting path is responsible for feature extraction of the images. It consists of blocks with two unpadded convolutional layers of size 3 x 3, at each block. The number of channels for each convolutional layer of a specific block is half the number of the respective layers of the next block. The output of each convolutional layer is fed to a ReLU activation function. After each block, a max pooling layer of size 2 x 2 and stride 2 is applied to reduce output size of the block to half.

The expanding path is responsible for upscaling the output of the contacting path to the size of the original initial input. It consists of blocks of the same number with the contracting path, where each block is initialized with an up-sampling transpose convolution of size 2 x 2 and stride 2, which halves the channels of the previous block and doubles the size of the previous feature map. The upscaled output is concatenated with the respective output feature map of the contracting path. Each block is completed with two successive convolutional layers of size 3 x 3 with the same number of channels, followed by a ReLU activation function . The final layer of the U-Net is a one channel convolutional

layer of size 1 x 1, which outputs a segmentation map of size equal to the size of the initial input image. On the final segmentation map, a sigmoid function classifies each pixel to one of the two classes - background pixel or pixel belonging to the segmented sector. All the weights in the network are initialized from a Gaussian distribution with a standard deviation of square root(2/N), with N denoting the number of input units in the weight tensor [18].

Dropout and Data augmentation are two methods that are usually implemented in the U-net with a small training dataset, in order to boost its training process and avoid overfitting [18]. Dropout is used after the contracting path preventing the overfitting of the CNN, while data augmentation can make use of shifting, rotation, gray variations, and various other deformations, increasing the number of the available images, from the existing ones.

5 Generative Adversarial Network (GAN)

5.1 Basic GAN

Gan is a combination of two neural networks and belongs to the generative models (*Fig. 5.1*). The designation of the two networks is Generator *G* and Discriminator *D*. The goal of the GAN is, through its training, to make the Generator capable to produce fake data, as real as possible, from a Gaussian data probability input and the Discriminator to distinguish the real data from Generator's fake data [19].

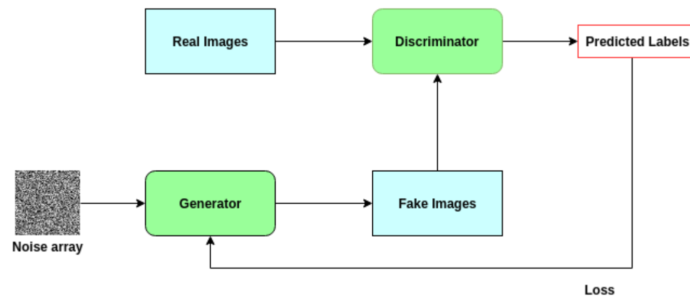


***Figure 5.1**, GAN basic semantic* GAN

In order to accomplish this task, during the training process, the two networks of the GAN are engaged into a mini-max, two-player game, where the Generator tries to produce fake, real like data, so as to deceive the Discriminator to classify them as real, while Discriminator tries to distinguish real data, coming from the data distribution, from the fake ones, coming from the Generator [19]. This game is described by the value function V(G,D) with equation [19]:

$$min_G \, max_D \, V(G,D) = E_{x \sim p_{data}(x)}[log \, D(x)\,] + E_{z \sim p_z(z)}\left[log\,\left(1 - D(G(z))\right)\right] \quad \textbf{(31)}$$

The Generator of a GAN can be any neural network of multilayer preceptrons or upscaling convolution layers (if data are images), which takes a noisy input *z*

from a Gaussian distribution p_z, transforms it through the network with parameters w_g, and outputs a fake data distribution p_g from its last layer. If G is the differentiable function that describes the Generator transformation, then the $G(z,w_g)$ is the mapping from the noisy distribution $p_z(z)$ to the data fake distribution p_g. The goal of the Generator is to find the right $G(z,w_g)$, so as its output distribution p_g to resample with the original data distribution ($p_g = p_{data}$) (*Fig. 5.2*). At this point the Generator produces fake images real enough to deceive the Discriminator to classify the fake image as real [19].

The Discriminator D can also be any neural network of multilayer preceptrons or convolution layers (if data are images), that can work as a classifier. In the classical form of the GAN, Discriminator maps its input to a scalar, which defines if the input belongs to the real data distribution p_{data} or to the Generator's fake data distribution p_g. In order to accomplish this task, during the training process, the Discriminator, with parameters w_d , tries to discover a function $D(x, w_d)$, which will maximize its output probability log$D(x)$ (the input x to belong to the real data distribution p_{data}). At the same time Generator tries to achieve its goal by maximizing the probability log$D(G(z))$, thus the Discriminator to classify the fake data $G(z)$ as real by assigning a big probability to it.

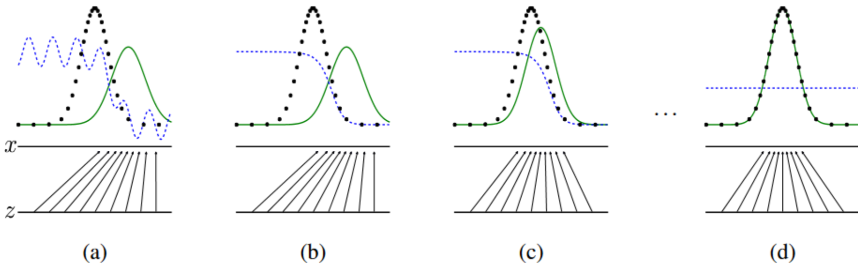


Figure 5.2, *GAN goal is, through training, the samples distribution p_g produced by the Generator (green solid line) to resample with the data distribution samples p_{data} (black dotted line) ((a) through (d)), while Discriminator discovers the distribution (blue dotted line) that discriminates fake from real samples.*

The basic algorithm that is implemented for a GAN, as it is described in Goodfellow I. J. et al (2014) [19] is the following:

GAN ALGORITHM:

for number of training iterations **do:**

for k steps **do:**

- Take a minibatch of m noise samples $\{z^{(1)}, z^{(2)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$
- Take a minibatch of m samples $\{x^{(1)}, x^{(2)}, \ldots, x^{(m)}\}$ from data distribution $p_{data}(x)$
- Train and update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[log\ D(x^i) + log\ \left(1 - D\left(G\left(z^{(i)}\right)\right)\right) \right]$$

end for

- Take a minibatch of m noise samples $\{z^{(1)}, z^{(2)}, \ldots, z^{(m)}\}$ from noise prior $p_g(z)$
- Train and update the generator by descending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} log\ \left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

end for

In the above algorithm the number discriminator's training steps k is a hyperparameter and its smallest possible value is 1. The goal of the algorithm in the discriminator's inner loop is to guide the discriminator to converge to a global optimum, which is $D^*(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g}$ for a fixed generator G. On the contrary, algorithm's outer loop goal is to train the generator so as to achieve the global minimum, where $p_g = p_{data}$ and the discriminator can distinguish the real data from fake, thus $D(x) = \frac{1}{2}$ [19].

5.2 Semi-Supervised Learning with GAN

Discriminator in a GAN, as a CNN it can be also used as a classifier. The intuition on this, is that, if we replace the sigmoid activation function at the last fully connected layer with the softmax activation, then we can train the Discriminator to distinguish an input data, not only between fake or real class but also in N additional classes, where the input may be classified [20]. Thus we can implement the semi-supervised learning with a GAN, while the Generator's task remains to try to output data close to the real data distribution.

In order to implement the semi-supervised training, the Discriminator keeps its main feature extraction structure as CNN except its last classiffication layer. For the classifying part, apart from fake or real classification, an activation function (sigmoid for binary classification, softmax for multiclass classification) is applied on the last layer, creating the supervised part of the Discriminator. Then the Discriminator is fitted with real data distribution p_{data} and their real respective labels and is trained to classify a data *x* into one of real classes N. It's goal on this step is to minimize it loss function *Lsupervised* (the negative log probability of the label, given that the data is real).

On a second step, after the Discriminator's feature layers, a sigmoid activation function is implemented to classify the input data as fake or real. For this part, firstly the unsupervised Discriminator is fitted with data from the real distribution p_{data}, labeled as real data and afterwards with data z from Generator's output distribution p_g, labeled as fake data. At this stage the Discriminator tries to maximize its loss function *Lunsupervised,* by maximizing probability *D(x)* and minimizing probability *D(G(z))*. The task of the Generator steal remains to produce as real as possible data. Finally the total loss function that Discriminator tries to minimize, during the semi-supervised training is[20]:

$$L = L_{supervised} + L_{unsupervised} =$$
$$= -E_{x,y \sim p_{data}(x,y)}[\log D(x,y)] - E_{x,y \sim p_{data}(x)}[\log D(x)]$$
$$- E_{x \sim p_g(z)}[1 - \log D(\mathrm{G(z)})]$$

Using semi-supervised GAN is very useful when we have limited labeled data, but large dataset of unlabeled data from the same distribution, which are, for sure, belongs to one of the distributions classes, but we do not know in which exactly. We this technique the Descriminator can be trained in the bigger unlabeled dataset to discover the specialized features of the whole data distribution, helping its with the small labelled dataset training task as a classifier.

6 Human Heart and TTE

Heart is the main organ of human's circulatory system. It works as a pump of the blood, which delivers oxygen and nutrients to the rest of the body's organs, for their proper and continuous function. Several malfunctions of the heart, such as its left atrial enlargement, can result in serious complications on a human's health [21]. Transthoracic echocardiography (TTE) is one of the most used and affordable methods to check and assess a heart's functional and structural integrity.

6.1 Heart anatomy

Heart is situated in the middle of the chest, between the two lungs, behind the sternum and slightly to its left side. It consists of four (4) main chambers, the left and right atria, and the left and right ventricles (*Fig. 6.1*). Atria is located to the upper part of the heart, while ventricles to the lower part. Each atrium communicates with the respective ventricle on the same side, through a valve. The left atrium is connected with the left ventricle through the mitral valve, while the right atrium communicates with the right ventricle through the tricuspid valve. Heart uses two additional valves for its external communication, the aortic and the pulmonary valve. The aortic valve is utilized by the left ventricle and is situated in its upper left part, providing access to the aorta, the main artery of the body. The pulmonary valve is utilized by the right ventricle and is located in its upper right section and allows communication with the pulmonary artery [21].

Heart's function is control by an internal electrical conduction system [21]. This system is managed by the human brain through the human neural system. It controls the rhythm and pace of the heartbeat. This heart's electrical system consists of the following [21]:

- Sinoatrial (SA) node: Controls the signal that regulates heartbeat.
- Atrioventricular (AV) node: it is responsible for signal communication between the upper chambers of the heart (right and left atria) and the lower champers (right and left ventricles) .

- Left bundle branch: supply the left ventricle with electrical signal
- Right bundle branch: supply the left ventricle with electrical signal

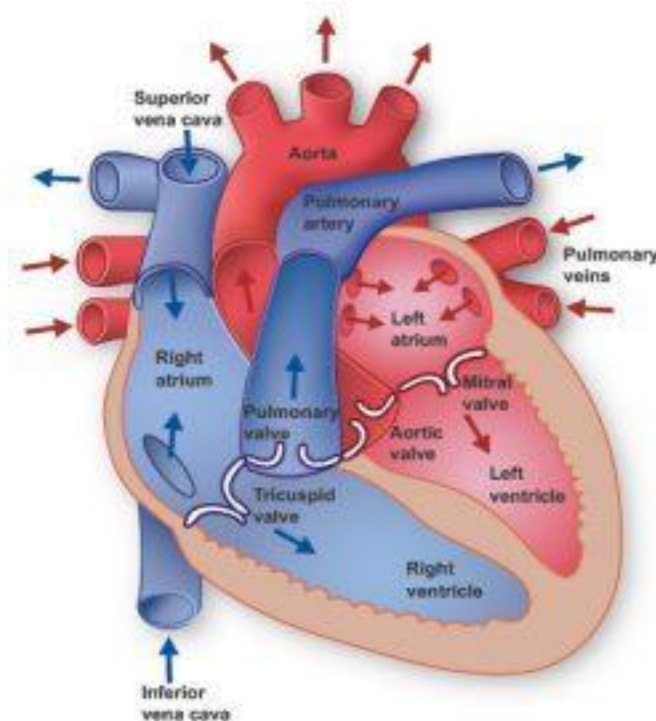


Figure 6.1, *Hearts anatomy. Blue areas are responsible for sending the blood to the lungs for oxygen enrichment, while the red ones receive the oxygen enriched blood and send it to the rest of the body*

Heart's function consists of two phases: the diastolic and the systolic phase. These two phases are controlled and triggered through electrical signals that are sent to the heart from the brain through the neural system and result in the cardiac cycle, which is a repeated sequence of alternating contractions and relaxation of the atria and the ventricles [21].

During the diastolic phase, heart fills with blood and its internal parts are conducting the following actions [21]:

- Left and right atria are expanded. Oxygen enriched blood is entered to the left atrium from the lungs through the pulmonary veins and oxygen poor blood is entered to the right atrium through the superior and

inferior vena cava. During the atria fulfillment, the mitral and the tricuspid valve are remaining closed.

- At some point, when the pressure in each atrium becomes greater than the pressure of the same side ventricle, the respective valve opens and allows a rapid flow of the blood from the atria to the ventricle.
- At a second stage of ventricles diastole phase, blood flows in the ventricles passively, due to pressure difference between the atria and ventricle cavities.
- The third stage of the ventricles diastole phase occurs during atria's contraction due to action potential from the sinoatrial node (SAN), where the remaining blood is actively pushed from the atria to the ventricles.

During the systolic phase, heart pumps the blood and its internal parts are conducting the following actions [21]:

- Left and right atria are forced to contract, when an electric signal is applied to the atria myocardium. This contraction increases the atria pressures and sends any remaining blood in the atria to the respective ventricle.
- Ventricle's systolic phase occurs during their contraction. After the signal on the atria myocardium is depolarized, it is applied, with a small delay, to the ventricles, forcing them to be contracted.
- During the ventricle's contraction, the pressure exceeds that of the atria, causing the mitral and the tricuspid valve to close.
- This contraction forces the aortic and the pulmonary valve to open. Then oxygen enriched blood is supplied from the left ventricle to the rest of the body, through the aortic valve and the aorta and from the right ventricle blood is sent to the lungs through the pulmonary artery for oxygen enrichment.

6.2 Cardiac Left Atrial (LA) Enlargement

LA enlargement is an anatomic variation to the size of the left atria and is a result of increased left atrial pressure for a significant period. Left atrial size has prognostic implications, and studies reveal that it can independently predict the development of clinically significant cardiovascular diseases and heart failure.

LA enlargement occurs when usually another dysfunction is present in the human heart [21]. LA's function is affected proportional to the severity of the Mitral's valve stenosis. In this case LA enlargement is developed and its ability to empty the blood to the LV is reduced. Mild to moderate LA enlargement occurs due to a progressive Mistral valve stenosis and severe LA enlargement is present in asymptomatic and symptomatic severe Mistral valve stenosis [21]. Mitral valve regurgitation leads also to LA enlargement, from a mild degree for the progressive Mitral regurgitation to a severe degree for asymptomatic and symptomatic Mitral regurgitation [21]. LV's malfunction during the diastolic and systolic phase can lead to LA enlargement. Moreover, LA fibrillation, hypertension can also cause the increase of the LA's size [21]. Nevertheless, an enlarged LA does not always mean a deviation from normal stage. Athletes and especially long-distance runners may develop an enlarged LA, without been a heart's malfunction [24]. The existence of LA enlargement dictates for further heart's abnormalities check [21].

According to Kou S. et al (2014) [31] the size of the LA cavity is irrelevant to the gender and the age of the individual. It is rather related to the individual's height and weight. Difference between males and females exists only because, in general, males are higher and heavier. In a male and a female with the same height and weight, the size of a normal LA cavity will not different greatly.

6.3 2D Transthoracic Echocardiography

6.3.1 Basic theory

The 2D TTE machines use the properties of the ultrasound waves, in order to display the tissue and the internal structure of the heart. Ultrasound waves are sound waves, thus mechanical vibrations, that cause the compression and decompression of a physical matter, through which are propagated. The main metrics of a soundwave are (*Fig. 6.2*) [22]:

- Frequency (*f*): the number of waves per second, measured in hertz (Hz) (1 Hz = 1/sec).
- Velocity of propagation v: the speed the wave is propagated through the body, measured in meters per sec (m/sec).

- Wavelength λ: the distance between two successive peaks, measured in millimeters (mm)
- Amplitude: height of the sound wavelength, measured in decibels (dB).

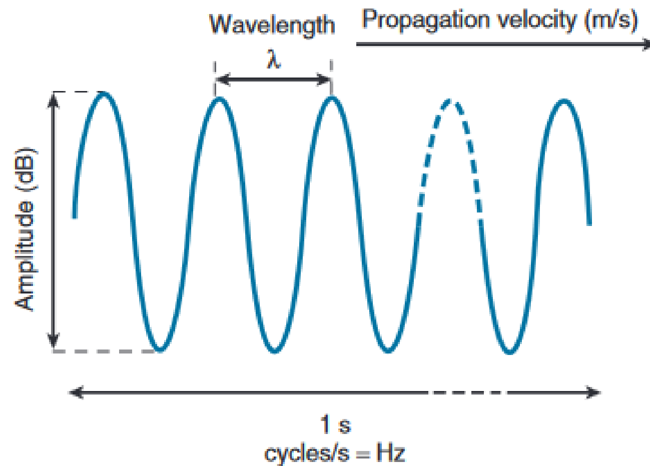


Figure 6.2, *Diagram of an ultrasound wave with its properties. (source [20])*

Ultrasound waves used in 2D TTE are in frequencies between 2.5 MHz and 3.5 Mhz, while their velocity in myocardium, valves, blood vessels, and blood is relatively constant at 1540 m/s [22]. The range of the amplitude used in 2D TTE is between 1 and 120dB. Additionally, the choice of the frequency determines directly the depth of the tissue the wave can reach. The higher the frequency is, the shorter is the depth of the tissue that can be displayed [22].

In 2D TTE the basic characteristic of the ultrasound waves, among others, is the reflection from the internal tissues (*Fig. 6.3*) [22]. Reflection occurs when the wave meets tissue boundaries and interfaces. The amount of reflection that a TTE machine receives, depends on two parameters, as it concerns the cardiac tissues: Difference in the tissue density and the angle of the arriving wave. When the tissue boundary's depth is larger than the arrived ultrasound wavelength, then this boundary acts as a mirror and returns the wave back to the machine as an echo]. In echocardiograms, in order to have an optimal display of the cardiac tissue, the angle of the arriving ultrasound wave over the examining tissue must be ninety degrees (90°) [22]. This perpendicular angle ensures that the received reflected wave will retain the energy of the initial one.

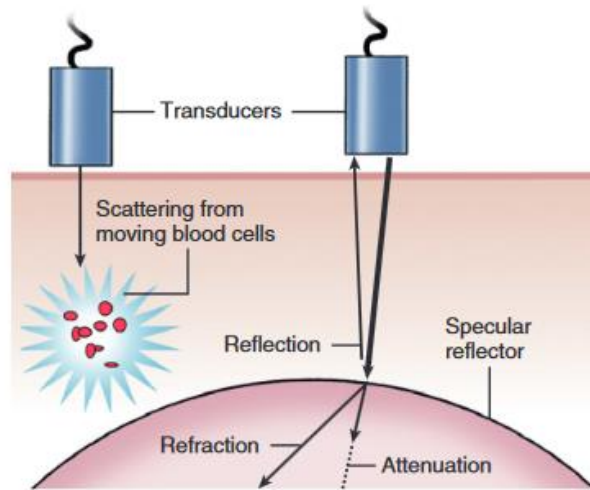


Figure 6.3, *Interaction between ultrasound wave and body tissue. In cardiac tissues reflection is the useful interaction that is used in 2d TTE*

6.4 2D TTE Standard Tomographic Views

Each view of the 2D TTE is described by the place of the machine's ultrasound head (acoustic window) on the thoracic area around the heart and the image plane over the heart's anatomy. According to this, a cardiologist may produce the following views, during a 2D TTE examination [22] (Fig. 6.4):

- Parasternal long axis (PLAX):
  This view gives access to the aortic and mitral valves, both ventricles, the left atrium, the aorta, and the coronary sinus.

- Parasternal short axis (PSAX):
  This view gives access to the aortic valve, the mitral valve level, the pulmonic valve, both ventricles and atria and the interatrial septum.

- Apical four-chamber (A4C):
  This view gives access to the mitral and tricuspid valves, to both ventricles and atria, the coronary sinus (posterior angulation from A4C) and to the aortic valve (anteriorly angulated A4C).

- Apical five-chamber (A5C):
  This view gives access to

- Apical two-chamber (A2C):
  This view gives access to the left ventricle and atrium and to the aorta (Modified A2C).

- Apical long axis:
  This view gives access to the aortic and the mitral valves and to the left ventricle and atrium.

- Subcostal four-chamber (S4C):
  This view gives access to the tricuspid valve, to both ventricles and atria and to the interatrial septum

- Suprasternal view:
  This view gives access to the aorta.

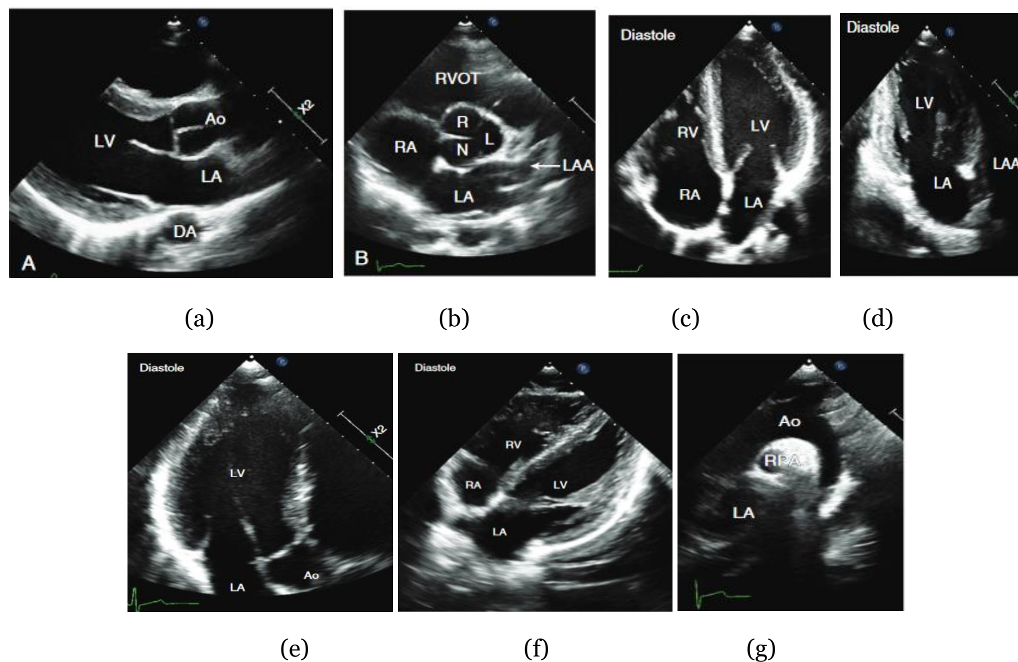


Figure 6.4, *2D TTE standard tomographic views on the diastolic phase: (a) PLAX, (b) PSAX, (c) A4C, (d) A2C, (e) Apical long-axis, (f) S4C, (g)* Suprasternal

6.4.1 Left Atria check with TTE

The assessment of the heart's left atria is accomplished through the A4C (Fig. 6.4(c)) and A2C (Fig. 6.4(d)) views combination of a 2D TTE. As stated in Galderisi M. et al (2017) [21] the proposed assessment metric for the LA is the

Biplane Volume of the left atria indexed by Body Surface Area (BSA). According to Lang M.R. et al (2015) [24] assessment of the left atria may be accomplished only by a single-plane view (A4C), if the acquisition of the two views cannot be obtained. The latest measurement assumes that the left atrial cavity is always a cylinder shape on the short axis plane, which is not true. The difference between the indexed volumes of the A4C and A2C is 1 to 2 mL/m^2 smaller in A4C view [24].

LA volume can be calculated using the disk summation technique (Fig. 6.5) by adding the volume of a stack of cylinders of height h and area calculated by orthogonal minor and major transverse axes (D1 and D2) assuming an oval shape [24]:

$$\frac{\pi}{4h} \sum D_1 D_2 \quad \textbf{(29)}$$

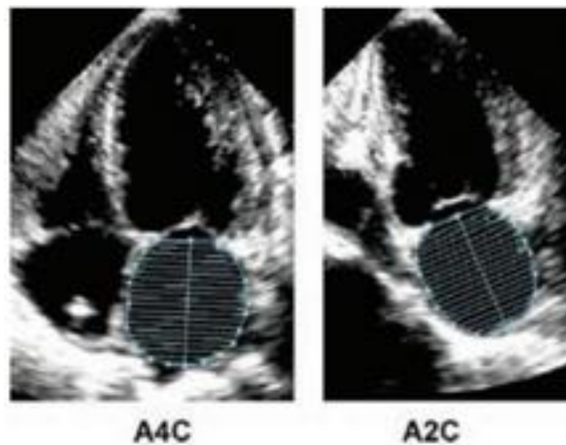


Figure 6.5, *Biplane Method of* disk summation technique *(source [26])*

BSA is a coefficient that is calculated by using the body height and weight of a person. It is used in clinical studies and is an indicator of fat-free mass. The most commonly used formulas for the BSA calculation are, the Du Bois formula [25] (weight W):

$$BSA = 0.007184 \times W^{0.425} \times H^{0.725}$$

and the Mosteller formula [24], which is simpler:

$$BSA = \frac{\sqrt{W \cdot H}}{60}$$

where the BSA unit is in m^2, weight (W) in *Kg* and height (H) in *cm*.

As it has been already mentioned, the size of the LA cavity is irrelevant to the gender and the age of the person and is related only to the body size [27]. Thus, the calculation of the Volume indexed by BSA allows establishing common limits for the two genders and preventing the need for gender specific limits due to general body size differences (according to Kou S. et al (2014) [27] mean BSA value for males is 1,94m^2 and for females 1.64m^2). Following this, the upper normal limit for 2D echocardiographic indexed LA volume is 34 mL/m^2 for both genders [23].

7 Experiments

In order to evaluate the capability of Deep Learning networks to assess the left cardiac atria enlargement, two methods have been deployed. The first one is a a U-net for the segmentation task over the LA area on A4C TTe images, and the second is a CNN for the classification of the LA size as normal or abnormal. A UNET – CNN pipeline of the two deep learning networks will be also evaluated.

7.1 Dataset

The dataset, that was used for the training and the evaluation of the networks, consists of 1151 A4C view images of TTE, taken with a Philips echo machine. Images were collected anonymously from a Greek Hospital, including only the statistical data of height, weight, year of birth and gender for a patient. For each A4C image, its respective assessment from an ECC certified cardiologist was applied. The assessment included the same A4C image with the left cardiac atrial area marked, the measured atria's volume and the cardiologist's assessment for the left cardiac atria's enlargement degree (Figure 7.1).

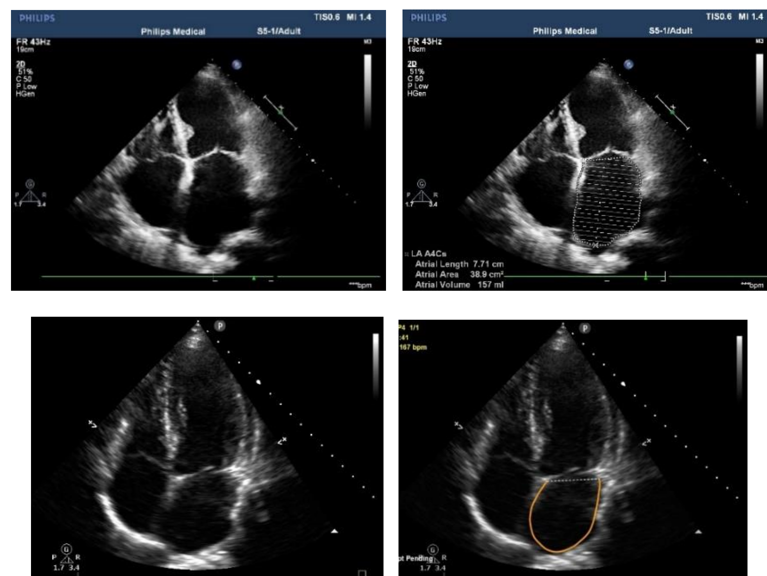


Figure 7.1, A4C view images. Left images are the echo machine's caption during the diastolic phase. Right images are the cardiologist's assessment with the marked left cardiac atrial.

7.1.1 Dataset statistics

The statistical data of the dataset include the gender , the date of birth for some of the samples, the height, the weight ,the volume and the assessment of their LA size. According to these, the 686 images belong to males and 464 to females of minimum age of 18 years old and maximum of 100 years. The average height and weight of the whole population is 170,1 cm and 74.9 Kg respectively. The LA's size has been assessed as normal for the 50,5 % of the population and 49,5% as abnormal.

The male population's age range is from 18 to 100 years old. The average male height and weight is 172,8 cm and 80.2 Kg respectively, while their LA size assessment is normal for the 49,7% and abnormal for the rest 50,3% . The female population's age range is 22 to 99 years old. Female average height is 166,2 cm, while the average weight is 67 Kg. The assessment of the female LA size revealed a 51,5% of normal size and the 48,5% of abnormal. Table 7.1 incorporates the above statistical information.

Table 7.1, *Dataset Statistics.*

	Population	Age Range	Average Height (cm)	Average Weight (Kg)	Average Volume Over BSA (kg/mm²)	Normal Cardiac Atria size	Abnormal Cardiac Atria size
Male	686 (59.86%)	18 - 100	172.8	80.2	36.3	341 (49.7 %)	345 (50.3 %)
Female	464 (40.3%)	22 - 99	166.2	67	35.5	239 (51.5 %)	255 (48.5 %)
Total	1151	18– 100	170.1	74.9	36	581 (50.5%)	570 (49.5 %)

7.2 Dataset Preparation

Before the implementation of the two deep learning methods, the dataset had to be prepared adequately. The images, both the original A4C and their respective assessments, had to be of 4:3 aspect ratio. Since, some of them were manipulated from a different assessment software, during the TTE, their aspect ratio was not the required one. In order to transform them in the desired aspect

ratio, we used the Microsoft Photo software of the Windows 10 OS to crop the desired area of the images (*Figure 7.2*). The resolutions of the dataset were the 768 x 1024 or the downscaled 600 X 800 for the cropped ones.

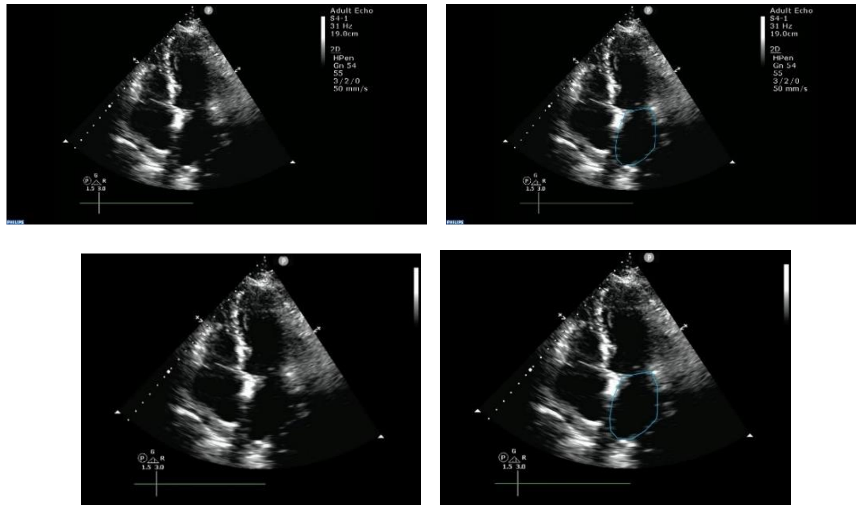


Figure 7.2*, Top images: produced from the TTE software. Bottom images: cropped in 4:3 aspect ratio and downsized to 600 x 800 resolution*

Our implementation on the pipeline U-Net – CNN, requires the existence of the mask for each image, over the left cardiac atria region. Each mask was used as the label of the respective image during the U-Net training and evaluation, and as a mean to filter the initial image, so as only the left cardiac atria region to be provided as an input to the CNN, or only the mask, for its classification as normal or abnormal. The production of the masks was accomplished with the use of the VGG Image Annotator (VIA), which is a standalone manual image annotation software [30]. VIA is based on HTML, Javascript and CSS and runs in a web browser.

Initially, for each A4C image, with the use of VIA (Figure 7.3), we produced the annotation area of two different kinds of masks. One mask includes an extended area around the left cardiac atria area and was used to isolate the area that will be fed to the CNN's input. The other kind of mask follows the annotated area of the left cardiac atria made by the cardiologist, on the assessment image. On the latter, the input image to the CNN will be directly the mask. The annotated masking areas were saved as *json* files.

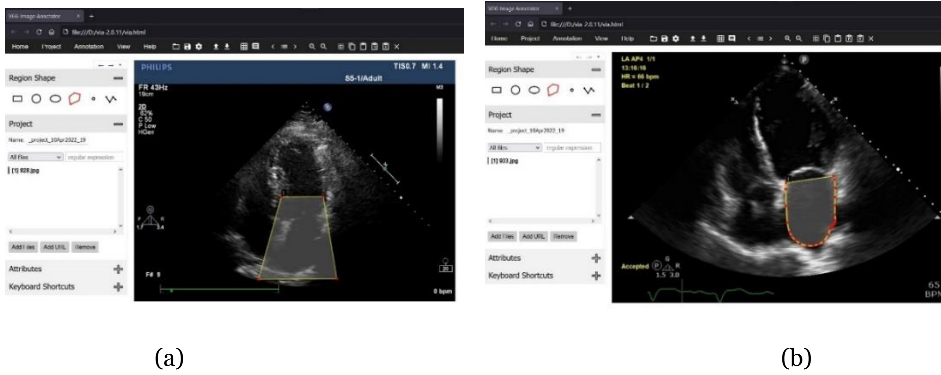


Figure 7.3, *Annotation the areas to pro of duce the corresponding masks with the VIA software: (a) Extended mask, (b) original mask from cardiologist assessment image*

The annotation of the masking areas followed the creation of the respective masks of images. For this task, an internal script of VIA software, written in Python programming language, was used. This script was fed with the locations of the original image and the respective json file with the right masking annotations, and produced the desired masks of the same size with the original images, at the coordinates of the left cardiac atria (*Figure 7.4*).

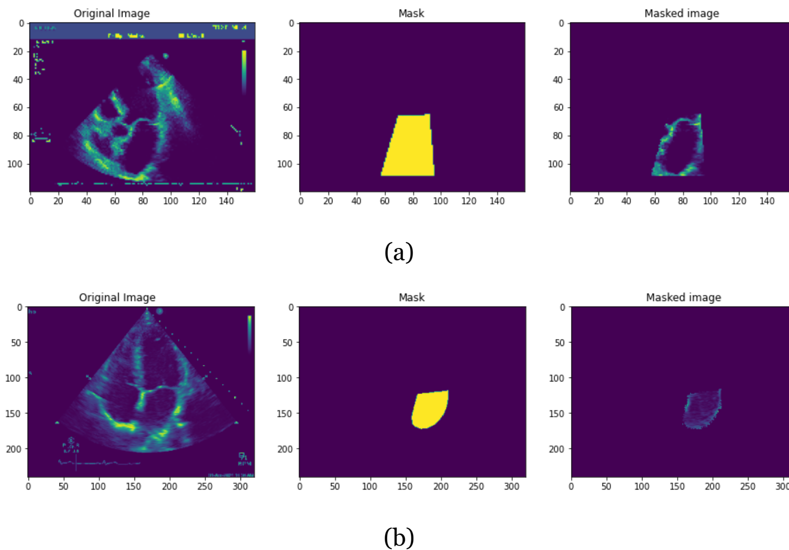


Figure 7.4, *The final masks (middle images) and their filtering over the original image (right images)(a) Extended mask, (b) Original mask*

The CNN, for its training and evaluation, will use two different datasets. The one will make use of the extended masked images as input image. These masked images are produced inside the code of our implementation. The other will use the original mask as input. The reason for the latter decision, as it is demonstrated in Figure 7-4, is that the masked images of the original masks have very little information for the CNN to manipulate. So, CNN's capability to discover the right patterns of the LA, in order to classify it efficiently, is reduced. Thus, we made the decision to use the original mask, hopping CNN will discover and simulate the least squares formula for the atria's volume measurement and to accomplish an efficient classification.

Additionally, to the input images of the CNN, for both datasets, we collected the aforementioned statistical data for all images in a csv file, in order to calculate the label of the atria as normal or abnormal. Then programmatically, for each image, we use the height and weight of its sample to calculate the BSA coefficient according to Mosteller's formula. Next, we calculate the Volume over BSA number (Indexed Volume) for each image. If the Volume over BSA is under 34 [25], then the image's atria is labeled as normal, otherwise it is abnormal.

Finally, for all the above models, the dataset of 1151 images is splited in a training set of 575 images, a validation set of 288 images and a test set of 288 images.

7.3 Methods

For the prepose of our experiment, our Unet and CNN and GAN have been implemented with the Tensorflow API and the code has been built to run o. The networks have been trained and evaluated on the following scenarios, as it concerns the kind of masks that were used, the image resolution and the use of augmentation techniques over the image datasets:

1. Unet:
    a. Extended mask dataset: The resolutions of 240 x 320 and 120 x 160 for image-mask dataset were used and for each resolution, a dataset without image augmentation and one with image augmentation.

b. Original mask dataset: The resolutions of 240 x 320 and 120 x 160 for image-mask dataset were used and for each resolution we used a dataset without image augmentation and with image augmentation

2. CNN:
   a. Extended mask dataset: The resolutions of 240 x 320 and 120 x 160 for masked image dataset were used and for each resolution, a dataset without image augmentation and one with image augmentation
   b. Original mask dataset: The resolutions of 240 x 320 and 120 x 160 for the original masks dataset were used and for each resolution we used a dataset without image augmentation and with image augmentation
3. GAN:
   For GAN we used the resolution of 130 x 130 as Generators output and Discriminators input. The input images were the A4C images without any masking.

The segmentation task of the A4C images was accomplished with the use of a modified Unet of Madani A. et al (2018) [2] (*Fig. 7.5*).

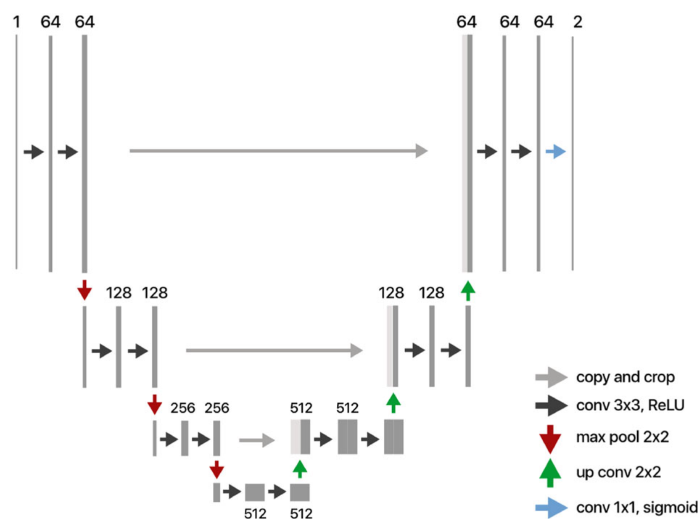


***Figure 7.5**, Implemented Unet architecture*

The applied modifications were the addition of a rescaling layer for the images in the range of (0,1), an augmented layer for random contrast and the use of same padding to all the convolutional layers, in order the output mask resolution to resemble with the input's resolution. Dropout regularization has been applied after the downscale part of the U-Net. In addition, the extra applied image augmentation included random rotation, width and height shifts, shear, zoom and brightness alterations. In order to maximize the use of the graphic card, batch size of 8 was used with the resolutions 240 x 320 and 16 with the resolutions of 120 x 160. The max training epochs of the U-Nets were 300 with a patience of 50 epochs and initial learning rate of 0.0001. The chosen loss function for the Unet was the *Power Jaccard Loss* with the power *p* of 1.75 [17] and the *Jaccard index* [16] as its metric. The model was evaluated with the test Jaccard index and F1 score over the pixels amount. For All tests a GTX 1050 GPU with 4GB ram was used.

The basic CNN architecture is the one used in Madani A. et al (2018) [2] (*Figure 7.6*). All the convolutional layers used kernels of size (3,3), ReLU as activation function, and same padding. For the resolution 240 X 320, the first two convolutional layers used a (7,7) kernel. Alterations from the above architecture have been implemented, such as the replacement of the Average pooling layers with Max pooling layers after each convolutional layer of kernel size (2,2), Dropout of 0.4 after the flatten layer and the fully connected layers and the standard weight initialization for the fully connected layers. The training epochs were 1000 with a patience of 50 for models with no data augmentation and 2500 for models with data augmentation, while the initial learning rate was of 0.0002. Binary cross entropy was used as the loss function and binary accuracy as the metric of the model. Model was evaluated with the test accuracy, F1 score and AUC-ROC score.

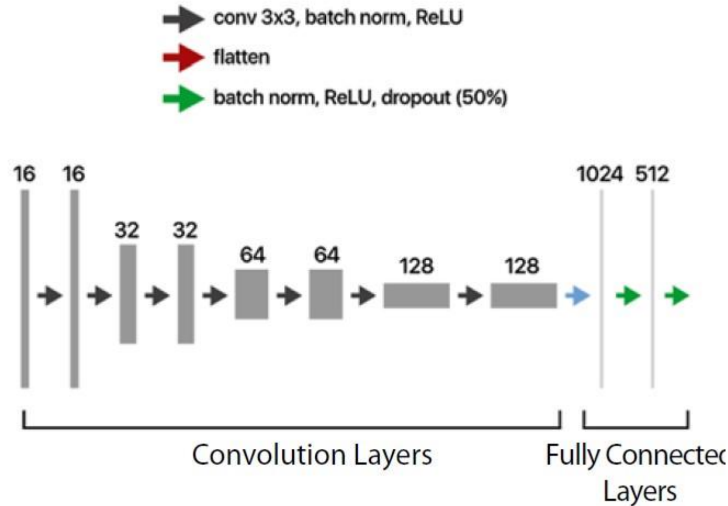


***Figure 7.6**, Implemented CNN architecture*

Finally, a Unet-CNN pipeline will be evaluated over the test set. From each category of extended and precise masking will be used the CNN with the best performance according the *F1* score and *AUC-ROC* score. For the selected CNN, the respective Unet will be used to produce the predicted masks. In the case of the extended masking, the predicted masks will be used over the test set images to produce the masked-image test set, which will be fed to its CNN for the classification task (*Fig. 7.7*). The predicted masks from the precise masking Unet will be fed directly to the respective CNN (*Fig. 7-8*).

All our models used the dataset with splits of 50% for training , 25% for validation and 25% for testing. We use the same seed for the splits in order to ensure that all models will use the same unseen test set for their evaluation. Thus, training set consists of 575 images and validation and test sets of 288 images each.

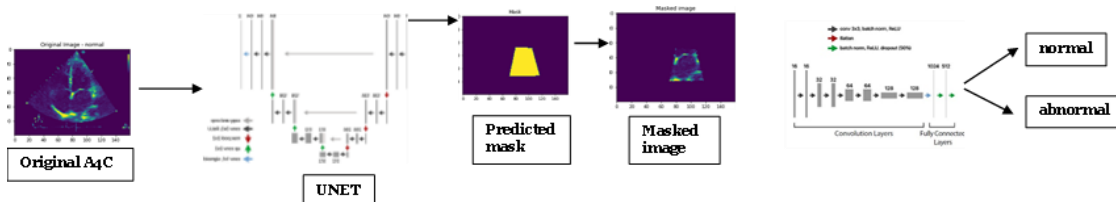


***Figure 7.7**, Unet - CNN pipeline for the extended masking models*

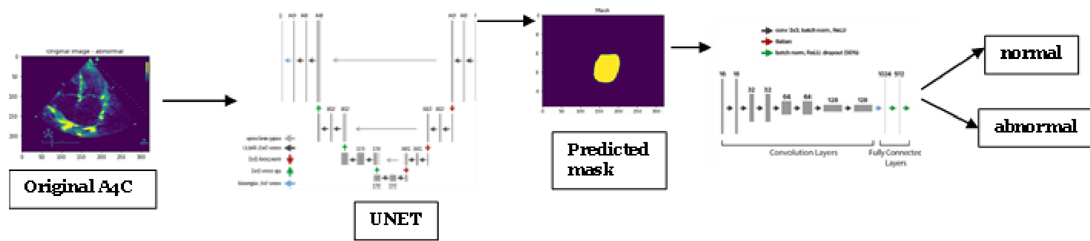


***Figure 7.8**, Unet - CNN pipeline for the precise masking models*

The semi-supervised task was implemented with a modified SGAN of Madani A. et al (2018) [2]. In our architecture the Generator (Fig. 7.9) receives as an input a Gaussian noise layer of size 1 x 1x 100, which gets passed through eight (8) conv-transpose block layers, and outputs images of size 130 × 130. Each of the first seven conv-transpose block consists convolutional transpose upscaling layer followed by a batch normalization layer. The upscaling layers use a (3,3) kernel with stride 2 and ReLU as activation function. The last upscaling layer uses the tanh as activation function.

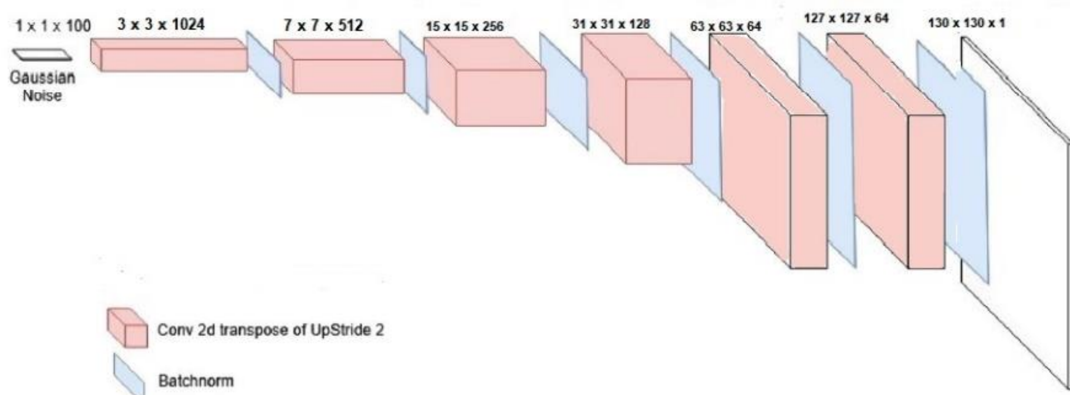


***Figure 7.9**, Generator of the Semi-supervised GAN*

SGAN's Discriminator consists of four convolutional blocks followed by a max pooling layer of size (2,2) and stride 2, and a flatten layer and a fully connected layer with LeakyReLU activation function. Each convolutional block consists of three convolutional layers of kernel size (3,3) followed by batch normalization layer and a LeakyReLU activation function. The first two conolotional layers use a stride of 1, while the third ones of stride 2. After each convolutional block and fully connencted layer a dropout layer of 0.5 is applied (Fig 7.10).

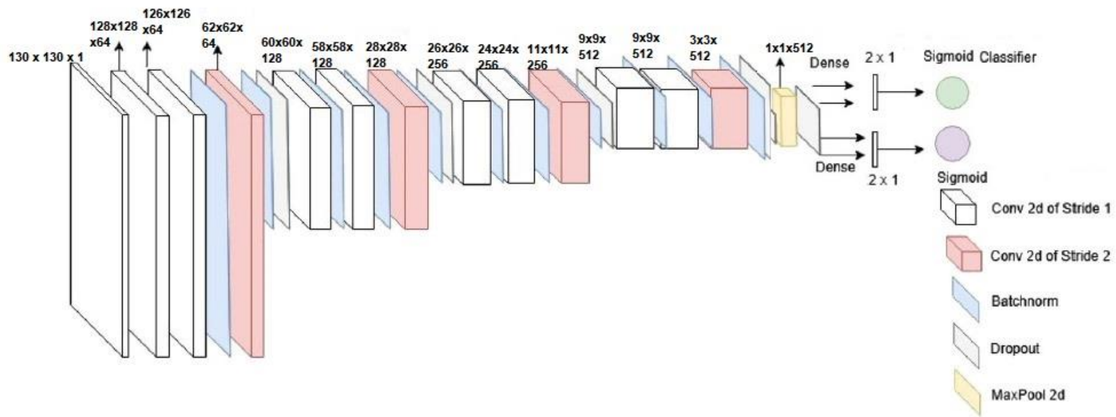


Figure 7.10, *Discriminator of the Semi-supervised GAN*

For the supervised and unsupervised part of the discriminator we used binary crossentropy as the loss function and on their respective output layer the sigmoid function. For the GAN we used a mean square error loss between the second last layer of the discriminator for our fake image and for an unlabeled image. For the SGAN training, in one iteration, we supply first a minibatch of 32 images and labels to the supervisd part of the Discriminator, then a same sized minibatch of real images with their labels of ones to the unsupervised part of the Discriminator, following a same sized minibatch of fake images with their labels of zeros, also to the unsupervised part of the Discriminator. The iterasion is completed with the Generator training, while the Discriminator stays untrainable. The dataset of 1151 unmasked images was splitted in three parts. 207 images were used for the supervised training of the Discriminator, 828 images as the unlabeled dataset and 116 images as Test set for the evaluation of the superivised Discriminator on the classifying the cardic LA as normal or abnormal,

7.4 Results

The results of the training and evaluating the U-net and the CNN models in the task of the segmentation of an A4C image of a TTE over the LA area and the classification of the LA's size as abnormal or normal reveal the significance of

the use of the data augmentation techniques in small datasets, such as those with medical data.

7.4.1 Unet

The results of the Unet model over the different image size inputs per masking type, showed the importance of the data augmentation techniques during Unet's training, which is one of U-net's main characteristics according to Ronneberger et al (2015) [18]. On all cases, data augmentation helped model to achieve better results on the segmentation task (*Tables 7.2, 7.3*). Models without data augmentation have high variance and are overfitting to the training data, failing to generalize well. On the contrary, producing and providing the model with more abstract data through data augmentation, allow the models to examine a wider set of data distributions around the original distribution of the initial training set, resulting in more generalized models, which are capable to predict much better the previously unseen test set.

A physical examination of the produced masks on the test set, for this model, without data augmentation and with data augmentation, reveals that model without data augmentation, in many cases, failed in detecting the exact position of the LA in the image, even if the size of the mask was close to the ground truth (Fig. 7.11).

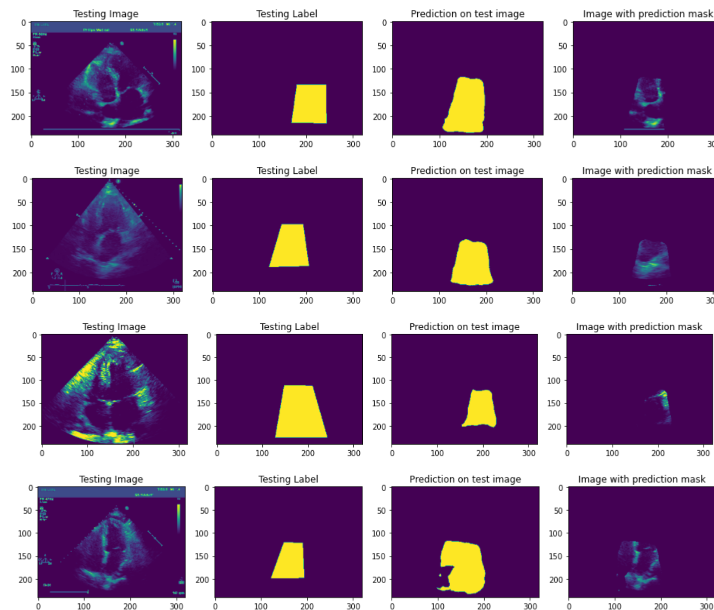


***Figure 7.11**, Segmentation results of 120 x 160 test set images from Unet with no data augmentation (extended masking)*

On the other hand, the models with data augmentation were able to detect the area of the LA and to draw it with the proper mask (Fig. 7.12). Any deviation of the latter from the ground truth, was just a bigger mask around the crucial area of the LA.

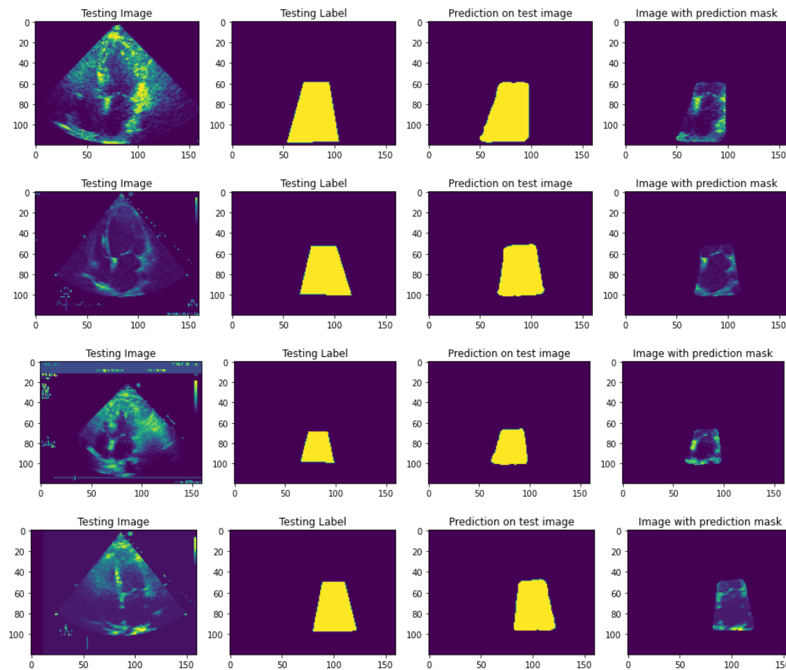
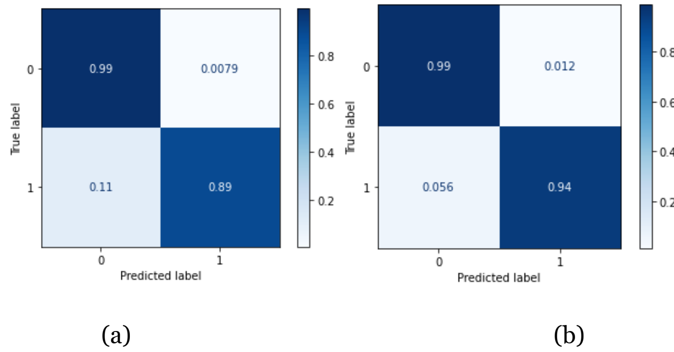


Figure 7.12, *Segmentation results of 120 x 160 test set images from Unet with data augmentation (extended masking)*

For the extended masking segmentation task, the model that was trained with dataset of 120 x 160 image size achieved better results, with 91% F1 score on masking pixels and 83.9 % Jaccard Index score over the test set predictions. Moreover, the per pixel normalized confusion matrices of the two resolution models, show also the better performance of the model with 120 x 160 resolution (Fig. 7.13). The difference is significant on the pixels that represent the mask (class 1), where the model with 120 x 60 achieves a 94% recall and 87% precision score, while that with 240 x 320 achieves an 89% recall and 91% precision score.



(a) (b)

Figure 7.13*, Unet's per pixel normalized matrices with DA of test set images (extended masking): (a) 240 x 320, (b) 120 160 (0:abnormal , 1:normal)*

Table 7.2*, Unet Extending Masking Results*

Extended Masking				
	240 x 320		120 x 160	
	Plain Data	Data Augmentation	Plain Data	Data Augmentation
Training Jaccard Index	94.4 %	84.9 %	85.7 %	87.4 %
Validation Jaccard Index	82.9 %	82.9 %	83.1 %	83.6 %
Test Jaccard Index Predictions	40.5 %	81.9 %	40.4 %	83.9 $
F1 score Masking Pixels	57.0 %	89.0 %	57.0 %	94.0 %

For the precise masking segmentation task, A physical examination of the produced masks on the test set, for the model, without data augmentation (Fig. 7.14) and with data augmentation (Fig. 7.15), revels the same behavior with extended masking model, where the model with data augmentation managed to detect the area of the LA and to draw it with the proper mask. Additionally, the model without data augmentation, in many cases, failed to discover the exact shape of the ground truth mask.

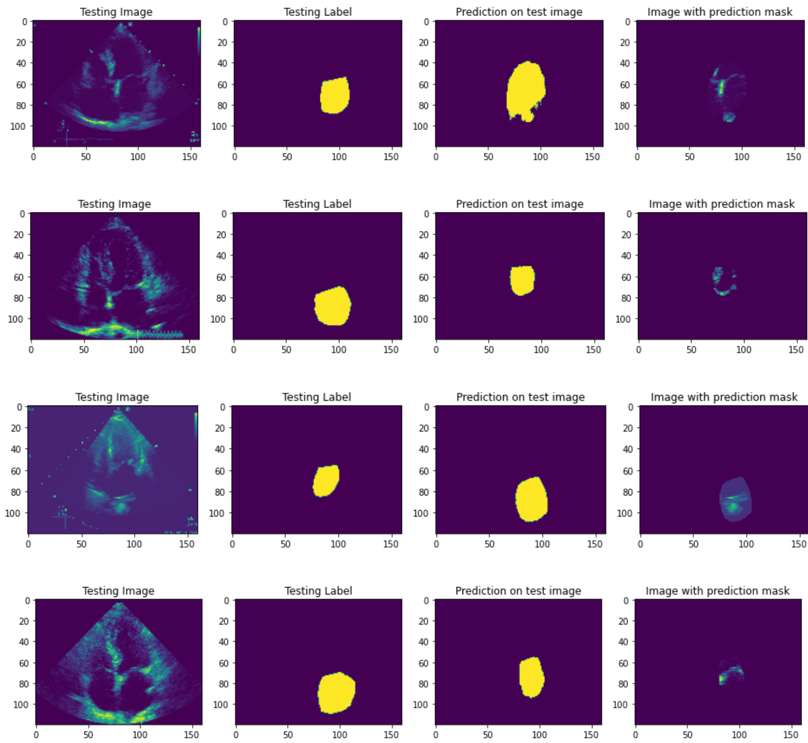


***Figure 7.14**, Segmentation results of 120 x 160 test set images from Unet without data augmentation (precise masking)*

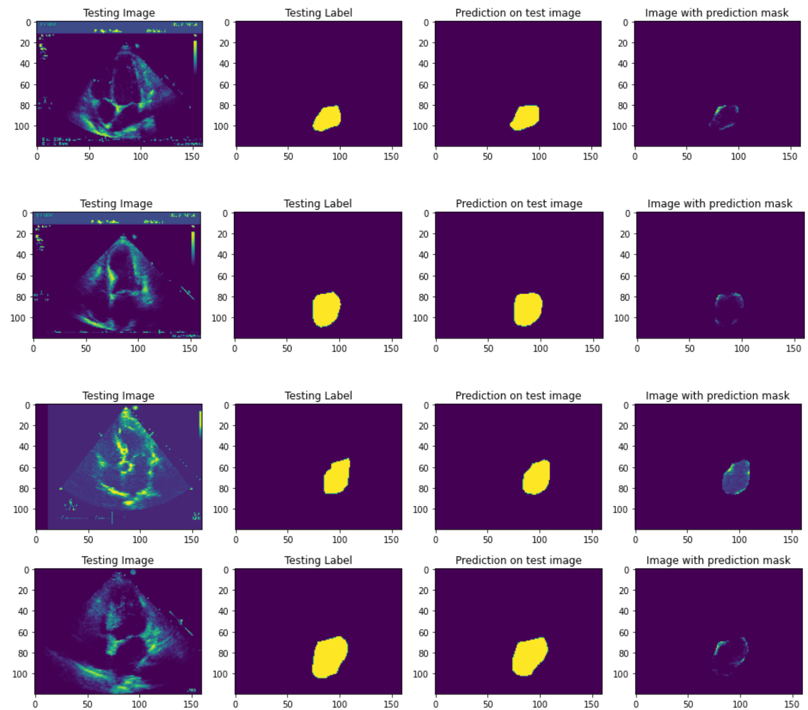
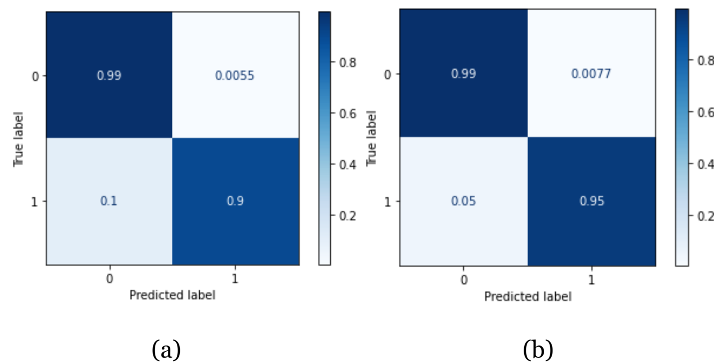


***Figure 7.15**, Segmentation results of 120 x 160 test set images from Unet with data augmentation (precise masking)*

The model that was trained with dataset of 120 x 1620 image size, achieved also slightly better results, with 88% F1 score on masking pixels and 81.32% Jaccard Index score over the test set predictions.

The per pixel normalized confusion matrices of the two resolution models, show also the better performance of the model with 120 x 160 resolution (Fig. 7.16). The difference is not very significant on the pixels that represent the mask (class 1), where the model with 120 x a60 achieves a 95% recall and 82% precision score, while that with 240 x 320 achieves an 90% recall and 85% precision score.



(a) (b)

***Figure 7.16**, Unet's per pixel normalized matrices with DA of test set images (precise masking): (a) 240 x 320, (b) 120 x 160(0:abnormal , 1:normal)*

***Table 7.3**, Unet Precise Masking Results*

Precise Masking				
	240 x 320		120 x 160	
	Plain Data	Data Augmentation	Plain Data	Data Augmentation
Training Jaccard Index	81.5 %	83.2 %	85.2 %	83.0 .0%
Validation Jaccard Index	78.0 %	78.35 %	80.9 %	80.1 %
Test Jaccard Index Predictions	30.0 %	78.32 %	29.9 %	81.3 %
F1 score Masking Pixels	45.0 %	87.0 %	45.0 %	88.0 %

7.4.2 CNN Results

The results of the CNN models over the different image size inputs per masking type, showed again the importance of the data augmentation techniques during CNN's training. On all cases, data augmentation helped model to achieve better results on the classification task (Tables 7.3, 7.4). However, the results are still not satisfactory, since even the models with data augmentation have low metrics. One reason is may be the low resolution for our task. As it is observed in the results, for the data augmented models a comparison between the two resolutions reveals that the 240 x 320 resolution gives better results. The overfitting is less, while *F1* and AUC_ROC scores are bigger.

For the extended masking classification task, the model that was trained with dataset of 240 x 320 image size achieved close results with that of 120 x 160 resolution, with 75% and 74% F1 score for the abnormal and the normal class respectively, 74% accuracy over the test set predictions and an 74% AUC-ROC score (*Table 7.3*). The normalized confusion matrices of the two resolution models, show also the almost similar performance of the model with 120 x 160 resolution on both classes, achieving better result for the normal LA class (*Fig. 7.17)*. The model of 240 x 340 achieves a 77% and 71% recall score for the abnormal and the normal class respectively, while that of 120 x 160 achieves an 78% and 64% recall score for the abnormal and the normal class respectively.

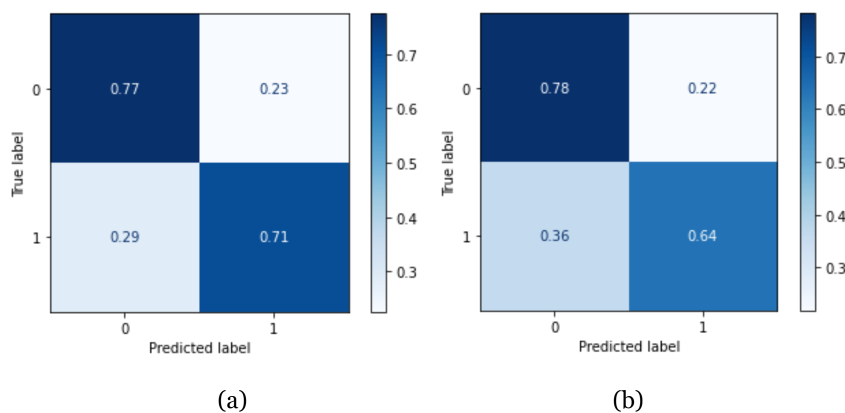
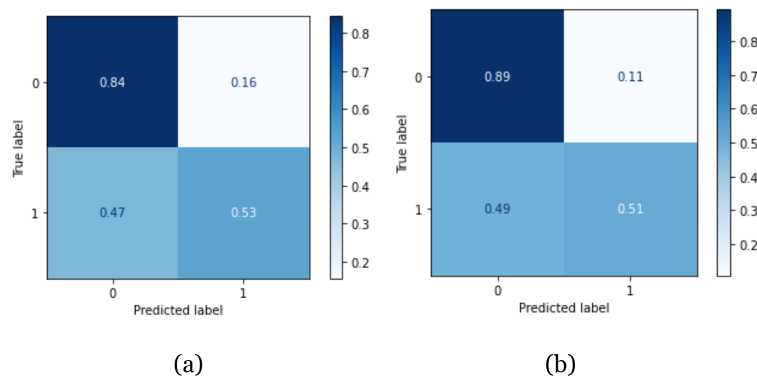


Figure 7.17*, CNN's normalized matrices with DA of test set images (extended masking): (a) 240 x 320, (b) 120 160 (0:abnormal , 1:normal)*

Table 7.3, *CNN Extending Masked Images Results*

Extended Masking				
	240 x 320		120 x 160	
	Plain Data	Data Augmentation	Plain Data	Data Augmentation
Training Accuracy	100 %	77.9 %	100 %	75.2 %
Validation Accuracy	71,7 %	73.1 %	69.1 %	72.9 %
Test accuracy	48.0 %	74.0 %	49.0 %	71.0 %
F1 score abnormal normal	50.0 % 45.0 %	75.0 % 74.0 %	52.0 % 45.0 %	73.0 % 69.0 %
AUC-ROC	48.0 %	74.0 %	49.0 %	71.0%

For the precise masks classification task, the model that was trained with dataset of 120 x 160 mask size achieved also close results with that of 240 x 320 resolution, with 75% and 63% F1 score for the abnormal and the normal class respectively, 70% accuracy over the test set predictions and an 70% AUC-ROC score *(Table 7.4)*. The normalized confusion matrices reveals also the balance on the performance for the models on both classes (*Fig. 7.18)*. The model of 120 x 160 achieves a 89% and 51% recall score for the abnormal and the normal class respectively, while that of 240 x 320 achieves an 84% and 53% recall score for the abnormal and the normal class respectively.



(a) (b)

Figure 7.18*, CNN's normalized matrices with DA of test set images (precise masking): (a) 240 x 320, (b) 120 160 (0:abnormal , 1:normal)*

Table 7.4, *CNN Precise Masks Results*

Precise Masking				
	240 x 320		120 x 160	
	Plain Data	Data Augmentation	Plain Data	Data Augmentation
Training Accuracy	100 %	73.7 %	100 %	73.7 %
Validation Accuracy	69.9 %	77.6 %	71.7 %	79.4 %
Test accuracy	49.0 %	69.0 %	50.0 %	70 %
F1 score abnormal normal	53.0 % 45.0 %	73.0 % 63.0 %	52.0 % 48.0 %	75 % 63 $
AUC-ROC	49.0 %	69.0 %	50.0 %	70.0 %

7.4.3 Unet - CNN Pipeline Results

In the case of the extended masking models, the CNN with the best results is the 240 x 320 model, which will be combined with the respective 240 x 320 resolution Unet model. The implementation of this pipeline reveals a slightly reduce performance of the CNN model. The cause for this is the Unet model, which in some cases failed to predict the masks well (*Fig. 7.18*). In Figure 7.19 the four images reveal the importance of the quality sampling during the TTE procedure for implementing LA segmentation. The Unet failed to discover the LA on the A4c image. Nevertheless, extended masking pipeline classification reached an *F1* score of 72 % and 74 % for the abnormal and normal class respectively, with 73 % *AUC-ROC* score and 73% accuracy on test set, reducing CNN's score only by one unit. Its normalized matrix is presented in Figure 7.20(a).

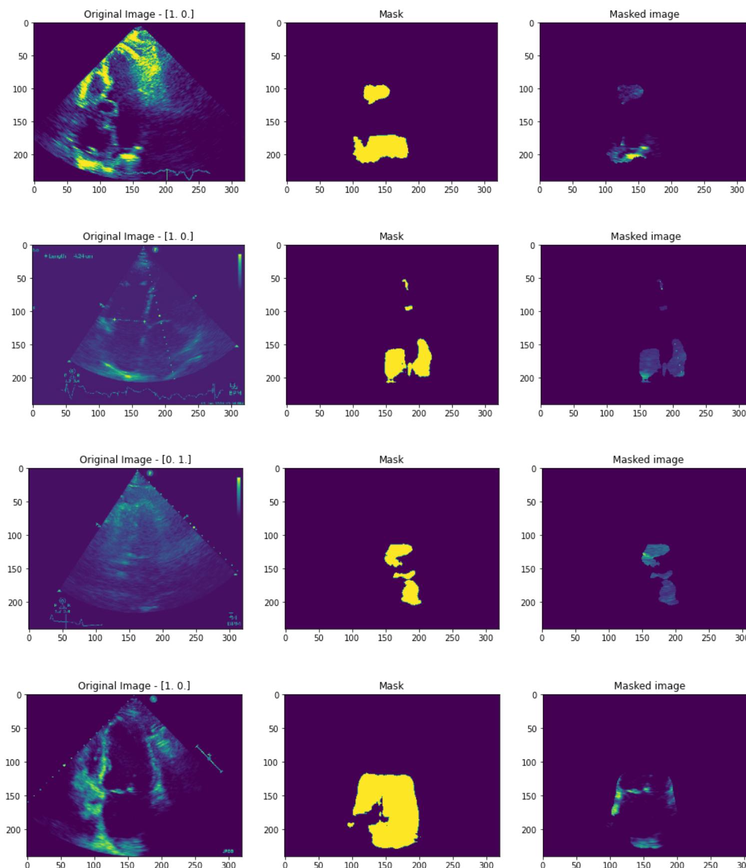
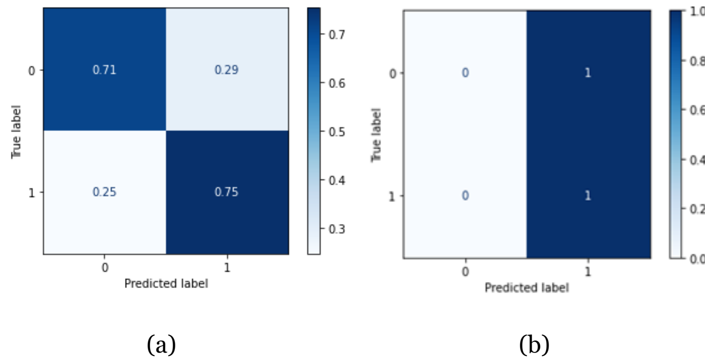


***Figure 7.19**, Segmentation results of 240 x 320 test set images from Unet with data augmentation (extended masking 240 x 320)*

In the case of the precise masking models, the CNN with the best results is the 120 x 1600 model, which will is combined with the respective 120 x 160 resolution Unet model, which performance is also the best in its category. The implementation of this pipeline reveals a significant downperformed CNN model. The cause for this, is the Unet's performance and the nature of CNN's training set. Although the precise masking CNN achieved closed scores in classification among all models, it was trained with the ground truth masks. Unet, while its results are very close with the relative extended masking model in its category, can not produce precise enough masks for the CNN. The precise masking pipeline reached an *F1* score of 0 % and 67 % for the abnormal and normal class respectively, with 50 % *AUC-ROC* score and 51% accuracy on test set. Its normalized matrix is presented in Figure 7.20 (b). Table 7.5 shows the results of the Unet-CNN pipeline classification models.



(a) (b)

***Figure 7.20**, Unet-CNN pipeline's normalized matrices on test set : (a) extended masking 240 x 320), (b) precise masking (160 x 120) (0:abnormal , 1:normal)*

***Table 7.5**, Unet - CNN Pipeline models*

Unet - CNN Pipeline		
	Extended Masking (240 x 320)	Precise Masking (240 x 320)
Test accuracy	73.0 %	53.0 %
F1 score abnormal normal	72.0 % 74.0 %	0 % 67.0 %
AUC-ROC	73.0 %	50.0 %

7.4.4 SGAN

The results of training the SGAN are not promising for the current task. The Discriminator failed to be trained in the task of the classification of the cardiac LA as normal or abnormal, while Generator is not capable to produce the A4C images of a TTE. The main reason for this fail is the small unlabeld dataset that was used. This small sized dataset has not supply the Discriminator with enough diverged images, in order to generalize on the data, and the Discriminator was overfiited on these data. The overfiiting of the Discriminator led into collapsed gradiends, which resulted in the Generator's failure to learn the inner patterns of the data for producing the desired realistic images.

7.5 Conclusion

Unet with data segmentation and the extended masks performed well even with a small dataset, with the image resolution of 120 x 160 to outperform the 240 x320. It succeeded to discover the LA in the A4C images of TTE. On the other hand, CNN achieved a relative low performance in classifying the LA size as abnormal or normal for both kind of datasets of extended masked images and masks as input. This is due to the small size of the dataset, which was 1151 images (528 for traing). Nevertheless, CNN performance was improving as the dataset become bigger, which shows that the model could discover the right patterns in the images. Our experiments showed clearly that the pipeline Unet-CNN with the extended masks images manages to give way better results than the precise masks. The reason was the need for a very precise prediction of the precise masks from the Unet, which could not be achieved efficiently with the current size of the dataset. SGAN, due to the small sized dataset, did not succeeded in classification task and in producing real A4C images. Mainly the unlabeled real A4C images were very few and unsupervised Discriminator could not discover the inner patterns of the images, which would help its supervised part.

Experiments with larger datasets, with images from different machine vendors, are expecting to give improved and more efficient results, especially for the classification tasks.

References

[1] Madani, A. et al, Fast and accurate view classification of echocardiograms using deep learning, npj Digital Medicine 1:6, 2018

[2] Madani, A. et al, Deep echocardiography: data-efficient supervised and semi-supervised deep learning towards automated diagnosis of cardiac disease, npj Digital Medicine 1:59, (2018)

[3] Jo, T., Machine Learning Foundations Supervised, Unsupervised, and Advanced Learning, Springer, 2021

[4] Murphy, P. K., Machine Learning: A Probabilistic Perspective, The MIT Press, (2012)

[5] Zhu, X., Goldberg, A.B., Introduction to Semi-supervised Learning (Synthesis Lectures on Artificial Intelligence and Machine Learning), Morgan & Claypool, (2009)

[6] Bishop, C.M., Pattern Recognition and Machine Learning, Springer, 2011

[7] LeCun, Y., Bengio, Y., Hinton, G., Deep learning, Nature, (2015)

[8] Goodfellow, I., Bengio, Y., A.Courville, Deep Learning, www.deeplearningbook.org, (2016)

[9] Nielsen, M., Neural Networks and Deep Learning [Internet]. Neuralnetworksanddeeplearning.com. 2020 [cited 16 February 2020]. Available from: http://neuralnetworksanddeeplearning.com/

[10] Wu, J. , Introduction to Convolutional Neural Networks, Nanjing University, China, (2017)

[11] Bishop, C. M., Pattern Recognition and Machine Learning, Springer, 2011

[12] Ioffe, S., Szegedy, Cr, .Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, Google , (2015)

[13] Srivastava, N. et al, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, University of Toronto, (2014)

[14] Hinton , G.E. et al, Improving neural networks by preventing co-adaptation of feature detectors, University of Toronto, (2012)

[15] Thoma, M., A Survey of Semantic Segmentation, arXiv:1602.06541v, (2016)

[16] Csurka, G., Larlus, D., Perronnin, F., What is a good evaluation measure for semantic segmentation, Xerox Research Centre Europe, (2013)

[17] Duque-Arias, D.et al, On Power Jaccard Losses for Semantic Segmentation, SCITEPRESS, (2021)

[18] Ronneberger, O., Fischer, Ph., Thomas, B.T., U-Net: Convolutional Networks for Biomedical Image Segmentation, University of Freiburg, (2015)

[19] Goodfellow I. J. et al, Generative Adversarial Nets, University of Montreal, (2014)

[20] Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V.,Radford, A., and Chen, X. Improved Techniques for Training GANs. ArXiv e-prints(2016)

[21] Libby, P. et al, Braunwald's Heart Disease: A Textbook of Cardiovascular Medicine12th edition, (2021)

[22] Otto MD, C. M. , Textbook of Clinical Echocardiography 6th edition, Elsevier, (2018)

[23] Galderisi, M. et al , Standardization of adult transthoracic echocardiography reporting in agreement with recent chamber quantification, diastolic function, and heart valve disease recommendations: an expert consensus document of the European Association of Cardiovascular Imaging, the European Society of Cardiology, (2017).

[24] Lang, M.R. et al, Recommendations for Cardiac Chamber Quantification by Echocardiography in Adults: An Update from the American Society of Echocardiography and the European Association of Cardiovascular Imaging, European Heart Journal – Cardiovascular Imaging, (2015)

[25] Du Bois, D., Du Bois EF , A formula to estimate the approximate surface area if height and weight be known, Archives of Internal Medicine, (1916)

[26] Mosteller, RD., Simplified calculation of body-surface area, N Engl J Med, (1987)

[27] Kou, S. et al, Echocardiographic reference ranges for normal cardiac chamber size: results from the NORRE study, European Heart Journal – Cardiovascular Imaging, (2014)

[28] Dutta, Abhishek, Zisserman, Andrew, The VGG Image Annotator (VIA), arXiv:1904.10699, (2019)

Appendix

1.1 Import Libraries Code

```
import cv2
import numpy as np
import pandas as pd
import math
import scipy
import matplotlib.pyplot as plt

import os
import pathlib
import shutil
import tempfile
from math import ceil
import PIL
import PIL.Image as Image
import random

import sklearn as skl
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix,
f1_score,roc_curve, auc, roc_auc_score, ConfusionMatrixDisplay
from sklearn.preprocessing import LabelBinarizer

import tensorflow as tf
import tensorflow.keras
from tensorflow.keras import models, metrics,layers, regularizers, utils,
preprocessing
```

```
from keras.models import Model, Sequential
from keras.preprocessing.image import ImageDataGenerator
from keras import backend as Kb
from keras.backend import int_shape

%matplotlib inline

random.seed  = 42

# Use of GPU
physical_devices = tf.config.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

1.2 Dataset Creation Code

```
#Retrieve Data - Calculate BSA
def upload_data(bsa_file, columns=[]):
   return pd.read_csv(bsa_file , sep=';', header=0, usecols= columns)
[columns]

def create_bsa_data_dict(bsa_dataframe):
   bsa_dataframe.set_index('image')
   bsa_data_dict = bsa_dataframe.to_dict('index')
   bsa_data_diction = {}
   for i in bsa_data_dict:
      bsa_data_diction[str(bsa_data_dict[i]['image'])]= {'height':
                                 bsa_data_dict[i]['height'],
                                    'weight': bsa_data_dict[i]['weight'],
                                    'long_axis': bsa_data_dict[i]['long_axis'],
```

```
                                       'area': bsa_data_dict[i]['area'],
                                       'volume': bsa_data_dict[i]['volume'],
                                       'birth': bsa_data_dict[i]['birth'],
                                       'gender': bsa_data_dict[i]['gender'],
                                       'bsa_coef': math.sqrt(
                                      (bsa_data_dict[i]['height']) *
                                             bsa_data_dict[i]['weight'] )/60}
  return bsa_data_diction

# bsa (body-surface-area) coeffiecent per (Mosteller's formula)
def bsa_calculation(bsa_data_diction):
  bsa_coef_dict = {}
  for image in bsa_data_diction.keys():
    bsa_coef_dict[image] = math.sqrt( (bsa_data_diction[image]['height']) *
                                              bsa_data_diction[image]['weight'] )/60
  return bsa_coef_dict

#Create images, masked_images and  labels for UNET-CNN
def sort_list_asc(image_list):
  sorted_list = []
  for image_name in list:
    image_name= os.path.splitext(image_name)[0]
    sorted_list.append(int(image_name))
  return sorted(sorted_list)
```

For UNET

```
def create_image_mask_label_sets ( image_directory, bsa_data_dict,
                                              img_width, img_height ):
  image_mask = []
```

```
image_mask_dataset= []
image_dataset = []
mask_dataset = []
label_dataset = {}
volume_over_bsa = {}

images = os.listdir(image_directory)
images = sort_list_asc(images)
for image_name in images:

    mask = cv2.imread(mask_directory + '/' +  str(image_name) + '.jpg',
                                    cv2.IMREAD_UNCHANGED)
    mask = cv2.resize(mask, (img_width, img_height), interpolation =
                                   cv2.INTER_NEAREST)
    mask_dataset.append(np.expand_dims(mask,2))

    image = cv2.imread(image_directory + '/' + str(image_name) + '.jpg',
                                    cv2.IMREAD_UNCHANGED)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    image = cv2.resize(image, (img_width, img_height), interpolation =
                                   cv2.INTER_NEAREST)
    image_dataset.append(np.expand_dims(image,2))

    image_mask_dataset.append(np.expand_dims([image,mask],3))
    volume_over_bsa= round   ( float ( bsa_data_dict [
                                             str(image_name)]['volume']) /
                                             bsa_data_dict[str(image_name)]['bsa_coef'],3)
   bsa_data_dict[str(image_name)]['volume_over_bsa']=
                                                                   volume_over_bsa
```

```
        if volume_over_bsa >= 34:
            label_dataset[str(image_name)]= 'abnormal'
        else:
            label_dataset[str(image_name)] = 'normal'

    image_mask_dataset= np.asarray(image_mask_dataset, dtype=np.float32)
    image_dataset = np.asarray(image_dataset, dtype=np.float32)
    mask_dataset =  np.asarray(mask_dataset, dtype=np.float32)
    return image_dataset, mask_dataset, image_mask_dataset, label_dataset
```

For CNN

```
def create_masked_image(images, masks):
    masked_image_dataset = []
    for i in range(len(images)):
        image = cv2.bitwise_and(images[i], images[i], mask= masks[i])
        masked_image_dataset.append(np.expand_dims(image,2))
    return np.asarray(masked_image_dataset)

def create_image_mask_label_sets ( image_directory, bsa_data_dict,
                                   img_width, img_height ):
    image_dataset = []
    mask_dataset = []
    label_dataset = {}
    volume_over_bsa = {}

    images = os.listdir(image_directory)
    images = sort_list_asc(images)
    for image_name in images:
```

```
    mask = cv2.imread(mask_directory + '/' +  str(image_name) + '.jpg',
                                cv2.IMREAD_UNCHANGED)
    mask = cv2.resize(mask, (img_width, img_height), interpolation =
                               cv2.INTER_NEAREST)
    mask_dataset.append(np.expand_dims(mask,2))

    image = cv2.imread(image_directory + '/' + str(image_name) + '.jpg',
                                cv2.IMREAD_UNCHANGED)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    image = cv2.resize(image, (img_width, img_height), interpolation =
                               cv2.INTER_NEAREST)
    image_dataset.append(np.expand_dims(image,2))

    volume_over_bsa= round   ( float ( bsa_data_dict [
                                      str(image_name)]['volume']) /
                                      bsa_data_dict[str(image_name)]['bsa_coef'],3)
   bsa_data_dict[str(image_name)]['volume_over_bsa']=
                                                        volume_over_bsa

    if volume_over_bsa >= 34:
       label_dataset[str(image_name)]= 'abnormal'
    else:
       label_dataset[str(image_name)] = 'normal'

image_masked_dataset=  create_masked_image(image_dataset,
                                  mask_dataset)
image_dataset = np.asarray(image_dataset, dtype=np.float32)
mask_dataset =  np.asarray(mask_dataset, dtype=np.float32)
```

```
    return image_dataset, image_masked_dataset, mask_dataset,
               label_dataset, bsa_data_dict
```

For GAN

```
def create_image_mask_label_sets ( image_directory, bsa_data_dict,
                                            img_width, img_height ):
    image_mask = []
    image_dataset = []
    label_dataset = {}
    volume_over_bsa = {}

    images = os.listdir(image_directory)
    images = sort_list_asc(images)
    for image_name in images:
        image = cv2.imread(image_directory + '/' + str(image_name) + '.jpg',
                                cv2.IMREAD_UNCHANGED)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        image = cv2.resize(image, (img_width, img_height), interpolation =
                                cv2.INTER_NEAREST)
        image_dataset.append(np.expand_dims(image,2))



        volume_over_bsa= round   ( float ( bsa_data_dict [
                                    str(image_name)]['volume']) /
                                    bsa_data_dict[str(image_name)]['bsa_coef'],3)
       bsa_data_dict[str(image_name)]['volume_over_bsa']=
                                                        volume_over_bsa

        if volume_over_bsa >= 34:
```

```
        label_dataset[str(image_name)]= 'abnormal'
      else:
        label_dataset[str(image_name)] = 'normal'

  image_dataset = np.asarray(image_dataset, dtype=np.float32)
  return image_dataset, label_dataset, bsa_data_dict

#Train, Validation,Test Set
def labels_one_hot_encode(labels):
  le = LabelBinarizer()
  labels = le.fit_transform(labels)
  labels = utils.to_categorical(labels, 2)
  return labels, le.classes_

#Training, Validation, Test sets for Unet-CNN Models
#rescaling images, masks to 0,1
def rescaling_images(x_train, mask_train, x_val, mask_val, x_test,
mask_test):
  x_train = x_train * 1./255
  mask_train = mask_train * 1./255
  x_val = x_val * 1./255
  mask_val =  mask_val * 1./255
  x_test = x_test * 1./255
  mask_test = mask_test * 1./255
  return x_train, mask_train, x_val, mask_val, x_test, mask_test

def CNN_sets_split(samples, labels, seed, split_1=0.5, split_2=0.5):
  X_train, X_val, y_train, y_val = train_test_split(samples, labels, test_size =
                          split_1, random_state=seed, shuffle=True, stratify=labels)
  X_val, X_test, y_val, y_test = train_test_split(X_val, y_val, test_size =
```

```
                    split_2, random_state=seed, shuffle=True, stratify=y_val)
    return X_train, y_train, X_val, y_val, X_test, y_test
```

For UNET

```
def separate_images_masks(data_array):
    images=[]
    masks=[]
    for image in data_array:
        images.append(image[0])
        masks.append(image[1])

    images = np.asarray(images, dtype=np.float32)
    masks = np.asarray(masks, dtype=np.float32)
    return images, masks

def datasets_UNET(image_directory, mask_directory, bsa_directory,
img_width, img_height, seed=42):

    bsa_data = upload_data(bsa_directory,
              ['image','height','weight','long_axis','area','volume','birth','gender'])
    bsa_data_dict = create_bsa_data_dict(bsa_data)
    image_dataset, mask_dataset, image_mask_dataset, label_dataset =
                          create_image_mask_label_sets(image_directory,
                     mask_directory, bsa_data_dict, img_width, img_height)

    labels = np.asarray(list(label_dataset.values()))

    lables_one_hot, classes = labels_one_hot_encode(labels)

    X_train, _, X_val, _, X_test, _ = CNN_sets_split(image_mask_dataset,
```

```
                                                    lables_one_hot, seed)

    X_train_UNET, mask_train_UNET = separate_images_masks(X_train)
    X_val_UNET, mask_val_UNET = separate_images_masks(X_val)
    X_test_UNET, mask_test_UNET = separate_images_masks(X_test)

    _, mask_train_UNET, _, mask_val_UNET, _, mask_test_UNET =
      rescaling_images(X_train_UNET, mask_train_UNET,
                                      X_val_UNET, mask_val_UNET,
                                      X_test_UNET, mask_test_UNET)
    return image_dataset, mask_dataset, X_train_UNET, mask_train_UNET,
           X_val_UNET, mask_val_UNET, X_test_UNET, mask_test_UNET
```

For CNN

```
def datasets_CNN_extended(image_directory, mask_directory, bsa_data_dict,
img_width, img_height, seed=42):

    image_dataset, image_masked_dataset, mask_dataset, label_dataset,
              bsa_data_dict = create_image_mask_label_sets(image_directory,
                     mask_directory, bsa_data_dict, img_width, img_height)

    labels = np.asarray(list(label_dataset.values()))

    lables_one_hot, classes, label_binarizer = labels_one_hot_encode(labels)
    X_train, y_train, X_val, y_val, X_test, y_test=
              CNN_sets_split(image_masked_dataset, lables_one_hot, seed)
    return image_dataset, image_masked_dataset, mask_dataset,
           label_dataset, bsa_data_dict, classes, label_binarizer, X_train,
           y_train, X_val, y_val, X_test, y_test
```

```
def datasets_CNN_original(image_directory, mask_directory, bsa_data_dict,
img_width, img_height, seed=42):

    image_dataset, image_masked_dataset, mask_dataset, label_dataset,
                    bsa_data_dict = create_image_mask_label_sets(image_directory,
                                  mask_directory, bsa_data_dict, img_width, img_height)

    labels = np.asarray(list(label_dataset.values()))

    lables_one_hot, classes, label_binarizer = labels_one_hot_encode(labels)

    X_train, y_train, X_val, y_val, X_test, y_test =
                                  CNN_sets_split(mask_dataset, lables_one_hot, seed)
    return image_dataset, image_masked_dataset, mask_dataset,
              label_dataset, bsa_data_dict, classes, label_binarizer, X_train,
              y_train, X_val, y_val, X_test, y_test

#Training, supervised, Unsupervised, Test sets for GAN Model
#rescaling images, masks to -1,1
def rescaling_images(x_train, mask_train, x_val, mask_val, x_test,
mask_test):
    x_train = (x_train – 127.5) * 1/127.5
    x_sup = ( x_sup – 127.5) * 1/127.5
    x_unsup = ( x_unsup – 127.5) * 1/127.5
    x_test = x_test – 127.5) * 1/127.5
    return x_train, mask_train, x_val, mask_val, x_test, mask_test

def GAN_sets_split(samples, labels, seed, split_1=0.1, split_2=0.7):
    X_train, X_test, y_train, y_test = train_test_split(samples, labels, test_size =
                                split_1, random_state=seed, shuffle=True, stratify=labels)
```

```
    X_sup, X_unsup, y_sup, y_unsup = train_test_split(X_train, y_train,
            test_size = split_2, random_state=seed, shuffle=True,
                         stratify=y_train)
    return X_train, y_train, X_sup, y_sup, X_unsup, y_unsup, X_test, y_test

def datasets_GAN(image_directory, bsa_data_dict, img_width, img_height,
seed=42):

    image_dataset, label_dataset, bsa_data_dict=
                            create_image_mask_label_sets(image_directory,
                                  bsa_data_dict, img_width, img_height)

    labels = np.asarray(list(label_dataset.values()))

    lables_one_hot, classes, label_binarizer = labels_one_hot_encode(labels)

    X_train, y_train, X_sup, y_sup, X_unsup, y_unsup, X_test, y_test =
                         GAN_sets_split(image_dataset, lables_one_hot, seed)

    X_train, X_sup, X_unsup , X_test = rescaling_images(X_train, X_sup,
                                                       X_unsup        ,
X_test)

    return image_dataset, X_train, y_train, X_sup, y_sup, X_unsup, y_unsup,
            X_test, y_test, label_dataset, bsa_data_dict, classes, label_binarizer

def generate_real_images(images, labels, batch):
    ix = randint(0, images.shape[0], batch)
    X, labels = images[ix], labels[ix]
    y = ones((batch, 1))
```

```
    y, _, _= labels_one_hot_encode(y)
    return [X, labels], y

def  generate_noise_points(gen_input, batch):
    z_input = randn(gen_input * batch)
    z_input = z_input.reshape(batch, 1, 1, gen_input)
    return z_input

def generate_fake_images(generator, gen_input, batch):
    z_input = generate_noise_points(gen_input, batch)
    fake_images = generator.predict(z_input)
    y = zeros((batch, 1))
    y, _, _= labels_one_hot_encode(y)
    return fake_images, y
```

FOR GAN

```
def GAN_sets_split(samples, labels, seed, split_1=0.1, split_2=0.8):
    X_train, X_test, y_train, y_test = train_test_split(samples, labels, test_size =
                                split_1, random_state=seed, shuffle=True, stratify=labels)
    X_sup, X_unsup, y_sup, y_unsup = train_test_split(X_train, y_train,
            test_size = split_2, random_state=seed, shuffle=True, stratify=y_train)
    return X_train, y_train, X_sup, y_sup, X_unsup, y_unsup, X_test, y_test

def  datasets_GAN(image_directory,  bsa_data_dict,  img_width,  img_height,
seed=42):
    image_dataset, label_dataset, bsa_data_dict =
                    create_image_mask_label_sets(image_directory,
                                bsa_data_dict, img_width, img_height)
    labels = np.asarray(list(label_dataset.values()))
```

```
    lables_one_hot, classes, label_binarizer = labels_one_hot_encode(labels)

    X_train, y_train, X_sup, y_sup, X_unsup, y_unsup, X_test, y_test =
                            GAN_sets_split(image_dataset, lables_one_hot, seed)

    X_train, X_sup, X_unsup , X_test = rescaling_images(X_train, X_sup,
                                                        X_unsup , X_test)

    return image_dataset, X_train, y_train, X_sup, y_sup, X_unsup, y_unsup,
            X_test, y_test, label_dataset, bsa_data_dict, classes, label_binarizer

def generate_real_images(images, labels, batch):
    ix = randint(0, images.shape[0], batch)
    X, labels = images[ix], labels[ix]
    y = ones((batch, 1))
    y, _, _= labels_one_hot_encode(y)
    return [X, labels], y

def  generate_noise_points(gen_input, batch):
    z_input = randn(gen_input * batch)
    z_input = z_input.reshape(batch, 1, 1, gen_input)
    return z_input

def generate_fake_images(generator, gen_input, batch):
    z_input = generate_noise_points(gen_input, batch)
    fake_images = generator.predict(z_input)
    y = zeros((batch, 1))
    y, _, _= labels_one_hot_encode(y)
    return fake_images, y
```

1.3 Unet Code

1.3.1 Code for Training the Unet Models

#Dataset Creation Functions Load

```
%run Datasets_Creation.ipynb
```

#Plot – Print Datasets Functions

```
def plot_images(image_set, mask_set, lines=6):
    image_indexs = []

    plt.figure(figsize=(15, 15))
    for i in range(lines):
        plt.figure(figsize=(15, 15))
        image = random.randint(0, len(image_set))
        ax = plt.subplot(121)
        plt.imshow(image_set[image,:,:,0])
        ax = plt.subplot(122)
        plt.imshow(mask_set[image,:,:,0])
        plt.axis("off")

def print_size_shapes(images, masks, images_name, masks_mame):
    # Dataset
    print(f'{images_name} Dataset size : {len(images)}')
    print(f'{images_name} shape: {images.shape}')
    print(f'Single {images_name} shape: {images[1].shape}')
    print(f'{masks_mame} shape: {masks.shape}')
    print(f'Single {masks_mame} shape: {masks[1].shape}')

def print_sets_size_shapes(dataset, images_name, masks_name):
    images, masks = tuple(zip(*dataset))
```

```
    images = np.asarray(images)
    masks = np.asarray(masks)

    # Dataset
    print(f'{images_name} Dataset size : {len(images) * len(images[1])}')
    print(f'{images_name} shape: {images.shape[0]}')
    print(f'Batched {images_name} shape: {images[0].shape}')
    print(f'Single {images_name} shape: {images[0][0].shape}')

    print(f'{masks_name} Dataset size : {len(masks) * len(masks[1])}')
    print(f'{masks_name} shape: {masks.shape[0]}')
    print(f'Batched {masks_name} shape: {masks[0].shape}')
    print(f'Single {masks_name} shape: {masks[0][0].shape}')
```

#Create dataset for the model- Without Augmentation

```
# Creating tensors data.Datasets of Train, Validation, Test sets
def to_tensors(x_train, mask_train, x_val, mask_val, x_test, mask_test):

    train_ds = tf.data.Dataset.from_tensor_slices( (x_train, mask_train) )
    val_ds = tf.data.Dataset.from_tensor_slices( (x_val, mask_val) )
    test_ds = tf.data.Dataset.from_tensor_slices( (x_test, mask_test) )
    return train_ds,val_ds,test_ds

# Configure Datasets for performance: cache, shuffle, prefetech
def configure_for_performance(ds, seed, buffer_size= 1000, batch_size=1):
    ds = ds.cache()
    ds = ds.shuffle(buffer_size= buffer_size, seed=seed)
```

```
    ds = ds.batch(batch_size)
    ds = ds.prefetch(buffer_size= tf.data.AUTOTUNE)
    return ds

def create_performance_dataset(X_train, mask_train, X_val, mask_val,
X_test, mask_test, seed, batch_size=1):

    train_ds, val_ds, test_ds = to_tensors(X_train, mask_train, X_val,
                                           mask_val, X_test, mask_test)

    train_ds = configure_for_performance(train_ds, seed, batch_size =
                                                          batch_size)
    val_ds = configure_for_performance(val_ds,seed, batch_size= batch_size)
    test_ds = configure_for_performance(test_ds,seed, batch_size= batch_size)
    return train_ds, val_ds, test_ds

# Dataset augmentation layers
data_augmentation_layers = [
        layers.RandomContrast(0.5), ]
data_augmentation = tf.keras.Sequential(
    data_augmentation_layers)

#CREATE AUGMENDED SETS FUNCTIONS
def train_validation_augmentation(x_train, mask_train, x_val, mask_val,
seed=42, batch_size=1):
    image_data_gen_args = dict(rotation_range = 5,
                    width_shift_range = 0.2,
                    height_shift_range = 0.2,
                    shear_range = 0.2,
                    zoom_range = 0.2,
```

```
                    brightness_range = (0.4, 0.8),
                    fill_mode= 'nearest')

mask_data_gen_args = dict(rotation_range = 5,
                   width_shift_range = 0.2,
                   height_shift_range = 0.2,
                   shear_range = 0.2,
                   zoom_range = 0.2,
                   brightness_range = (0.4, 0.8),
                   fill_mode= 'nearest',
                   preprocessing_function = lambda x: np.where(x>0, 1, 0).
                                        astype(x.dtype))

# Train images and masks generators
# Train images
image_data_generator = ImageDataGenerator(**image_data_gen_args)
image_data_generator.fit(x_train, augment = True, seed = seed)

train_image_generator = image_data_generator.flow(x_train, seed = seed,
                                 shuffle=True,
                                 batch_size=batch_size)

# Train masks
mask_data_generator = ImageDataGenerator(**mask_data_gen_args)
mask_data_generator.fit(mask_train, augment = True, seed = seed)

train_mask_generator = mask_data_generator.flow(mask_train, seed =
                                  seed,shuffle=True,  batch_size=batch_size)
```

```
  # Validation images and masks generators
  # Validation images
  valid_image_gen = ImageDataGenerator(**image_data_gen_args)
  valid_image_gen.fit(x_val, augment = True, seed = seed )
  valid_img_generator = valid_image_gen.flow(x_val, seed = seed,
                                             shuffle=True,
                                             batch_size=batch_size)
  # Validation masks
  valid_mask_gen = ImageDataGenerator(**mask_data_gen_args)
  valid_mask_gen.fit(mask_val, augment = True, seed = seed)
  valid_mask_generator = valid_mask_gen.flow(mask_val, seed = seed,
                                             shuffle=True,
                                          batch_size=batch_size)
  return  train_image_generator, train_mask_generator,
              valid_img_generator,  valid_mask_generator

def image_mask_generator_set(image_generator, mask_generator):
  return zip(image_generator, mask_generator)

def plot_Gerarator_Images(img_gen, mask_gen):
  for i in range(0,8):
    x = img_gen.next()
    y = mask_gen.next()
    image = x[i]
    mask = y[i]
    plt.subplot(1,2,1)
    plt.imshow(image[:,:,0])
    plt.subplot(1,2,2)
    plt.imshow(mask[:,:,0])
    plt.show()
```

```
#Model's functions
#Model construction
def build_U_net_model(input_shape, start_neurons= 64):
    input_layer = tf.keras.Input(shape=input_shape)
    # Image augmentation block
    input_layer = data_augmentation(input_layer)
    input_resc = layers.Rescaling(scale=1./255)(input_layer)

    conv1 = layers.Conv2D(start_neurons * 1, (3, 3), activation="relu",
                padding="same", kernel_initializer='he_normal')(input_resc)
    conv1 = layers.Conv2D(start_neurons * 1, (3, 3), activation="relu",
                padding="same", kernel_initializer='he_normal')(conv1)
    pool1 = layers.MaxPooling2D(pool_size=(2, 2), strides=2)(conv1)

    conv2 = layers.Conv2D(start_neurons * 2, (3, 3), activation="relu",
                padding="same",kernel_initializer='he_normal')(pool1)
    conv2 = layers.Conv2D(start_neurons * 2, (3, 3), activation="relu",
                padding="same",kernel_initializer='he_normal')(conv2)
    pool2 = layers.MaxPooling2D(pool_size=(2, 2), strides=2)(conv2)

    conv3 = layers.Conv2D(start_neurons * 4, (3, 3), activation="relu",
                padding="same",kernel_initializer='he_normal')(pool2)
    conv3 = layers.Conv2D(start_neurons * 4, (3, 3), activation="relu",
                padding="same",kernel_initializer='he_normal')(conv3)
    pool3 = layers.MaxPooling2D(pool_size=(2, 2), strides=2)(conv3)
    drop = layers.Dropout(0.5)(pool3)
```

```
# Middle
convm = layers.Conv2D(start_neurons * 8, (3, 3), activation="relu",
            padding="same",kernel_initializer='he_normal')(drop)
convm = layers.Conv2D(start_neurons * 8, (3, 3), activation="relu",
            padding="same",kernel_initializer='he_normal')(convm)

deconv3 = layers.Conv2DTranspose(start_neurons * 4, (2, 2), strides=2,
            padding="same")(convm)
uconv3 = layers.concatenate([deconv3, conv3])

uconv3 = layers.Conv2D(start_neurons * 4, (3, 3), activation="relu",
            padding="same",kernel_initializer='he_normal')(uconv3)
uconv3 = layers.Conv2D(start_neurons * 4, (3, 3), activation="relu",
            padding="same",kernel_initializer='he_normal')(uconv3)

deconv2 = layers.Conv2DTranspose(start_neurons * 2, (2, 2), strides=2,
            padding="same")(uconv3)
uconv2 = layers.concatenate([deconv2, conv2])

uconv2 = layers.Conv2D(start_neurons * 2, (3, 3), activation="relu",
            padding="same",kernel_initializer='he_normal')(uconv2)
uconv2 = layers.Conv2D(start_neurons * 2, (3, 3), activation="relu",
            padding="same",kernel_initializer='he_normal')(uconv2)

deconv1 = layers.Conv2DTranspose(start_neurons , (2, 2), strides=2,
            padding="same")(uconv2)
uconv1 = layers.concatenate([deconv1, conv1])

uconv1 = layers.Conv2D(start_neurons * 1, (3, 3), activation="relu",
            padding="same", kernel_initializer='he_normal')(uconv1)
```

```
    uconv1 = layers.Conv2D(start_neurons * 1, (3, 3), activation="relu",
                padding="same", kernel_initializer='he_normal')(uconv1)
    output_layer = layers.Conv2D(1, (1,1), activation="sigmoid")(uconv1)

    return tf.keras.Model(input_layer,output_layer)

# Learning Rate optimezer setup
def learning_scheduler(initial_rate):
    return  tf.keras.optimizers.schedules.InverseTimeDecay(
            initial_rate,
            decay_steps=STEPS_PER_EPOCH*25,
            decay_rate=1.0,
            staircase=False)

def get_optimizer(initial_rate):
    return tf.keras.optimizers.Adam(learning_scheduler(initial_rate))

# plot learning rate degradation
def plot_learning_schedule(initial_rate, batch_size):
    step = np.linspace(0,25000)
    lr_schedule = learning_scheduler(initial_rate)
    lr = lr_schedule(step)
    plt.figure(figsize = (8,6))
    plt.plot(step/STEPS_PER_EPOCH, lr)
    plt.title(f'Initial Learnig rate : {initial_rate} with Batch size: {batch_size}
                Decay Ratio', fontsize=15)
    plt.ylim([0,max(plt.ylim())])
    plt.xlabel('Epoch')
    _ = plt.ylabel('Learning Rate')
```

```
# callbacks logs for each model
def get_callbacks(name, patience, monitor):
   return [
      tf.keras.callbacks.EarlyStopping(
         monitor= monitor,
         verbose=2,
         patience= patience,
         restore_best_weights= True),
      tf.keras.callbacks.TensorBoard(logdir/name),
   ]
def checkpointer(model_name):
   return tf.keras.callbacks.ModelCheckpoint(model_name, verbose=1,
            save_best_only=True)
#Model Setup
def jaccard_index(masks_true, masks_predicted):
   masks_true_flatten = Kb.flatten(masks_true)
   masks_predicted_flatten = Kb.flatten(masks_predicted)
   intersection = Kb.sum(masks_true_flatten * masks_predicted_flatten)
  # Addittion of 10 for avoiding divide with zero
   return (intersection + 10.0) / (Kb.sum(masks_true_flatten) +
            Kb.sum(masks_predicted_flatten) - intersection + 10.0)


def jaccard_loss(y_true, y_pred, p_value=1.75, smooth = 10):
   y_true_f = Kb.flatten(y_true)
   y_pred_f = Kb.flatten(y_pred)


   intersection = Kb.sum(y_true_f * y_pred_f)
   term_true = Kb.sum(Kb.pow(y_true_f, p_value))
   term_pred = Kb.sum(Kb.pow(y_pred_f, p_value))
   union = term_true + term_pred - intersection
```

```
    return 1 - ((intersection + smooth) / (union + smooth))


def models_compiler(model, optimizer, loss=jaccard_loss, metrics=
            [jaccard_index]):
    return model.compile(optimizer=optimizer, loss=loss, metrics= metrics)

#Model fit Without Augmentation
def u_net_compile_fit(model, name, train_ds, val_ds, optimizer=None,
            learning_rate=0.0001, max_epochs=50, patience=5,
            monitor='val_loss', class_weights=None, batch_size=1):
    if optimizer is None:
        optimizer = get_optimizer(learning_rate)

    models_compiler(model= model, optimizer= optimizer)

    model.summary()

    history = model.fit(
        train_ds,
        steps_per_epoch = STEPS_PER_EPOCH,
        epochs = max_epochs,
        validation_data = val_ds,
        class_weight = class_weights,
        callbacks = get_callbacks(name, patience, monitor),
        verbose=1,)
    return history

#Model fit With Augmentation
```

```
def u_net_compile_fit_AUGM(model, name, train_generator, val_generator,
                train_length, validation_length, optimizer=None,
                learning_rate=0.00001, max_epochs=50, patience=5,
                monitor='val_loss', class_weights=None, batch_size=1):
    if optimizer is None:
        optimizer = get_optimizer(learning_rate)

    models_compiler(model= model, optimizer= optimizer)
    model.summary()
    history = model.fit(
        train_generator,
        steps_per_epoch = STEPS_PER_EPOCH,
        epochs = max_epochs,
        validation_data = val_generator,
        validation_steps = validation_length,
        class_weight = class_weights,
        callbacks = get_callbacks(name, patience, monitor),
        verbose=1,)
    return history

# Model's Graphs
# Plotting Net Loss cs Val_Loss and Acuuracy vs Val_Accuracy
def plot_model_graphs(model_name, type_histories):

    loss_unet = type_histories[model_name].history['loss']
    val_loss_unet = type_histories[model_name].history['val_loss']

    epochs = range(1, len(loss_unet) + 1)

    jaccard_matric_unet = type_histories[model_name].history['jaccard_index']
```

```
val_jaccard_matric_unet =
        type_histories[model_name].history['val_jaccard_index']

plt.figure(figsize=(40,40))

ax = plt.subplot(2, 2,  1)
plt.plot(epochs, loss_unet, 'b', label='Training loss')
plt.plot(epochs, val_loss_unet, 'r', label='Validation loss')
plt.title(model_name + ' Training and Validation loss', fontsize=30)
plt.xlabel('Epochs', fontsize=30)
plt.ylabel(' Loss', fontsize=30)
plt.legend(loc='upper right', fontsize=20)

ax = plt.subplot(2, 2,  2)
plt.plot(epochs, jaccard_matric_unet, 'b', label='Training Jacard Coefficient
        metric')
plt.plot(epochs, val_jaccard_matric_unet, 'r', label='Validation Jacard
        Coefficient metric')
plt.title(model_name + ' Training and Validation Jacard Coefficient metric',
        fontsize=30)
plt.xlabel('Epochs', fontsize=30)
plt.ylabel('Jacard Coefficient', fontsize=30)
plt.legend(loc='lower right', fontsize=20)
plt.show()

#RUN MODELS
#Extended Masks Paradigm ( 240 x 340)
# Constants
IMG_WIDTH= 320
IMG_HEIGHT = 240
```

```
CHANNELS = 1
START_CONVS = 64
BATCH_SIZE = 6 # 8 for augmented sets
SEED = 42

# Insert Data
image_directory = 'D:/A4C/CNN/EXTENDED/images_full_info_extented'
mask_directory = 'D:/A4C/CNN/EXTENDED/masks_extended_full_info'
bsa_directory = 'D:/A4C/CNN/EXTENDED/bsa_data_full_info_extended.csv'

# upload - create datasets
Images, Masks, X_train, mask_train, X_val, mask_val, X_test, mask_test=
        datasets_UNET(image_directory, mask_directory, bsa_directory,
        IMG_WIDTH, IMG_HEIGHT, seed=42)
# Create Dataset for the model- Without Augmentation
train_set, val_set, test_set = create_performance_dataset(X_train,
        mask_train, X_val, mask_val, X_test, mask_test, SEED,
        batch_size=BATCH_SIZE)



# Shapes of uploaded original images, masks sets (without augmentation)
print_size_shapes(Images, Masks, 'Images', 'Masks')
print_sets_size_shapes(train_set, 'Training', 'Training Masks')
print_sets_size_shapes(val_set, 'Validation', 'Validation Masks')
print_sets_size_shapes(test_set, 'Test', 'Test Masks')

# Dataset Augmented with Image Generator
image_generator, mask_generator, valid_img_generator,
        valid_mask_generator = train_validation_augmentation(X_train,
```

```
                mask_train, X_val, mask_val, seed=SEED,
                batch_size=BATCH_SIZE)

train_Generator = image_mask_generator_set(image_generator,
                mask_generator)
validation_Generator = image_mask_generator_set(valid_img_generator,
                valid_mask_generator)

# Shapes of uploaded original images, masks sets for augmented sets with
    Image Generator
print_size_shapes(Images, Masks, 'Images', 'Masks')
print_size_shapes(X_train, mask_train, 'X_train', 'Masks_train')
print_size_shapes(X_val, mask_val, 'X_val', 'Masks_val')
print_size_shapes(X_test, mask_test, 'X_test', 'Masks_test')

# Plot original image, mask pairs
plot_images(Images, Masks, lines=6)
# Plot Generator Images
plot_Gerarator_Images(image_generator, mask_generator)
# Set Learning Rate
STEPS_PER_EPOCH = len(train_set)
# for augmented sets
#STEPS_PER_EPOCH = len(X_train)//BATCH_SIZE
LEARNING_RATE = 0.0001
plot_learning_schedule(LEARNING_RATE, BATCH_SIZE)

# RUN MODEL Extended masks 240x320
# Create new model
image_size = (IMG_HEIGHT , IMG_WIDTH)
```

```
unet = build_U_net_model(input_shape= image_size + (CHANNELS,),
    start_neurons= START_CONVS)
# Run model without augmentation Datasets
type_histories['unet_extended_batch8_bigDim']= u_net_compile_fit(unet,
    'unet_extended_batch8_bigDim', train_set,
    val_set,learning_rate=LEARNING_RATE,max_epochs=300, patience=50,
    batch_size=BATCH_SIZE)
# Run model with Augmented Sets
type_histories['unet_extended_batch8_bigDim_AUGM']=
    u_net_compile_fit_AUGM(unet, 'unet_extended_batch8_bigDim_AUGM',
    train_Generator, validation_Generator,len(X_train), len(X_val),
    learning_rate=LEARNING_RATE,max_epochs=300, patience=50,
    batch_size=BATCH_SIZE)

# Save and Plot models
unet.save('saved_U_net_models/unet_extended_batch8_bigDim')
unet.save('saved_U_net_models/unet_extended_batch8_bigDim_AUGM')
plot_model_graphs("unet_extended_batch8_bigDim", type_histories)
plot_model_graphs("unet_extended_batch8_bigDim_AUGM", type_histories)
```

1.3.2 Code for Evaluating Unet the Models

```
#Dataset Creation Functions Load
%run Datasets_Creation.ipynb

# Plot Datasets Functions
# Sanity check plots
def plot_images(image_set, mask_set, lines=6):
   image_indexs = []

   plt.figure(figsize=(15, 15))
   for i in range(lines):
```

```
        plt.figure(figsize=(15, 15))
        image = random.randint(0, len(image_set))
        ax = plt.subplot(121)
        plt.imshow(image_set[image,:,:,0])
        ax = plt.subplot(122)
        plt.imshow(mask_set[image,:,:,0])
        plt.axis("off")

# Print dimensions
def print_size_shapes(images, masks, images_name, masks_mame):
    # Dataset
    print(f'{images_name} Dataset size : {len(images)}')
    print(f'{images_name} shape: {images.shape}')
    print(f'Single {images_name} shape: {images[1].shape}')
    print(f'{masks_mame} shape: {masks.shape}')
    print(f'Single {masks_mame} shape: {masks[1].shape}')

def print_sets_size_shapes(dataset, images_name, masks_name):
    images, masks = tuple(zip(*dataset))
    images = np.asarray(images)
    masks = np.asarray(masks)

    # Dataset
    print(f'{images_name} Dataset size : {len(images) * len(images[1])}')
    print(f'{images_name} shape: {images.shape[0]}')
    print(f'Batched {images_name} shape: {images[0].shape}')
    print(f'Single {images_name} shape: {images[0][0].shape}')

    print(f'{masks_name} Dataset size : {len(masks) * len(masks[1])}')
```

```
    print(f'{masks_name} shape: {masks.shape[0]}')
    print(f'Batched {masks_name} shape: {masks[0].shape}')
    print(f'Single {masks_name} shape: {masks[0][0].shape}')


# Create dataset for the model- Without Augmentation
# Creating tensors data.Datasets of Train, Validation, Test sets
def  to_tensors(x_train, mask_train, x_val, mask_val, x_test, mask_test):
    train_ds = tf.data.Dataset.from_tensor_slices( (x_train, mask_train) )
    val_ds = tf.data.Dataset.from_tensor_slices( (x_val, mask_val) )
    test_ds = tf.data.Dataset.from_tensor_slices( (x_test, mask_test) )
    return train_ds,val_ds,test_ds

# Configure Datasets for performance: cache, shuffle, prefetech
def configure_for_performance(ds, seed, buffer_size= 1000, batch_size=1):
    ds = ds.cache()
    ds = ds.shuffle(buffer_size= buffer_size, seed=seed)
    ds = ds.bataach(batch_size)
    ds = ds.prefetch(buffer_size= tf.data.AUTOTUNE)
    return ds

def create_performance_dataset(X_train, mask_train, X_val, mask_val,
     X_test, mask_test, seed, batch_size=1):

    train_ds, val_ds, test_ds  = to_tensors(X_train, mask_train, X_val,
     mask_val, X_test, mask_test)

    train_ds = configure_for_performance(train_ds,seed, batch_size=
     batch_size)
    val_ds = configure_for_performance(val_ds,seed, batch_size= batch_size)
```

```
  test_ds = configure_for_performance(test_ds,seed, batch_size= batch_size)
  return train_ds, val_ds, test_ds

# CREATE  SETS FOR AUGMENDED MODELSdef  def
def train_validation_augmentation(x_test, mask_test, seed=42, batch_size=1):

  # Test images and masks generators
  # Test images
  test_image_gen = ImageDataGenerator()
  test_image_gen.fit(x_test, augment = True, seed = seed )
  test_img_generator = test_image_gen.flow(x_test, seed = seed,
                                   shuffle=True, batch_size=batch_size)
  # Test masks
  test_mask_gen = ImageDataGenerator()
  test_mask_gen.fit(mask_test, augment = True, seed = seed)
  test_mask_generator = test_mask_gen.flow(mask_test, seed = seed,
                                   shuffle=True, batch_size=batch_size)
  return test_img_generator, test_mask_generator

def image_mask_generator_set(image_generator, mask_generator):
  return zip(image_generator, mask_generator)

# Model Setup
# Jaccard-Coefficient (Intersection Over Union) metric for semantic
  segmentation
def jaccard_index(masks_true, masks_predicted):
  masks_true_flatten = Kb.flatten(masks_true)
  masks_predicted_flatten = Kb.flatten(masks_predicted)
  intersection = Kb.sum(masks_true_flatten * masks_predicted_flatten)
```

```
    return (intersection + 10.0) / (Kb.sum(masks_true_flatten) +
                Kb.sum(masks_predicted_flatten) - intersection + 10.0)

def jaccard_loss(y_true, y_pred, p_value=1.75,smooth = 10):
    y_true_f = Kb.flatten(y_true)
    y_pred_f = Kb.flatten(y_pred)
    intersection = Kb.sum(y_true_f * y_pred_f)
    term_true = Kb.sum(Kb.pow(y_true_f, p_value))
    term_pred = Kb.sum(Kb.pow(y_pred_f, p_value))
    union = term_true + term_pred - intersection

    return 1 - ((intersection + smooth) / (union + smooth))
 # Evaluation of the model Functions
 # Take trained Unet model
def get_unet_model(model_name):
    return models.load_model(model_name,
     custom_objects={"jaccard_loss":jaccard_loss,"jaccard_index":jaccard_index
     })

# Evaluate Unet on unknown Test set
# without augmentation
def print_unet_test_evaluation(unet_model, unet_model_name, test_set,
     verbose=2):
    _, acc = unet_model.evaluate(test_set, verbose = verbose)
    print(f"Evaluation Jaccard Index of {unet_model_name} U-net model is:
     {acc*100.0} %  ")

# with augmentation
def print_unet_test_evaluation_AUGM(unet_model, unet_model_name,
     X_test,mask_test, batch_size, verbose=2):
```

```
  _, acc = unet_model.evaluate(X_test,mask_test, batch_size=batch_size,
   verbose = verbose)
  print(f"Evaluation Jaccard Index of {unet_model_name} U-net model is:
   {acc*100.0} %  ")

#Predict Jaccard coefficient (IOU) Test set with Unet
# without augmentation
def calc_predictions(unet_model,test_ds, threshold=0.5):
  masks_predictions = unet_model.predict(test_ds)
  masks_pred_thresholed = masks_predictions > threshold

  return masks_pred_thresholed
# with augmentation
def calc_predictions_AUGM(unet_model,X_test, batch_size, threshold=0.5):
  masks_predictions = unet_model.predict(X_test, batch_size=batch_size)
  masks_pred_thresholed = masks_predictions > threshold

  return masks_pred_thresholed

def test_jaccard_index(unet_model, masks_test, model_masks_predicted):
  intersection = np.logical_and(masks_test, model_masks_predicted)
  union = np.logical_or(masks_test, model_masks_predicted)
  jaccard_index_score = np.sum(intersection)/np.sum(union)
  return jaccard_index_score

def print_unet_jaccard_score(model_name, jaccard_index_score):
   print(f"Jaccard Index Score (IOU) of unet model {model_name} is:
   {jaccard_index_score}")

# Check predicted masks vs true masks of images in Test set
```

```
def create_masked_image(image, mask):
    return cv2.bitwise_and(image, image, mask= mask)

def unbatch_set(dataset):
    unbatached_dataset = []

    for batch in dataset:
        for image in batch:
            unbatached_dataset.append(image)
    return np.asarray(unbatached_dataset)

def test_image_sunnity_check(unet_model, test_ds,
     model_masks_predicted):
    test_images, test_masks = tuple(zip(*test_ds))
    test_images = np.asarray(test_images)
    test_masks = np.asarray(test_masks)
    test_images = unbatch_set(test_images)
    test_masks = unbatch_set(test_masks)

    for i in range(test_images.shape[0]):
        test_img = test_images[i]
        ground_truth = test_masks[i]
        test_img_input=np.expand_dims(test_img, 0)
        prediction_mask = model_masks_predicted[i].astype(np.uint8)
        predicted_masked_image = create_masked_image(test_img,
        prediction_mask)
        print (f'Test Image : {i+1}')
        plt.figure(figsize=(16, 12))
        plt.subplot(241)
        plt.title('Testing Image')
```

```
        plt.imshow(test_img)
        plt.subplot(242)
        plt.title('Testing Label')
        plt.imshow(ground_truth)
        plt.subplot(243)
        plt.title('Prediction on test image')
        plt.imshow(prediction_mask)
        plt.subplot(244)
        plt.title('Image with prediction mask')
        plt.imshow(predicted_masked_image)
        plt.show()
def test_image_sunnity_check_AUGM (unet_model, X_test,mask_test,
model_masks_predicted):
    for i in range(X_test.shape[0]):
        test_img = X_test[i]
        ground_truth = mask_test[i]
        test_img_input=np.expand_dims(test_img, 0)
        prediction_mask = model_masks_predicted[i].astype(np.uint8)

        predicted_masked_image = create_masked_image(test_img,
        prediction_mask)

        print (f'Test Image : {i+1}')
        plt.figure(figsize=(16, 12))
        plt.subplot(241)
        plt.title('Testing Image')
        plt.imshow(test_img)
        plt.subplot(242)
        plt.title('Testing Label')
        plt.imshow(ground_truth)
```

```
        plt.subplot(243)
        plt.title('Prediction on test image')
        plt.imshow(prediction_mask)
        plt.subplot(244)
        plt.title('Image with prediction mask')
        plt.imshow(predicted_masked_image)
        plt.show()

# confusion_matrix without augmnetation
def plot_confusion_matrix(unet_model, model_name, mask_test,
model_masks_predicted):

    mask_test = mask_test == 1
    mask_test = mask_test.flatten()

    mask_test_predictions = model_masks_predicted.flatten()

    cm = confusion_matrix(mask_test, mask_test_predictions, normalize='true')

    display_cm = ConfusionMatrixDisplay(confusion_matrix=cm,
     display_labels=['0','1'])

    display_cm.plot(cmap=plt.cm.Blues)
    plt.title(f'Confusion Matrix on {model_name} Test set \n')
    plt.show()

# confusion_matrix with augmnetation
def plot_confusion_matrix_AUGM(unet_model, model_name, mask_test,
     model_masks_predicted):
```

```
    mask_test = mask_test == 1
    mask_test = mask_test.flatten()

    mask_test_predictions = model_masks_predicted.flatten()

    cm = confusion_matrix(mask_test, mask_test_predictions, normalize='true')

    display_cm = ConfusionMatrixDisplay(confusion_matrix=cm,
    display_labels=['0','1'])

    display_cm.plot(cmap=plt.cm.Blues)
    plt.title(f'Confusion Matrix on {model_name} Test set \n')
    plt.show()

# Display Models metrics without augmnetation
def display_model_metrics(unet_model, model_name, mask_test,
     model_masks_predicted):

    mask_test = mask_test == 1
    mask_test = mask_test.flatten()

    mask_test_predictions = model_masks_predicted.flatten()


    report = classification_report(mask_test, mask_test_predictions )
    print(f'Model\'s {model_name} metrics:\n {report}')

# Display Models metrics with augmnetation
def display_model_metrics_AUGM(unet_model, model_name, mask_test,
     model_masks_predicted):
```

```
    mask_test = mask_test == 1
    mask_test = mask_test.flatten()

    mask_test_predictions = model_masks_predicted.flatten()


    report = classification_report(mask_test, mask_test_predictions )
    print(f'Model\'s {model_name} metrics:\n {report}')


# RUN EVALUATE MODELS (extended masks paradigm)
# Constants
IMG_WIDTH= 320
IMG_HEIGHT = 240
CHANNELS = 1
START_CONVS = 64
BATCH_SIZE = 6 # 8 for augmented
SEED = 42

# Insert Data
image_directory = 'D:/A4C/CNN/EXTENDED/images_full_info_extented'
mask_directory = 'D:/A4C/CNN/EXTENDED/masks_extended_full_info'
bsa_directory = 'D:/A4C/CNN/EXTENDED/bsa_data_full_info_extended.csv'

Images, Masks, X_train, mask_train, X_val, mask_val, X_test, mask_test=
    datasets_UNET(image_directory, mask_directory, bsa_directory,
    IMG_WIDTH, IMG_HEIGHT, seed=42)

# Dataset for the model- without Augmentation
```

```
train_set, val_set, test_set = create_performance_dataset(X_train,
    mask_train, X_val, mask_val, X_test, mask_test, SEED,
    batch_size=BATCH_SIZE)

# Shapes of uploaded original images, masks sets
print_size_shapes(Images, Masks, 'Images', 'Masks')
print_sets_size_shapes(train_set, 'Training', 'Training Masks')
print_sets_size_shapes(val_set, 'Validation', 'Validation Masks')
print_sets_size_shapes(test_set, 'Test', 'Test Masks')



# Dataset Augmented with Image Generator
print_size_shapes(Images, Masks, 'Images', 'Masks')
print_size_shapes(X_train, mask_train, 'X_train', 'Masks_train')
print_size_shapes(X_val, mask_val, 'X_val', 'Masks_val')
print_size_shapes(X_test, mask_test, 'X_test', 'Masks_test')

# Plot original image, mask pairs
plot_images(X_test, mask_test, lines=6)

# EVALUATE MODEL Extended masks 240x320 without
    Augmentation
unet_extended_batch8_bigDim =
    get_unet_model('saved_U_net_models/unet_extended_batch8_bigDim')

print_unet_test_evaluation(unet_extended_batch8_bigDim,
    'unet_extended_batch8_bigDim', test_set)
```

```
unet_extended_batch8_bigDim_masks_predicted =
    calc_predictions(unet_extended_batch8_bigDim, test_set) # threshold to
    0.5

unet_extended_batch8_bigDim_prediction_score =
    test_jaccard_index(unet_extended_batch8_bigDim, mask_test,
    unet_extended_batch8_bigDim_masks_predicted)

print_unet_jaccard_score('unet_extended_batch8_bigDim',
    unet_extended_batch8_bigDim_prediction_score)

test_image_sunnity_check(unet_extended_batch8_bigDim, test_set,
    unet_extended_batch8_bigDim_masks_predicted)

plot_confusion_matrix(unet_extended_batch8_bigDim,'unet_extended_batch
    8_bigDim', mask_test, unet_extended_batch8_bigDim_masks_predicted)

display_model_metrics(unet_extended_batch8_bigDim,'unet_extended_batch
    8_bigDim', mask_test, unet_extended_batch8_bigDim_masks_predicted)

# EVALUATE AUGMENTED MODEL Extended masks 240x320 with
    Augmentation
unet_extended_batch8_bigDim_AUGM =
    get_unet_model('saved_U_net_models/unet_extended_batch8_bigDim_A
    UGM')

print_unet_test_evaluation_AUGM(unet_extended_batch8_bigDim_AUGM,
    'unet_extended_batch8_bigDim_AUGM', X_test, mask_test, BATCH_SIZE)

# Predictions for Image Generator Augmented Model
```

```
unet_extended_batch8_bigDim_masks_predicted_AUGM =
    calc_predictions_AUGM(unet_extended_batch8_bigDim_AUGM,X_test,
    BATCH_SIZE)


unet_extended_batch8_bigDim_prediction_score_AUGM =
    test_jaccard_index(unet_extended_batch8_bigDim_AUGM, mask_test,
    unet_extended_batch8_bigDim_masks_predicted_AUGM)


print_unet_jaccard_score('unet_extended_batch8_bigDim_AUGM',
    unet_extended_batch8_bigDim_prediction_score_AUGM)


test_image_sunnity_check_AUGM(unet_extended_batch8_bigDim_AUGM,
    X_test, mask_test,
    unet_extended_batch8_bigDim_masks_predicted_AUGM)
plot_confusion_matrix_AUGM(unet_extended_batch8_bigDim_AUGM,'unet
    _extended_batch8_bigDim_AUGM', mask_test,
    unet_extended_batch8_bigDim_masks_predicted_AUGM)


display_model_metrics_AUGM(unet_extended_batch8_bigDim_AUGM,'unet
    _extended_batch8_bigDim_AUGM', mask_test,
    unet_extended_batch8_bigDim_masks_predicted_AUGM)
```

1.4 CNN Code

1.4.1 Code For Training CNN Models

```
# Dataset Creation Functions Import
%run Datasets_Creation.ipynb


# Plot Functions
def plot_images(images,masks, masked_images, labels, lines=20):
  for i in range(100,120):
     image = random.randint(0, images.shape[0])
```

```
    print(list(labels.keys())[image])

    plt.figure(figsize=(16, 12))
    plt.subplot(131)
    plt.title(f'Original Image - {list(labels.values())[image]}')
    plt.imshow(images[image])
    plt.subplot(132)
    plt.title('Mask')
    plt.imshow(masks[image])
    plt.subplot(133)
    plt.title('Masked image')
    plt.imshow(masked_images[image])
    plt.show()

# Print dimensions
def print_size_shapes(images, masks, masked_images, labels, images_name,
masks_name, masked_images_name, labels_name ):
  # Dataset
  print(f'{images_name} Dataset size : {len(images)}')
  print(f'{images_name} shape: {images.shape}')
  print(f'Single {images_name} shape: {images[1].shape}')
  print(f'{masks_name} shape: {masks.shape}')
  print(f'Single {masks_name} shape: {masks[1].shape}')
  print(f'{masked_images_name} shape: {masked_images.shape}')
  print(f'Single {masked_images_name} shape: {masked_images[1].shape}')
  print(f'{labels_name} shape: {np.asarray(list(labels.values

#  Train, Validation,Test Set
def print_sets_size_shapes(X_train, y_train, X_val, y_val, X_test, y_test):
  # Train dataset
```

```
    print(f'Training Dataset size : {len(X_train)}')
    print(X_train.shape)
    print(X_train[1].shape)
    print(y_train.shape)

    # Validation Dataset
    print(f'Validation Dataset size : {len(X_val)}')
    print(X_val.shape)
    print(X_val[1].shape)
    print(y_val.shape)

    # Test Dataset
    print(f'Test Dataset size : {len(X_test)}')
    print(X_test.shape)
    print(X_test[1].shape)
    print(y_test.shape)

# Create dataset for the model without Data Augmentation
# Creating tensors data.Datasets of Train, Validation, Test sets
def to_tensors(x_train, y_train, x_val, y_val, x_test, y_test):

    train_ds = tf.data.Dataset.from_tensor_slices( (x_train, y_train) )
    val_ds = tf.data.Dataset.from_tensor_slices( (x_val, y_val) )
    test_ds = tf.data.Dataset.from_tensor_slices( (x_test, y_test) )
    return train_ds,val_ds,test_ds

# cache, shuffle, prefetch
# Configure Datasets for performance: cache, shuffle, prefetch
def configure_for_performance(ds, seed, buffer_size= 1000, batch_size=1):
```

```
    ds = ds.cache()
    ds = ds.shuffle(buffer_size= buffer_size, seed=seed)
    ds = ds.batch(batch_size)
    ds = ds.prefetch(buffer_size= tf.data.AUTOTUNE)
    return ds

def create_performance_dataset(X_train, y_train, X_val, y_val, X_test, y_test,
seed=42, batch_size=1):

    train_ds,  val_ds,  test_ds   =  to_tensors(X_train,  y_train,  X_val,  y_val,
X_test, y_test)

    train_ds= configure_for_performance(train_ds,seed, batch_size=
        batch_size)
    val_ds = configure_for_performance(val_ds,seed, batch_size= batch_size)
    test_ds = configure_for_performance(test_ds,seed, batch_size= batch_size)
    return train_ds, val_ds, test_ds, X_test, y_test

def create_performance_dataset(X_train, y_train, X_val, y_val, X_test, y_test,
seed=42, batch_size=1):

    train_ds,  val_ds,  test_ds   =  to_tensors(X_train,  y_train,  X_val,  y_val,
X_test,
     y_test)

    train_ds = configure_for_performance(train_ds,seed, batch_size=
     batch_size)
    val_ds = configure_for_performance(val_ds,seed, batch_size= batch_size)
    test_ds = configure_for_performance(test_ds,seed, batch_size= batch_size)
```

```
    return train_ds, val_ds, test_ds, X_test, y_test

# Augmentations for Tensor set
# Dataset augmentation layers
data_augmentation_layers = [
        layers.RandomContrast(0.5), ]
data_augmentation = tf.keras.Sequential(
    data_augmentation_layers)

# CREATE AUGMENDED SETS FUNCTIONS
def  train_validation_augmentation(x_train,  y_train,  x_val,  y_val,  seed=42,
batch_size=1):
    image_data_gen_args = dict(rotation_range = 10,
                              width_shift_range = 0.1,
                              height_shift_range = 0.1,
                              shear_range = 0.1,
                              zoom_range = 0.2,
                              brightness_range = (0.2, 0.9),
                              fill_mode= 'nearest')
    # Train images
    image_data_generator = ImageDataGenerator(**image_data_gen_args)
    image_data_generator.fit(x_train, augment = True, seed = seed)

    train_image_generator = image_data_generator.flow(x_train, y_train,
                                                      seed = seed,
                                                      shuffle=True,
                                                      batch_size=batch_size)
    # Validation images
    valid_image_gen = ImageDataGenerator(**image_data_gen_args)
```

```
    valid_image_gen.fit(x_val, augment = True, seed = seed )
    valid_img_generator = valid_image_gen.flow(x_val, y_val,
                                               seed = seed,
                                               shuffle=True,
                                               batch_size=batch_size)
    return train_image_generator, valid_img_generator

def plot_Gerarator_Images(img_gen):
    for i in range(0,8):
        x = img_gen.next()
        image = x[i]
        plt.imshow(image[:,:,0])
        plt.show()

# Model's functions
# Model construction
def build_CNN_model(input_shape, start_neurons= 16, firstTwo_kernels = (7,7)):

    input_layer = tf.keras.Input(shape=input_shape)

    # Image augmentation block
    input_layer = data_augmentation(input_layer)

    input_resc = layers.Rescaling(scale=1./255)(input_layer)

    conv1a = layers.Conv2D(start_neurons * 1, firstTwo_kernels,
                           padding="same", activation="relu")(input_resc)
    batch1a = layers.BatchNormalization()(conv1a)
```

```
conv1b = layers.Conv2D(start_neurons * 1, firstTwo_kernels, padding="same"
                                                , activation="relu")(batch1a)
batch1b = layers.BatchNormalization()(conv1b)
pool1 = layers.MaxPooling2D(pool_size=(2, 2), strides=2)(batch1b)

conv2a = layers.Conv2D(start_neurons * 2, (3, 3), padding="same",
                                                activation="relu")(pool1)
batch2a = layers.BatchNormalization()(conv2a)

conv2b = layers.Conv2D(start_neurons * 2, (3, 3), padding="same",
                                            activation="relu")(batch2a)
batch2b = layers.BatchNormalization()(conv2b)
pool2 = layers.MaxPooling2D(pool_size=(2, 2), strides=2)(batch2b)

conv3a = layers.Conv2D(start_neurons * 4, (3, 3), padding="same",
                                              activation="relu")(pool2)
batch3a = layers.BatchNormalization()(conv3a)
conv3b = layers.Conv2D(start_neurons * 4, (3, 3), padding="same",
                                            activation="relu")(batch3a)
batch3b = layers.BatchNormalization()(conv3b)
pool3 = layers.MaxPooling2D(pool_size=(2, 2), strides=2)(batch3b)

conv4a = layers.Conv2D(start_neurons * 8, (3, 3), padding="same",
                                              activation="relu")(pool3)
batch4a = layers.BatchNormalization()(conv4a)
conv4b = layers.Conv2D(start_neurons * 8, (3, 3), padding="same",
                                            activation="relu")(batch4a)
batch4b = layers.BatchNormalization()(conv4b)
pool4 = layers.MaxPooling2D(pool_size=(2, 2), strides=2)(batch4b)
```

```
    flatten = layers.Flatten()(pool4)

    drop_f = layers.Dropout(0.4)(flatten)

    full_con1 = layers.Dense(start_neurons * 64, activation='relu',
                             kernel_regularizer=regularizers.l2(0.05))(drop_f)
    drop_1 = layers.Dropout(0.4)(full_con1)

    full_con2 = layers.Dense(start_neurons * 32, activation='relu',
                             kernel_regularizer=regularizers.l2(0.05))(drop_1)
    drop_2 = layers.Dropout(0.4)(full_con2)

    output_layer = layers.Dense(2, activation='sigmoid')(drop_2)

    return tf.keras.Model(input_layer,output_layer)

# Set Learning rate scheduler
# Learning Rate optimezer setup
def learning_scheduler(initial_rate):
    return  tf.keras.optimizers.schedules.InverseTimeDecay(
            initial_rate,
            decay_steps=STEPS_PER_EPOCH*5,
            decay_rate=1,
            staircase=False)

def get_optimizer(initial_rate):
    return tf.keras.optimizers.Adam(learning_scheduler(initial_rate))
```

```
# plot learning rate degradation
def plot_learning_schedule(initial_rate):
    step = np.linspace(0,2500)
    lr_schedule = learning_scheduler(initial_rate)
    lr = lr_schedule(step)
    plt.figure(figsize = (8,6))
    plt.plot(step/STEPS_PER_EPOCH, lr)
    plt.ylim([0,max(plt.ylim())])
    plt.xlabel('Epoch')
    _ = plt.ylabel('Learning Rate')

# Callbacks - Earling Stoping
# callbacks logs for each model
def get_callbacks(name, patience, monitor):
    return [
        tf.keras.callbacks.EarlyStopping(
            monitor= monitor,
            verbose=2,
            patience= patience,
            restore_best_weights= True),
        tf.keras.callbacks.TensorBoard(logdir/name),
    ]
def checkpointer(model_name):
    return tf.keras.callbacks.ModelCheckpoint(model_name, verbose=1,
                                              save_best_only=True)

# Model Setup
# Compiler
def models_compiler(model, optimizer,
```

```
loss=tf.keras.losses.BinaryCrossentropy(from_logits=False), metrics=
[tf.keras.metrics.BinaryAccuracy()]):
    return model.compile(optimizer=optimizer, loss=loss, metrics= metrics)

# Model fit
# without augmentation
def CNN_compile_fit(model, name, train_ds, val_ds, optimizer=None,
learning_rate=0.02, max_epochs=200, patience=5, monitor='val_loss',
class_weights=None, batch_size=1):
    if optimizer is None:
        optimizer = get_optimizer(learning_rate)

    models_compiler(model= model, optimizer= optimizer)

    model.summary()

    history = model.fit(
        train_ds,
        steps_per_epoch = STEPS_PER_EPOCH,
        epochs = max_epochs,
        validation_data = val_ds,
        class_weight = class_weights,
        callbacks = get_callbacks(name, patience, monitor),
        verbose=1,)
    return history

# with augmentation
def CNN_compile_fit_AUGM(model, name, train_generator, val_generator,
train_length, validation_length, optimizer=None, learning_rate=0.02,
```

```
max_epochs=200,    patience=5,    monitor='val_loss',    class_weights=None,
batch_size=1):
    if optimizer is None:
        optimizer = get_optimizer(learning_rate)

    models_compiler(model= model, optimizer= optimizer)

    model.summary()

    history = model.fit(
        train_generator,
        steps_per_epoch = STEPS_PER_EPOCH,
        epochs = max_epochs,
        validation_data = val_generator,
        #validation_steps = train_length,
        class_weight = class_weights,
        callbacks = get_callbacks(name, patience, monitor),
        verbose=1,)
    return history

# Model's Graphs
# Plotting Net Loss cs Val_Loss and Acuuracy vs Val_Accuracy
def plot_model_graphs(model_name, type_histories):

    loss_CNN = type_histories[model_name].history['loss']
    val_loss_CNN = type_histories[model_name].history['val_loss']

    epochs = range(1, len(loss_CNN) + 1)

    binary_accuracy_metric_CNN =
```

```
                        type_histories[model_name].history['binary_accuracy']
val_binary_accuracy_CNN =
                    type_histories[model_name].history['val_binary_accuracy']

plt.figure(figsize=(40,40))

ax = plt.subplot(2, 2,  1)
plt.plot(epochs, loss_CNN, 'b', label='Training loss')
plt.plot(epochs, val_loss_CNN, 'r', label='Validation loss')
plt.title(model_name + ' Training and Validation Loss', fontsize=30)
plt.xlabel('Epochs', fontsize=30)
plt.ylabel(' Loss', fontsize=30)
plt.legend(loc='lower right', fontsize=20)



ax = plt.subplot(2, 2,  2)
plt.plot(epochs, binary_accuracy_metric_CNN, 'b', label='Training Binary
                                                       Accuracy metric')
plt.plot(epochs, val_binary_accuracy_CNN, 'r', label='Validation Binary
                                                       Accuracy metric')
plt.title(model_name + ' Training and Validation Binary Accuracy metric',
                                                         fontsize=30)
plt.xlabel('Epochs', fontsize=30)
plt.ylabel('Binary Accuracy', fontsize=30)
plt.legend(loc='lower right', fontsize=20)

plt.show()

# Create Logs for the different nets
```

```
logdir = pathlib.Path(tempfile.mkdtemp())/"tensorboard_logs"
shutil.rmtree(logdir, ignore_errors=True)

#initialize model history dictionary
type_histories = {}

# RUN MODELS (extended mask paradigm 240 x 320)
# Insert Data
image_directory = 'D:/A4C/CNN/EXTENDED/images_full_info_extented'
mask_directory = 'D:/A4C/CNN/EXTENDED/masks_extended_full_info'
bsa_directory = 'D:/A4C/CNN/EXTENDED/bsa_data_full_info_extended.csv'

# Dataset BSA calculation
# BSA SETUP
bsa_data = upload_data(bsa_directory,
['image','height','weight','long_axis','area','volume','birth','gender'])

bsa_data.head()
bsa_data.tail()

# Body Mass Area (BSA) calculation per image
bsa_data_dict = create_bsa_data_dict(bsa_data)
bsa_data_dict
# BSA calculation
bsa_coef_dict = bsa_calculation(bsa_data_dict)
bsa_coef_dict

# Image Size 240 x 320
# Constants
```

```
IMG_WIDTH= 320
IMG_HEIGHT = 240
CHANNELS = 1
START_CONVS = 16
BATCH_SIZE = 32
SEED = 42

# Upload images and masks - Create masked images dataset, calculate labels
image_dataset, image_masked_dataset, mask_dataset, label_dataset, bsa_data_dict, classes, _, X_train, y_train, X_val, y_val, X_test, y_test = datasets_CNN_extended(image_directory, mask_directory, bsa_data_dict, IMG_WIDTH, IMG_HEIGHT)

# Shapes of uploaded original images, masks sets, labels
print_size_shapes(image_dataset, mask_dataset, image_masked_dataset, label_dataset, 'Original Images', 'Masks', 'Masked Images', 'Labels' )

# Print images, masks,masked-images and respective labels
print_sets_size_shapes(X_train, y_train, X_val, y_val, X_test, y_test)

# Data Inspection
bsa_data_dict
label_dataset
plot_images(image_dataset, mask_dataset, image_masked_dataset, label_dataset, lines=10)
classes

# Dataset for the model- Without Augmentation
```

```
train_set, val_set, test_set, test_images, test_true_labels = create_performance_dataset(X_train, y_train, X_val, y_val, X_test, y_test,seed=SEED, batch_size=BATCH_SIZE)

# Shapes of uploaded original images sets
# shape of train and label batch
for image_batch, labels_batch in train_set:
    print('Train batch size:', image_batch.shape)
    print('Training Labels batch size:', labels_batch.shape)
    break
for image_batch, labels_batch in val_set:
    print('Validation batch size:', image_batch.shape)
    print('Validation Labels batch size:', labels_batch.shape)
    break
for image_batch, labels_batch in test_set:
    print('Test batch size:', image_batch.shape)
    print('Test Labels batch size:', labels_batch.shape)
    break
train_Generator, validation_Generator =
train_validation_augmentation(X_train, y_train, X_val, y_val, seed=SEED, batch_size=BATCH_SIZE)

# Set Learning Rate
LEARNING_RATE = 0.0002
STEPS_PER_EPOCH = len(train_set)
# choose with augmented dataset
STEPS_PER_EPOCH = len(X_train)//BATCH_SIZE
plot_learning_schedule(LEARNING_RATE)

# RUN MODEL Extended masks 240 x 320
```

```
image_size = (IMG_HEIGHT , IMG_WIDTH)
cnn = build_CNN_model(input_shape= image_size + (CHANNELS,),
start_neurons= START_CONVS, firstTwo_kernels = (7,7))

# Run model with Datasets without augmentation
type_histories['cnn_extended_batch32_bigDim']= CNN_compile_fit(cnn,
'cnn_extended_batch32_bigDim', train_set,
val_set,learning_rate = LEARNING_RATE, max_epochs=1000, patience=50,
batch_size=BATCH_SIZE)

# Run model with Augmented Sets
type_histories['cnn_extended_batch32_bigDim_AUGM']=
CNN_compile_fit_AUGM( cnn, 'cnn_extended_batch32_bigDim_AUGM',
train_Generator, validation_Generator, len(X_train), len(X_val),
learning_rate=LEARNING_RATE,max_epochs=2500, patience=50,
batch_size=BATCH_SIZE)

# Save model
cnn.save('saved_CNN_models/cnn_extended_batch32_bigDim')
cnn.save('saved_CNN_models/cnn_extended_batch32_bigDim_AUGM')

#  Plot model
plot_model_graphs("cnn_extended_batch32_smallDim", type_histories)
plot_model_graphs("cnn_extended_batch32_smallDim_AUGM",
type_histories)
```

1.4.2 Code for Evaluating CNN Models

```
# Dataset Creation Function Import
%run Datasets_Creation.ipynb
```

```
# Plot Functions
def plot_images(images,masks, masked_images, labels, lines=20):

  for i in range(100,120):
    image = random.randint(0, images.shape[0])
    print(list(labels.keys())[image])

    plt.figure(figsize=(16, 12))
    plt.subplot(131)
    plt.title(f'Original Image - {list(labels.values())[image]}')
    plt.imshow(images[image])
    plt.subplot(132)
    plt.title('Mask')
    plt.imshow(masks[image])
    plt.subplot(133)
    plt.title('Masked image')
    plt.imshow(masked_images[image])
    plt.show()

# Print dimensions
def print_size_shapes(images, masks, masked_images, labels, images_name,
masks_name, masked_images_name, labels_name ):
  # Dataset
  print(f'{images_name} Dataset size : {len(images)}')
  print(f'{images_name} shape: {images.shape}')
  print(f'Single {images_name} shape: {images[1].shape}')
  print(f'{masks_name} shape: {masks.shape}')
  print(f'Single {masks_name} shape: {masks[1].shape}')
  print(f'{masked_images_name} shape: {masked_images.shape}')
  print(f'Single {masked_images_name} shape: {masked_images[1].shape}')
```

```
    print(f'{labels_name} shape: {np.asarray(list(labels.values())).shape}')

#  Train, Validation,Test Set
def print_sets_size_shapes(X_train, y_train, X_val, y_val, X_test, y_test):
    # Train dataset
    print(f'Training Dataset size : {len(X_train)}')
    print(X_train.shape)
    print(X_train[1].shape)

    # Validation Dataset
    print(f'Validation Dataset size : {len(X_val)}')
    print(X_val.shape)
    print(X_val[1].shape)

    # Test Dataset
    print(f'Test Dataset size : {len(X_test)}')
    print(X_test.shape)
    print(X_test[1].shape)

# Create dataset for the model- Without Augmentation
# Creating tensors data.Datasets of Train, Validation, Test sets
def to_tensors(x_train, y_train, x_val, y_val, x_test, y_test):
    train_ds = tf.data.Dataset.from_tensor_slices( (x_train, y_train) )
    val_ds = tf.data.Dataset.from_tensor_slices( (x_val, y_val) )
    test_ds = tf.data.Dataset.from_tensor_slices( (x_test, y_test) )
    return train_ds,val_ds,test_ds

# cache, shuffle, prefetch
def configure_for_performance(ds, seed, buffer_size= 1000, batch_size=1):
```

```
 ds = ds.cache()
ds = ds.shuffle(buffer_size= buffer_size, seed=seed)
ds = ds.batch(batch_size)
ds = ds.prefetch(buffer_size= tf.data.AUTOTUNE)
return ds

def create_performance_dataset(X_train, y_train, X_val, y_val, X_test,
y_test, seed=42, batch_size=1):

train_ds, val_ds, test_ds  = to_tensors(X_train, y_train, X_val, y_val,
X_test,
        y_test)



train_ds = configure_for_performance(train_ds,seed, batch_size=
                                                        batch_size)
val_ds = configure_for_performance(val_ds,seed, batch_size= batch_size)
test_ds = configure_for_performance(test_ds,seed, batch_size= batch_size)
return train_ds, val_ds, test_ds, X_test, y_test

# CREATE  SETS FOR AUGMENDED MODELS
def train_validation_augmentation(x_train, y_train, x_val, y_val, seed=42,
batch_size=1):
image_data_gen_args = dict(rotation_range = 10,
                    width_shift_range = 0.1,
                    height_shift_range = 0.1,
                    shear_range = 0.1,
                    zoom_range = 0.2,
                    brightness_range = (0.2, 0.9),
                    fill_mode= 'nearest')
```

```
    # Train images
    image_data_generator = ImageDataGenerator(**image_data_gen_args)
    image_data_generator.fit(x_train, augment = True, seed = seed)

    train_image_generator = image_data_generator.flow(x_train, y_train,
                                                      seed = seed,
                                                      shuffle=True,
                                                      batch_size=batch_size)

    # Validation images
    valid_image_gen = ImageDataGenerator(**image_data_gen_args)
    valid_image_gen.fit(x_val, augment = True, seed = seed )
    valid_img_generator = valid_image_gen.flow(x_val, y_val,
                                               seed = seed,
                                               shuffle=True,
                                               batch_size=batch_size)
    return train_image_generator, valid_img_generator

# EVALUATION FUNCTIONS
# Take trained CNN_model mode
def get_CNN_model(CNN_model_name):
    return models.load_model(CNN_model_name)

# Evaluate CNN_model on unknown Test set
# without augmentation
def print_CNN_test_evaluation(CNN_model_name, CNN_model, test_set,
verbose=2):
    _, acc = CNN_model.evaluate(test_set, verbose = verbose)
    print(f"Binary Accuracy evaluation of {CNN_model_name} CNN model is:
    {acc*100.0} %  ")
```

```
# with augmentation
def  print_CNN_test_evaluation_AUGM(CNN_model_name,  CNN_model, X_test,y_test, batch_size, verbose=2):
    _, acc = CNN_model.evaluate(X_test,y_test, batch_size=batch_size, verbose
     = verbose)
    print(f"Binary Accuracy evaluation of {CNN_model_name} CNN model is:
    {acc*100.0} %  ")

# Calculate Predictions
def labels_to_strings(label):
    if label == 0:
        return 'abnormal'
    else:
        return 'normal'

def labels_to_binary(label):
    if label == 'abnormal':
        return 0
    else:
        return 1

# without augmentation
def calc_predictions(CNN_model,test_ds, binarizer):
    pred = CNN_model.predict(test_ds)
    predictions = binarizer.inverse_transform(pred)
    return np.asarray(predictions)

# with augmentation
def calc_predictions_AUGM(CNN_model,X_test, batch_size, binarizer):
```

```
    pred = CNN_model.predict(X_test, batch_size=batch_size)
    predictions = binarizer.inverse_transform(pred)
    return np.asarray(predictions)

# Check predicted vs true labels of images in Test set
def unbatch_set(dataset):
    unbatached_dataset = []
    for batch in dataset:
        for image in batch:
            unbatached_dataset.append(image)
    return np.asarray(unbatached_dataset)

def test_image_sunnity_check(test_images, test_true_labels, predictions,
binarizer, lines=9):
    test_labels = np.asarray(binarizer.inverse_transform(test_true_labels))
    predicted_labels = predictions
    plt.figure(figsize=(15, 15))
    for i in range(15):
        ax = plt.subplot(4, 4, i + 1)
        plt.imshow(test_images[i])
        plt.title(f'True: {test_labels[i]}\nPredicted: {predicted_labels[i]}')
        plt.axis("off")

def plot_confusion_matrix(model_name, test_true_labels, predictions,
binarizer):
    true_labels = np.asarray(binarizer.inverse_transform(test_true_labels))

    cm = confusion_matrix(true_labels, predictions, normalize='true')
    print(f"Confusion matrix:\n {cm}")
    display_cm = ConfusionMatrixDisplay(confusion_matrix=cm)
```

```
    display_cm.plot(cmap=plt.cm.Blues)
    plt.title(f'Confusion Matrix on {model_name} Test set \n')
    plt.show()

# Display Models metrics
def display_model_metrics(model_name, test_true_labels, predictions, binarizer):
    true_labels = np.asarray(binarizer.inverse_transform(test_true_labels))

    report = classification_report(true_labels, predictions,  zero_division=0 )
    print(f'Model\'s {model_name} metrics:\n {report}')

# Plot Auc-Roc curves
def plot_auc_roc(model_name, test_true_labels, predictions, binarizer, class_names=['abnormal','normal'],  average="macro", zero_division=0):
    true_labels = np.asarray(binarizer.inverse_transform(test_true_labels))
    true_labels = np.asarray(list(map(labels_to_binary, true_labels)))
    predictions = np.asarray(list(map(labels_to_binary, predictions)))

    false_positive, true_positive, threshold = roc_curve( true_labels, predictions)
    auc_score = auc(false_positive, true_positive)

    plt.figure(figsize=(10, 10))
    plt.style.use('seaborn')
    plt.plot(false_positive, true_positive, linestyle='--',color='darkorange', label=f"ROC curve (area = {'{:.2f}'.format(auc_score)})")
    plt.plot(false_positive, false_positive, linestyle='--', color='blue')
    plt.title('ROC curve for ' + model_name, fontsize=20)
    plt.xlim([0, 1])
```

```
    plt.ylim([0, 1])
    plt.xlabel('False Positive Rate', fontsize=20)
    plt.ylabel('True Positive Rate', fontsize=20)
    plt.legend(loc='lower right', fontsize=15)
    plt.show()


# EVALUATION OF THE MONDELS (Extended masks paradigm)
# Insert Data
image_directory = 'D:/A4C/CNN/EXTENDED/images_full_info_extented'
mask_directory = 'D:/A4C/CNN/EXTENDED/masks_extended_full_info'
bsa_directory = 'D:/A4C/CNN/EXTENDED/bsa_data_full_info_extended.csv'

# Dataset BSA calculation
# BSA SETUP
bsa_data = upload_data(bsa_directory,
['image','height','weight','long_axis','area','volume','birth','gender'])

bsa_data.head()
bsa_data.tail()

# Body Mass Area (BSA) calculation per image
bsa_data_dict = create_bsa_data_dict(bsa_data)
bsa_data_dict
# BSA calculation
bsa_coef_dict = bsa_calculation(bsa_data_dict)
bsa_coef_dict

# Image Size 240 x 320
```

```
# Constants
IMG_WIDTH= 320
IMG_HEIGHT = 240
CHANNELS = 1
START_CONVS = 16
BATCH_SIZE = 32
SEED = 42

# Upload images and masks - Create masked images dataset, calculate labels
image_dataset, image_masked_dataset, mask_dataset, label_dataset, bsa_data_dict, classes, _, X_train, y_train, X_val, y_val, X_test, y_test = datasets_CNN_extended(image_directory, mask_directory, bsa_data_dict, IMG_WIDTH, IMG_HEIGHT)

# Shapes of uploaded original images, masks sets, labels
print_size_shapes(image_dataset, mask_dataset, image_masked_dataset, label_dataset, 'Original Images', 'Masks', 'Masked Images', 'Labels' )

# Print images, masks,masked-images and respective labels
print_sets_size_shapes(X_train, y_train, X_val, y_val, X_test, y_test)

# Data Inspection
bsa_data_dict
label_dataset
plot_images(image_dataset, mask_dataset, image_masked_dataset, label_dataset, lines=10)
classes

# Dataset for the model- Without Augmentation
```

```
train_set, val_set, test_set, test_images, test_true_labels = create_performance_dataset(X_train, y_train, X_val, y_val, X_test, y_test,seed=SEED, batch_size=BATCH_SIZE)

# Shapes of uploaded original images sets
# shape of train and label batch
for image_batch, labels_batch in train_set:
    print('Train batch size:', image_batch.shape)
    print('Training Labels batch size:', labels_batch.shape)
    break
for image_batch, labels_batch in val_set:
    print('Validation batch size:', image_batch.shape)
    print('Validation Labels batch size:', labels_batch.shape)
    break
for image_batch, labels_batch in test_set:
    print('Test batch size:', image_batch.shape)
    print('Test Labels batch size:', labels_batch.shape)
    break
train_Generator, validation_Generator =
train_validation_augmentation(X_train, y_train, X_val, y_val, seed=SEED, batch_size=BATCH_SIZE)

# Evaluate MODEL Extended masks 240x320 - Without Augmentation
cnn_extended_batch32_bigDim = get_CNN_model
('saved_CNN_models/cnn_extended_batch32_bigDim')

print_CNN_test_evaluation("cnn_extended_batch32_bigDim",
cnn_extended_batch32_bigDim, test_set)
```

```
cnn_extended_batch32_bigDim_predictions =
calc_predictions(cnn_extended_batch32_bigDim, test_set, label_binarizer)
cnn_extended_batch32_bigDim_predictions

test_image_sunnity_check(test_images, test_true_labels,
cnn_extended_batch32_bigDim_predictions, label_binarizer)

# confusion matrix
plot_confusion_matrix('cnn_extended_batch32_bigDim',test_true_labels,
cnn_extended_batch32_bigDim_predictions, label_binarizer)

display_model_metrics('cnn_extended_batch32_bigDim',test_true_labels,
cnn_extended_batch32_bigDim_predictions,label_binarizer)



plot_auc_roc('CNN Extended Maks 320x240',test_true_labels,
cnn_extended_batch32_bigDim_predictions,label_binarizer)

# EVALUATE AUGMENTED MODEL Extended masks 240x320
cnn_extended_batch32_bigDim_AUGM = get_CNN_model
( 'saved_CNN_models/cnn_extended_batch32_bigDim_AUGM')

print_CNN_test_evaluation_AUGM("cnn_extended_batch32_bigDim_AUGM
", cnn_extended_batch32_bigDim_AUGM, X_test, y_test, BATCH_SIZE)

cnn_extended_batch32_bigDim_AUGM_predictions =
calc_predictions_AUGM(cnn_extended_batch32_bigDim_AUGM, X_test,
BATCH_SIZE, label_binarizer)
cnn_extended_batch32_bigDim_AUGM_predictions
```

```
test_image_sunnity_check(X_test, y_test,
cnn_extended_batch32_bigDim_AUGM_predictions, label_binarizer)

plot_confusion_matrix('cnn_extended_batch32_bigDim_AUGM',y_test,
cnn_extended_batch32_bigDim_AUGM_predictions, label_binarizer)

display_model_metrics('cnn_extended_batch32_bigDim_AUGM',y_test,
cnn_extended_batch32_bigDim_AUGM_predictions,label_binarizer)

plot_auc_roc('CNN Augmented Extended Masks 240x320',y_test,
cnn_extended_batch32_bigDim_AUGM_predictions,label_binarizer)
```

1.5 CNN-UNET Pipeline Evaluation Code

```
# Dataset Creation Functions Import
%run Datasets_Creation.ipynb

# Plot Functions
def plot_images_UNET(image_set, mask_set, lines=6):
  image_indexs = []
  plt.figure(figsize=(15, 15))
  for i in range(lines):
    plt.figure(figsize=(15, 15))
    image = random.randint(0, len(image_set))
    ax = plt.subplot(121)
    plt.imshow(image_set[image,:,:,0])
    ax = plt.subplot(122)
    plt.imshow(mask_set[image,:,:,0])
    plt.axis("off")

def plot_images_CNN(images,masks, masked_images, labels):
```

```
  for image in range(len(images)):
    print(labels[image])

    plt.figure(figsize=(16, 12))
    plt.subplot(131)
    plt.title(f'Original Image - {labels[image]}')
    plt.imshow(images[image])
    plt.subplot(132)
    plt.title('Mask')
    plt.imshow(masks[image])
    plt.subplot(133)
    plt.title('Masked image')
    plt.imshow(masked_images[image])
    plt.show()

def plot_images_CNN_original_masks(images,masks, masked_images,
labels):
  for image in range(len(images)):
    print(labels[image])

    plt.figure(figsize=(16, 12))
    plt.subplot(131)
    plt.title(f'Original Image - {labels[image]}')
    plt.imshow(images[image])
    plt.subplot(132)
    plt.title('Mask')
    plt.imshow(masks[image])
    plt.subplot(133)
    plt.title('Predicted Mask')
    plt.imshow(masked_images[image])
```

```
    plt.show()

# Print dimensions
def print_size_shapes(images, masks, masked_images, labels, images_name,
masks_name, masked_images_name, labels_name ):
  # Dataset
  print(f'{images_name} Dataset size : {len(images)}')
  print(f'{images_name} shape: {images.shape}')
  print(f'Single {images_name} shape: {images[1].shape}')
  print(f'{masks_name} shape: {masks.shape}')
  print(f'Single {masks_name} shape: {masks[1].shape}')
  print(f'{masked_images_name} shape: {masked_images.shape}')
  print(f'Single {masked_images_name} shape: {masked_images[1].shape}')
  print(f'{labels_name} shape: {np.asarray(list(labels.values())).shape}')

#  Train, Validation,Test Set
def print_sets_size_shapes(X_train, y_train, X_val, y_val, X_test, y_test):
  # Train dataset
  print(f'Training Dataset size : {len(X_train)}')
  print(X_train.shape)
  print(X_train[1].shape)


  # Validation Dataset
  print(f'Validation Dataset size : {len(X_val)}')
  print(X_val.shape)
  print(X_val[1].shape)


  # Test Dataset
```

```
    print(f'Test Dataset size : {len(X_test)}')
    print(X_test.shape)
    print(X_test[1].shape)

# EVALUATION FUNCTIONS
# UNET metric and Loss
def jaccard_index(masks_true, masks_predicted):
    masks_true_flatten = Kb.flatten(masks_true)
    masks_predicted_flatten = Kb.flatten(masks_predicted)
    intersection = Kb.sum(masks_true_flatten * masks_predicted_flatten)
    return (intersection + 10.0) / (Kb.sum(masks_true_flatten) +
Kb.sum(masks_predicted_flatten) - intersection + 10.0)

def jaccard_loss(y_true, y_pred, p_value=1.75,smooth = 10):
    y_true_f = Kb.flatten(y_true)
    y_pred_f = Kb.flatten(y_pred)

    intersection = Kb.sum(y_true_f * y_pred_f)
    term_true = Kb.sum(Kb.pow(y_true_f, p_value))
    term_pred = Kb.sum(Kb.pow(y_pred_f, p_value))
    union = term_true + term_pred - intersection
    return 1 - ((intersection + smooth) / (union + smooth))

# Take UNET and CNN_model model
def get_unet_model(model_name):
    return models.load_model(model_name,
custom_objects={"jaccard_loss":jaccard_loss,"jaccard_index":jaccard_index})

def get_CNN_model(CNN_model_name):
    return models.load_model(CNN_model_name)
```

```
# UNET predictions
def calc_unet_predictions_AUGM(unet_model,X_test, batch_size,
threshold=0.5):
   masks_predictions = unet_model.predict(X_test, batch_size=batch_size)
   masks_pred_thresholed = masks_predictions > threshold
   return masks_pred_thresholed.astype(np.uint8)

def create_masked_images(images, masks):
   masked_image_dataset = []
   for i in range(len(images)):
      image = cv2.bitwise_and(images[i], images[i], mask= masks[i])
      masked_image_dataset.append(np.expand_dims(image,2))
   return np.asarray(masked_image_dataset).astype(np.uint8)

def create_masked_image(test_images, mask_predictions):
   return cv2.bitwise_and(test_images, mask_predictions, mask= mask)

def test_image_sunnity_check_UNET(X_test,mask_test,
model_masks_predicted):

   for i in range(X_test.shape[0]):
      test_img = X_test[i]
      ground_truth = mask_test[i]
      test_img_input=np.expand_dims(test_img, 0)
      prediction_mask = model_masks_predicted[i].astype(np.uint8)

      predicted_masked_image = create_masked_image(test_img,
       prediction_mask)
```

```
    print (f'Test Image : {i+1}')
    plt.figure(figsize=(16, 12))
    plt.subplot(241)
    plt.title('Testing Image')
    plt.imshow(test_img)
    plt.subplot(242)
    plt.title('Testing Label')
    plt.imshow(ground_truth)
    plt.subplot(243)
    plt.title('Prediction on test image')
    plt.imshow(prediction_mask)
    plt.subplot(244)
    plt.title('Image with prediction mask')
    plt.imshow(predicted_masked_image)
    plt.show()

  # Evaluate CNN_model on unknown Test set
def print_CNN_test_evaluation_AUGM(CNN_model_name, CNN_model,
X_test,y_test, batch_size, verbose=2):
  _, acc = CNN_model.evaluate(X_test,y_test, batch_size=batch_size, verbose
                                                              = verbose)
  print(f"Binary Accuracy evaluation of {CNN_model_name} CNN model is:
{acc*100.0} %  ")

# Calculate CNN Predictions
def calc_predictions_AUGM(CNN_model,X_test, batch_size, binarizer):
  pred = CNN_model.predict(X_test, batch_size=batch_size)
  predictions = binarizer.inverse_transform(pred)
  return np.asarray(predictions)
```

```
# Check predicted vs true labels of images in Test set
def test_image_sunnity_check_CNN(test_images, test_true_labels,
predictions, binarizer, lines=9):
    test_labels = np.asarray(binarizer.inverse_transform(test_true_labels))
    predicted_labels = predictions

    plt.figure(figsize=(15, 15))
    for i in range(15):
        ax = plt.subplot(4, 4, i + 1)
        plt.imshow(test_images[i])
        plt.title(f'True: {test_labels[i]}\nPredicted: {predicted_labels[i]}')
        plt.axis("off")

def plot_confusion_matrix(model_name, test_true_labels, predictions,
binarizer):
    true_labels = np.asarray(binarizer.inverse_transform(test_true_labels))

    cm = confusion_matrix(true_labels, predictions, normalize='true')
    print(f"Confusion matrix:\n {cm}")
    display_cm = ConfusionMatrixDisplay(confusion_matrix=cm)

    display_cm.plot(cmap=plt.cm.Blues)
    plt.title(f'Confusion Matrix on {model_name} Test set \n')
    plt.show()

# Display Models metrics
def display_model_metrics(model_name, test_true_labels, predictions,
binarizer):
    true_labels = np.asarray(binarizer.inverse_transform(test_true_labels))
```

```
    report = classification_report(true_labels, predictions,  zero_division=0 )
    print(f'Model\'s {model_name} metrics:\n {report}')

def labels_to_binary(label):
    if label == 'abnormal':
        return 0
    else:
        return 1



# Plot Auc-Roc curves
def plot_auc_roc(model_name, test_true_labels, predictions, binarizer, class_names=['abnormal','normal'],  average="macro", zero_division=0):
    true_labels = np.asarray(binarizer.inverse_transform(test_true_labels))
    true_labels = np.asarray(list(map(labels_to_binary, true_labels)))
    predictions = np.asarray(list(map(labels_to_binary, predictions)))

    false_positive, true_positive, threshold = roc_curve( true_labels, predictions)
    auc_score = auc(false_positive, true_positive)

    plt.figure(figsize=(10, 10))
    plt.style.use('seaborn')
    plt.plot(false_positive, true_positive, linestyle='--',color='darkorange', label=f"ROC curve (area = {'{:.2f}'.format(auc_score)})")
    plt.plot(false_positive, false_positive, linestyle='--', color='blue')
    plt.title('ROC curve for ' + model_name, fontsize=20)
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.xlabel('False Positive Rate', fontsize=20)
    plt.ylabel('True Positive Rate', fontsize=20)
```

```
    plt.legend(loc='lower right', fontsize=15)
    plt.show()


# EVALUATION OF THE MONDELS PIPELINE
#(Extended Masks Paradigm 120 x 160 with augmentation)
# Insert Data
image_directory = 'D:/A4C/CNN/EXTENDED/images_full_info_extented'
mask_directory = 'D:/A4C/CNN/EXTENDED/masks_extended_full_info'
bsa_directory = 'D:/A4C/CNN/EXTENDED/bsa_data_full_info_extended.csv'

# Upload images and masks - Create masked images dataset, calculate labels
image_dataset, mask_dataset,image_masked_dataset, X_train, mask_train,
y_train, X_val, mask_val, y_val, X_test, mask_test, y_test, label_dataset,
bsa_data_dict, classes, label_binarizer =
datasets_UNET(image_directory, mask_directory,bsa_directory,
IMG_WIDTH, IMG_HEIGHT)

# Shapes of uploaded original images, masks sets, labels
print_size_shapes(image_dataset, mask_dataset, image_masked_dataset,
label_dataset, 'Original Images', 'Masks', 'Masked Images', 'Labels' )

# Print images, masks,masked-images and respective labels
print_sets_size_shapes(X_train, y_train, X_val, y_val, X_test, y_test)

# Data Inspection
bsa_data_dict
label_dataset

# Plot images, masks,masked-images and respective labels
```

```
plot_images_UNET(image_dataset, mask_dataset, lines=6)
classes

# EVALUATE Unet - CNN Pipeline
unet_model = get_unet_model
('models/unet_extended_batch8_bigDim_AUGM')

# Predictions for Image Generator Augmented Model
unet_masks_predictions = calc_unet_predictions_AUGM(unet_model,X_test,
BATCH_SIZE)

test_masked_images = create_masked_images(X_test,
unet_masks_predictions)

plot_images_CNN(X_test, unet_masks_predictions, test_masked_images,
y_test)

# Cnn evaluation with predicted masked images
BATCH_SIZE = 32
cnn_model = get_CNN_model
('models/cnn_extended_batch32_bigDim_AUGM')

print_CNN_test_evaluation_AUGM("cnn_extended_batch32_smallDim_AUG
M", cnn_model, test_masked_images, y_test, BATCH_SIZE)

cnn_model_predictions = calc_predictions_AUGM(cnn_model,
test_masked_images, BATCH_SIZE, label_binarizer)

cnn_model_predictions
```

```
test_image_sunnity_check_CNN(test_masked_images, y_test,
cnn_model_predictions, label_binarizer)

plot_confusion_matrix('cnn_extended_batch32_smallDim_AUGM',y_test,
cnn_model_predictions, label_binarizer)

display_model_metrics('cnn_extended_batch32_smallDim_AUGM',y_test,
cnn_model_predictions,label_binarizer)

plot_auc_roc('Unet-CNN Augmented Extended Masks 240x320',y_test,
cnn_model_predictions,label_binarizer)
```

1.6 GAN CODE

1.6.1 Code for Train GAN

```
# Dataset Creation Functions import
%run Datasets_Creation.ipyn

# Plot Datasets Functions
def plot_images(images, labels, lines=10):

  for i in range(lines):
    image = random.randint(0, images.shape[0])
    plt.figure(figsize=(16, 12))
    plt.title(f'Original Image - {list(labels.values())[image]}')
    plt.imshow(images[image])
    plt.show()

# Print dimensions
def print_size_shapes(images, labels, images_name,labels_name ):
```

```
    # Dataset
    print(f'{images_name} Dataset size : {len(images)}')
    print(f'{images_name} shape: {images.shape}')
    print(f'Single {images_name} shape: {images[1].shape}')
    print(f'{labels_name} shape: {np.asarray(list(labels.values())).shape}')

def print_sets_size_shapes(X_sup, y_sup, X_unsup, y_unsup, X_test, y_test):
    # supervised dataset
    print(f'Supervised Dataset size : {len(X_sup)}')
    print(X_sup.shape)
    print(y_sup.shape)

    # unsupervised dataset
    print(f'Unsupervised Dataset size : {len(X_unsup)}')
    print(X_unsup.shape)
    print(y_unsup.shape)

    # Test Dataset
    print(f'Test Dataset size : {len(X_test)}')
    print(X_test.shape)
    print(y_test.shape)

# Model's functions
# Model construction
def build_generator(noise_shape=(1,1,100), input_shape=(110,110,1) ,
start_neurons= 16, n_classes=2):

    noise_input = tf.keras.Input(noise_shape)
   deconv1 = layers.Conv2DTranspose(start_neurons * 64, (3, 3), strides=2,
                                                          activation='relu')(noise_input)
```

```
    batch1 = layers.BatchNormalization()(deconv1)

    deconv2 = layers.Conv2DTranspose(start_neurons * 32, (3, 3), strides=2,
activation='relu')(batch1)
    batch2 = layers.BatchNormalization()(deconv2)

    deconv3 = layers.Conv2DTranspose(start_neurons * 16, (3, 3), strides=2,
activation='relu')(batch2)
    batch3 = layers.BatchNormalization()(deconv3)

    deconv4 = layers.Conv2DTranspose(start_neurons * 8, (3, 3), strides=2,
activation='relu')(batch3)
    batch4 = layers.BatchNormalization()(deconv4)

    deconv5 = layers.Conv2DTranspose(start_neurons * 4, (3, 3), strides=2,
activation='relu')(batch4)
    batch5 = layers.BatchNormalization()(deconv5)

    deconv6 = layers.Conv2DTranspose(start_neurons * 4, (3, 3), strides=2,
activation='relu')(batch5)
    batch6 = layers.BatchNormalization()(deconv5)

    deconv7 = layers.Conv2DTranspose(start_neurons * 4, (3, 3), strides=2,
activation='relu')(batch6)
    batch7 = layers.BatchNormalization()(deconv7)

    fake_img = layers.Conv2DTranspose(1, (4, 4), activation='tanh')(batch7)
    model = tf.keras.Model(noise_input,fake_img)
    return model
```

```
def build_discriminator(input_shape , start_neurons= 16, n_clasess=2):

   input_layer = tf.keras.Input(shape=input_shape)


   conv_1a = layers.Conv2D(start_neurons * 4, (3, 3),strides=1)(input_layer) #,
   batch_1a = layers.BatchNormalization()(conv_1a)
   activ_1a = layers.LeakyReLU()(batch_1a)
   conv_1b = layers.Conv2D(start_neurons * 4, (3, 3),strides=1)(activ_1a) #,
   batch_1b = layers.BatchNormalization()(conv_1b)
   activ_1b = layers.LeakyReLU()(batch_1b)
   conv_1c = layers.Conv2D(start_neurons * 4, (3, 3),strides=2)(activ_1b) #,
   batch_1c = layers.BatchNormalization()(conv_1c)
   activ_1c = layers.LeakyReLU()(batch_1c)
   drop_1 = layers.Dropout(0.4)(activ_1c)

   conv_2a = layers.Conv2D(start_neurons * 8, (3, 3),strides=1)(drop_1) #,
   batch_2a = layers.BatchNormalization()(conv_2a)
   activ_2a = layers.LeakyReLU()(batch_2a)
   conv_2b = layers.Conv2D(start_neurons * 8, (3, 3),strides=1)(activ_2a) #,
   batch_2b = layers.BatchNormalization()(conv_2b)
   activ_2b = layers.LeakyReLU()(batch_2b)
   conv_2c = layers.Conv2D(start_neurons * 8, (3, 3),strides=2)(activ_2b) #,
   batch_2c = layers.BatchNormalization()(conv_2c)
   activ_2c = layers.LeakyReLU()(batch_2c)
   drop_2 = layers.Dropout(0.4)(activ_2c)

   conv_3a = layers.Conv2D(start_neurons * 16, (3, 3),strides=1)(drop_2) #,
```

```
batch_3a = layers.BatchNormalization()(conv_3a)
activ_3a = layers.LeakyReLU()(batch_3a)
conv_3b = layers.Conv2D(start_neurons * 16, (3, 3),strides=1)(activ_3a) #,
batch_3b = layers.BatchNormalization()(conv_3b)
activ_3b = layers.LeakyReLU()(batch_3b)
conv_3c = layers.Conv2D(start_neurons * 16, (3, 3),strides=2)(activ_3b) #,
batch_3c = layers.BatchNormalization()(conv_3c)
activ_3c = layers.LeakyReLU()(batch_3c)
drop_3 = layers.Dropout(0.4)(activ_3c)

conv_4a = layers.Conv2D(start_neurons * 32, (3, 3),strides=1)(drop_3) #,
batch_4a = layers.BatchNormalization()(conv_4a)
activ_4a = layers.LeakyReLU()(batch_4a)
conv_4b = layers.Conv2D(start_neurons * 32, (3, 3),strides=1)(activ_4a) #,
batch_4b = layers.BatchNormalization()(conv_4b)
activ_4b = layers.LeakyReLU()(batch_4b)
conv_4c = layers.Conv2D(start_neurons * 32, (3, 3),strides=2)(activ_4b) #,
batch_4c = layers.BatchNormalization()(conv_4c)
activ_4c = layers.LeakyReLU()(batch_4c)
drop_4 = layers.Dropout(0.4)(activ_4c)
pool = layers.MaxPooling2D(pool_size=(2, 2), strides=2)(drop_4)

flatten = layers.Flatten()(pool)
drop_f = layers.Dropout(0.5)(flatten)

full_con = layers.Dense(start_neurons * 64)(drop_f )
activ_full = layers.LeakyReLU()(full_con)
output_layer = layers.Dense(n_clasess)(activ_full)
model = tf.keras.Model(input_layer, output_layer)
```

```
    return model
def supervised_discriminator(disc_model):
    model = Sequential()
    model.add(disc_model)
    model.add(layers.Activation('sigmoid'))

    opt = tf.keras.optimizers.Adam(learning_rate=0.0003)
    model.compile(loss='binary_crossentropy', optimizer = opt,
                                    metrics=[tf.keras.metrics.BinaryAccuracy()])
    return model

def unsupervised_discriminator(disc_model):
    model = Sequential()
    model.add(disc_model)
    model.add(layers.Activation('sigmoid'))
    #model.add(layers.Lambda(norm_activation))

    opt = tf.keras.optimizers.Adam(learning_rate=0.0003)
    model.compile(loss='binary_crossentropy', optimizer = opt,
                          metrics=[tf.keras.metrics.BinaryAccuracy()])
    return model

def compined_gan(generator, unsupervided_disciminator):
    unsupervided_disciminator.trainable = False
    # get image output from the generator model
    gen_output = generator.output

    # generator image output and corresponding inputlabelare inputs to the
      discriminator
```

```
    gan_output = unsupervided_discriminator(gen_output)

    model = tf.keras.Model(generator.input, gan_output)

    opt = tf.keras.optimizers.Adam(learning_rate=0.0003)
    model.compile(loss=tf.keras.losses.MeanSquaredError(), optimizer = opt,
                                          metrics=[tf.keras.metrics.BinaryAccuracy()])
    return model

def check_performance(step, gen_model, sup_disc, unsup_disc, gen_input,
images,labels, epoch, batch=32):

    X,_ = generate_fake_images(gen_model, gen_input,batch)

    X = (X + 1) / 2.0

    for i in range(20):
        plt.subplot(5, 5, i + 1)
        plt.axis("off")
        plt.imshow(X[i, :, :, 0], cmap = 'gray_r')

    filename1 = 'gan_images/generated_plot_'+ str(epoch) + '.png'
    plt.savefig(filename1)

    #X_real, y_real= images, labels
    _, acc = sup_disc.evaluate(images,labels, verbose=0)
    print('Discriminator accuracy: %.3f%%' % (acc * 100))

    filename2 = 'saved_GAN_models/gen_model_'+ str(epoch)
    gen_model.save(filename2)
```

```
    filename3= 'saved_Sup_Discriminator_models/sup_disc_'+ str(epoch)
    sup_disc.save(filename3)

    filename4= 'saved_unSup_Discriminator_models/unsup_disc_'+ str(epoch)
    unsup_disc.save(filename4)

    print(f'> Saved: {filename1} , {filename2}, {filename3}, {filename4} ')

def train_gan(generator, unsup_disc, sup_disc, gan_model, X_train, y_train,
X_sup, y_sup, X_unsup, y_unsup, X_test,y_test, sup_length, gen_input= 100,
epochs=50, n_batch=32, seed=42, start=0):

    batch_per_epoch = int(sup_length / n_batch)

    n_steps = batch_per_epoch * epochs

    half_batch = int(n_batch / 2)
    print(f'n_epochs={epochs}, n_batch={n_batch}, 1/2={half_batch},
b/e={batch_per_epoch}, steps={n_steps}')

    for i in range(n_steps):

        [X_sup_real, y_sup_real], _ = generate_real_images(X_sup, y_sup,
                                                            half_batch)
        sup_loss, sup_acc = sup_disc.train_on_batch(X_sup_real, y_sup_real)

        [X_unsup_real, _], y_unsup_real = generate_real_images(X_train,
                                                        y_train, half_batch)
        d_loss_real = unsup_disc.train_on_batch(X_unsup_real, y_unsup_real)
```

```
        X_fake, y_fake = generate_fake_images(generator, gen_input, half_batch)
        d_loss_fake = unsup_disc.train_on_batch(X_fake, y_fake)

        X_gan, y_gan = generate_noise_points(gen_input, n_batch),
                                                        ones((n_batch, 1))
        gan_loss = gan_model.train_on_batch(X_gan, y_gan)

        print(f'> {i+1}, c[{sup_loss}, {sup_acc*100}], D[{d_loss_real},
                                                {d_loss_fake}], G[{gan_loss}]')
        if (i+1) % (batch_per_epoch * 50) == 0:
            epoch = int((i+1) / (batch_per_epoch))
            check_performance(i, generator, sup_disc, unsup_disc, gen_input,
                                        X_train, y_train, epoch + start, n_batch)

def get_trained_models(sup_disc_model, unsup_disc_model, gen_model):
    return models.load_model(sup_disc_model) ,
           models.load_model(unsup_disc_model),
           models.load_model(gen_model)

# RUN MODEL
# Constants
IMG_WIDTH= 130
IMG_HEIGHT = 130
CHANNELS = 1
START_CONVS = 16
BATCH_SIZE = 32
SEED = 42
GEN_INPUT = 100
```

```
# Insert Data
image_directory = 'D:/A4C/CNN/EXTENDED/images_full_info_extented'
mask_directory = 'D:/A4C/CNN/EXTENDED/masks_extended_full_info'
bsa_directory = 'D:/A4C/CNN/EXTENDED/bsa_data_full_info_extended.csv'

# Dataset BSA calculation
bsa_data = upload_data(bsa_directory,
['image','height','weight','long_axis','area','volume','birth','gender'])

bsa_data.tail()

bsa_data_dict = create_bsa_data_dict(bsa_data)

# BSA calculation
bsa_coef_dict = bsa_calculation(bsa_data_dict)

# Upload images - Create supervised, unsupervised datasets, calculate labels
image_dataset, X_train, y_train, X_sup, y_sup, X_unsup, y_unsup, X_test,
y_test, label_dataset, bsa_data_dict, classes, label_binarizer=
datasets_GAN(image_directory, bsa_data_dict, IMG_WIDTH, IMG_HEIGHT)

# Shapes of uploaded original images, masks sets, labels
print_size_shapes(image_dataset, label_dataset, 'Original Images', 'Labels' )

print_sets_size_shapes(X_sup, y_sup, X_unsup, y_unsup, X_test, y_test)

# Plot images and respective ladels
plot_images(image_dataset, label_dataset, lines=10)
```

```
# Create Models
image_size = (IMG_HEIGHT , IMG_WIDTH)

discriminator = build_discriminator(input_shape= image_size +
(CHANNELS,))
sup_disc = supervised_discriminator(discriminator)
unsup_disc = unsupervised_discriminator(discriminator)

generator = build_generator()
gan_model = compined_gan(generator, unsup_disc)

total_images = X_sup.shape[0]
print(f'Total supervised images : {total_images}')

discriminator.summary()

sup_disc.summary()

generator.summary()

# Train
train_gan(generator, unsup_disc, sup_disc, gan_model, X_train, y_train,
X_sup, y_sup, X_unsup,y_unsup, X_test, y_test, total_images, gen_input=
GEN_INPUT, epochs=300, n_batch=BATCH_SIZE)
```

1.6.1 Code for Evaluating GAN's supervised Discriminator

```
# Dataset Creation Functions Import
%run Datasets_Creation.ipynb
```

```
def labels_to_binary(label):
    if label == 'abnormal':
        return 0
    else:
        return 1

# Plot Datasets Function
def plot_images(images, labels, lines=10):

    #plt.figure(figsize=(15, 15))
    for i in range(lines):
        image = random.randint(0, images.shape[0])
        plt.figure(figsize=(16, 12))
        plt.title(f'Original Image - {list(labels.values())[image]}')
        plt.imshow(images[image])
        plt.show()

# Print dimensions
def print_size_shapes(images, labels, images_name,labels_name ):
    # Dataset
    print(f'{images_name} Dataset size : {len(images)}')
    print(f'{images_name} shape: {images.shape}')
    print(f'Single {images_name} shape: {images[1].shape}')
    print(f'{labels_name} shape: {np.asarray(list(labels.values())).shape}')

def print_sets_size_shapes(X_sup, y_sup, X_unsup, y_unsup, X_test, y_test):
    # supervised dataset
    print(f'Training Dataset size : {len(X_sup)}')
    print(X_sup.shape)
```

```
    print(y_sup.shape)

    # unsupervised dataset
    print(f'Training Dataset size : {len(X_unsup)}')
    print(X_unsup.shape)
    print(y_unsup.shape)


    # Test Dataset
    print(f'Test Dataset size : {len(X_test)}')
    print(X_test.shape)
    print(y_test.shape)

# Supervised Discriminator Evaluation Functions
# Take Supervised Discriminatorl model
def get_sup_Discr_model(sup_Discr_model):
    return models.load_model(sup_Discr_model)

# Evaluate CNN_model on unknown Test set
def print_sup_Discr_test_evaluation(sup_Discr_model_name,
sup_Discr_model, X_test,y_test, batch_size, verbose=2):
    _, acc = sup_Discr_model.evaluate(X_test,y_test, batch_size=batch_size,
                                                        verbose = verbose)
    print(f"Binary Accuracy evaluation of {sup_Discr_model_name} Supervised
                                   Discriminator model is: {acc*100.0} %  ")

# Calculate Predictions
def calc_predictionsM(sup_Discr_model,X_test, batch_size, binarizer):
    pred = sup_Discr_model.predict(X_test, batch_size=batch_size)
```

```
    predictions = binarizer.inverse_transform(pred)

    return np.asarray(predictions)

# Check predicted vs true labels of images in Test set
def supervised_discriminator(disc_model):
    model = Sequential()
    model.add(disc_model)
    model.add(layers.Activation('softmax'))

    opt = tf.keras.optimizers.Adam(learning_rate=0.0003)
    model.compile(loss='categorical_crossentropy', optimizer = opt,
                  metrics=[tf.keras.metrics.BinaryAccuracy()])

    return model

def test_image_sunnity_check(test_images, test_true_labels, predictions,
binarizer, lines=9):


    test_labels = np.asarray(binarizer.inverse_transform(test_true_labels))
    predicted_labels = predictions



    plt.figure(figsize=(15, 15))
    for i in range(15):
        ax = plt.subplot(4, 4, i + 1)
        plt.imshow(test_images[i])
        plt.title(f'True: {test_labels[i]}\nPredicted: {predicted_labels[i]}')
        plt.axis("off")
```

```
def plot_confusion_matrix(model_name, test_true_labels, predictions,
binarizer):
   true_labels = np.asarray(binarizer.inverse_transform(test_true_labels))

   cm = confusion_matrix(true_labels, predictions, labels=binarizer.classes_,
normalize='true' )
   print(f"Confusion matrix:\n {cm}")
   display_cm = ConfusionMatrixDisplay(confusion_matrix=cm)

   display_cm.plot(cmap=plt.cm.Blues)
   plt.title(f'Confusion Matrix on {model_name} Test set \n')
   plt.show()

# Display Models metrics
def display_model_metrics(model_name, test_true_labels, predictions,
binarizer):
   true_labels = np.asarray(binarizer.inverse_transform(test_true_labels))

   report = classification_report(true_labels, predictions,  zero_division=0)
   print(f'Model\'s {model_name} metrics:\n {report}')

# Plot Auc-Roc curves
def plot_auc_roc(model_name, test_true_labels, predictions, binarizer,
class_names=['abnormal','normal'],  average="macro", zero_division=0):

   true_labels = np.asarray(binarizer.inverse_transform(test_true_labels))
   true_labels = np.asarray(list(map(labels_to_binary, true_labels)))
   predictions = np.asarray(list(map(labels_to_binary, predictions)))

   false_positive, true_positive, threshold = roc_curve( true_labels, predictions)
```

```
    auc_score = auc(false_positive, true_positive)

    plt.figure(figsize=(10, 10))
    plt.style.use('seaborn')
    plt.plot(false_positive, true_positive, linestyle='--',color='darkorange',
                    label=f"ROC curve (area = {'{:.2f}'.format(auc_score)})")
    plt.plot(false_positive, false_positive, linestyle='--', color='blue')
    plt.title('ROC curve for ' + model_name, fontsize=20)
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.xlabel('False Positive Rate', fontsize=20)
    plt.ylabel('True Positive Rate', fontsize=20)
    plt.legend(loc='lower right', fontsize=15)
    plt.show()

#RUN EVALUATION
 # Constants
IMG_WIDTH= 130
IMG_HEIGHT = 130
CHANNELS = 1
START_CONVS = 16
BATCH_SIZE = 40
SEED = 42
GEN_INPUT = 100

# Insert Data
image_directory = 'D:/A4C/CNN/EXTENDED/images_full_info_extented'
mask_directory = 'D:/A4C/CNN/EXTENDED/masks_extended_full_info'
bsa_directory = 'D:/A4C/CNN/EXTENDED/bsa_data_full_info_extended.csv'
```

```
# Dataset BSA calculation
bsa_data = upload_data(bsa_directory,
['image','height','weight','long_axis','area','volume','birth','gender'])

bsa_data.head()

bsa_data.tail()

bsa_data_dict = create_bsa_data_dict(bsa_data)

bsa_data_dict

# BSA calculation
bsa_coef_dict = bsa_calculation(bsa_data_dict)
bsa_coef_dict

# Upload images - Create images datasets, calculate labels
image_dataset, X_train, y_train, X_sup, y_sup, X_unsup, y_unsup, X_test,
y_test, label_dataset, bsa_data_dict, classes, label_binarizer =
datasets_GAN(image_directory, bsa_data_dict, IMG_WIDTH, IMG_HEIGHT)

# Shapes of uploaded original images, masks sets, labels
print_size_shapes(image_dataset, label_dataset, 'Original Images', 'Labels' )

print_sets_size_shapes(X_sup, y_sup, X_train, y_train, X_test, y_test)

# Plot images and respective ladels
plot_images(image_dataset, label_dataset, lines=10)

# EVALUATE Supervised Discriminator MODEL
```

```
sup_discr_model = get_sup_Discr_model('saved_Sup_Discriminator_models/sup_disc_300')

print_sup_Discr_test_evaluation("sup_disc_300", sup_discr_model, X_test, y_test, BATCH_SIZE)

sup_discr_predictions = calc_predictionsM(sup_discr_model, X_test,
                                          BATCH_SIZE, label_binarizer)
sup_discr_predictions

plot_confusion_matrix('sup_disc_300',y_test, sup_discr_predictions,
                                                  label_binarizer)

display_model_metrics('sup_disc_300',y_test, sup_discr_predictions,
                                                  label_binarizer)

plot_auc_roc('sup_disc_300',y_test, sup_discr_predictions, label_binarizer)
```

1.7 DATASET STATISTICS

```
# Dataset Creation
# create bsa_data
def upload_data(bsa_file, columns=[]):
  df = pd.read_csv(bsa_file , sep=';', header=0, usecols= columns)[columns]
  df['gender'].replace('m','Male', inplace=True )
  df['gender'].replace('f','Female' , inplace=True)
  return df
```

```
def create_bsa_data_dict(dataframe):
    dataframe.set_index('image')
    data_dict = dataframe.to_dict('index')
    data_diction = {}
    for i in data_dict:
        data_diction[str(data_dict[i]['image'])]= {'Height':  data_dict[i]['height'],
                                                   'Weight':  data_dict[i]['weight'],
                                                   'Long_Axis':  data_dict[i]['long_axis'],
                                                   'Area':  data_dict[i]['area'],
                                                   'Volume':  data_dict[i]['volume'],
                                                   'Birth':  data_dict[i]['birth'],
                                                   'Gender':   data_dict[i]['gender'],
                                                   'BSA_coef': math.sqrt( ( data_dict[i]['height']) *
                                                                      data_dict[i]['weight'] )/60}
    return data_diction

# calculate Body Mass Area(BSA) coefficient
# bsa (body-surface-area) coeffiecent per (Mosteller's formula)
def bsa_calculation(data_diction):
    bsa_coef_dict = {}
    for image in data_diction.keys():
        bsa_coef_dict[image] = math.sqrt( (data_diction[image]['Height']) *
                                          data_diction[image]['Weight'] )/60
    return bsa_coef_dict

# Crete masked_images and normal, ubnormal labels
# Apply Original Masks to Image
def create_masked_image(images, masks):
    masked_image_dataset = []
```

```
    for i in range(len(images)):
        image = cv2.bitwise_and(images[i], images[i], mask= masks[i])
        masked_image_dataset.append(np.expand_dims(image,2))
    return np.asarray(masked_image_dataset)

def sort_list_asc(image_list):
    sorted_list = []
    for image_name in image_list:
        image_name= os.path.splitext(image_name)[0]
        sorted_list.append(int(image_name))
    return  sorted(sorted_list)

def      create_image_mask_label_sets(image_directory,      mask_directory,
bsa_data_dict, img_width, img_height ):
    image_dataset = []
    mask_dataset = []
    label_dataset = {}
    volume_over_bsa = {}

    images = os.listdir(image_directory)
    images = sort_list_asc(images)
    for image_name in images:
        mask = cv2.imread(mask_directory + '/' +  str(image_name) + '.jpg',
                                                cv2.IMREAD_UNCHANGED)
        mask = cv2.resize(mask, (img_width, img_height), interpolation =
                                                    cv2.INTER_NEAREST)
        mask_dataset.append(np.expand_dims(mask,2))



        image = cv2.imread(image_directory + '/' + str(image_name) + '.jpg',
```

```
                                    cv2.IMREAD_UNCHANGED)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    image = cv2.resize(image, (img_width, img_height), interpolation =
                                              cv2.INTER_NEAREST)
    image_dataset.append(np.expand_dims(image,2))

    volume_over_bsa =
                     round(float(bsa_data_dict[str(image_name)]['Volume'])
                      / bsa_data_dict[str(image_name)]['BSA_coef'],3)

    bsa_data_dict[str(image_name)]['Volume_Over_BSA'] =
                                                    volume_over_bsa
    if volume_over_bsa >= 34:
      bsa_data_dict[str(image_name)]['Label'] = 'abnormal'
      label_dataset[str(image_name)] = 'abnormal'
    else:
      bsa_data_dict[str(image_name)]['Label'] = 'normal'
      label_dataset[str(image_name)] =  'normal'

  image_masked_dataset= create_masked_image(image_dataset,
                                                    mask_dataset)
  image_dataset = np.asarray(image_dataset, dtype=np.float32)
  mask_dataset = np.asarray(mask_dataset, dtype=np.float32)

  return image_dataset, image_masked_dataset, mask_dataset,
          label_dataset, bsa_data_dict

def print_size_shapes(images, masks, masked_images, labels, images_name,
masks_name, masked_images_name, labels_name ):
  # Dataset
```

```
    print(f'{images_name} Dataset size : {len(images)}')
    print(f'{images_name} shape: {images.shape}')
    print(f'Single {images_name} shape: {images[1].shape}')
    print(f'{masks_name} shape: {masks.shape}')
    print(f'Single {masks_name} shape: {masks[1].shape}')
    print(f'{masked_images_name} shape: {masked_images.shape}')
    print(f'Single {masked_images_name} shape: {masked_images[1].shape}')
    print(f'{labels_name} shape: {np.asarray(list(labels.values())).shape}')

def plot_images(images,masks, masked_images, labels, lines=20):

    for i in range(lines):
        image = random.randint(0, images.shape[0])
        plt.figure(figsize=(16, 12))
        plt.subplot(131)
        plt.title(f'Original Image - {list(labels.values())[image]}')
        plt.imshow(images[image])
        plt.subplot(132)
        plt.title('Mask')
        plt.imshow(masks[image])
        plt.subplot(133)
        plt.title('Masked image')
        plt.imshow(masked_images[image])
        plt.show()

# DISPLAY STATISTICS
# Insert Data
image_directory = 'D:/A4C/CNN/EXTENDED/images_full_info_extented'
mask_directory = 'D:/A4C/CNN/EXTENDED/masks_extended_full_info'
```

```
bsa_directory = 'D:/A4C/CNN/EXTENDED/bsa_data_full_info_extended.csv

# Dataset BSA calculation
# BSA SETUP
dataframe = upload_data(bsa_directory,
['image','height','weight','long_axis','area','volume','birth','gender'])

dataframe.head()

dataframe.tail()

# Body Mass Area (BSA) calculation per image
data_dict = create_bsa_data_dict(dataframe)

data_dict['57']

# BSA calculation
bsa_coef_dict = bsa_calculation(data_dict)
bsa_coef_dict['57']

# Load Images-Masks
# Constants
IMG_WIDTH= 320
IMG_HEIGHT = 240

image_dataset, image_masked_dataset, mask_dataset, label_dataset,
data_dictionary= create_image_mask_label_sets(image_directory,
mask_directory, data_dict, IMG_WIDTH, IMG_HEIGHT)

data_dictionary['100']
```

```
# Shapes of uploaded original images, masks sets, labels
print_size_shapes(image_dataset, mask_dataset, image_masked_dataset, label_dataset, 'Original Images', 'Masks', 'Masked Images', 'Labels' )

# Plot images, masks,masked-images and respective ladels
plot_images(image_dataset, mask_dataset, image_masked_dataset, label_dataset, lines=10)

# Labels
labels = np.asarray(list(label_dataset.values()))
labels

# Statisitics Pressentation
# Population Gender
dataframe['Volume_Over_BSA'] = [data_dictionary[i]['Volume_Over_BSA'] for i in data_dictionary.keys()]
dataframe['Label'] = [data_dictionary[i]['Label'] for i in data_dictionary.keys()]

gender_plot =
dataframe['gender'].value_counts().plot(kind='pie',autopct='%1.1f%%', figsize=(8,8), fontsize=15, colors=['lightblue','pink'])
gender_plot.set_title('Populatation Gender Distribution', size=20)

# Population Age Range
current_yeara = datetime.today().year
younger = current_year - dataframe['birth'].max()
younger
```

```
oldest = current_year - dataframe['birth'].min()
oldest

# Average Height, Weight and Volume Over BSA
dataframe['height'].mean()

dataframe['weight'].mean()

dataframe['Volume_Over_BSA'].mean()

# Dataset Balance
print(Counter(labels))

labels_count_dict = dict(Counter(labels))
labels_count_dict

labels_classes = list(labels_count_dict.keys())
labels_classes_counts = list(labels_count_dict.values())

plt.figure(figsize = (8,8))
plt.pie(labels_classes_counts, labels=labels_classes, autopct='%1.1f%%', textprops={"fontsize":15}, colors=['red', 'lightgreen'])
plt.title('Cardiac Left Atria Enlargement \nClassification Distribution\n', size=20)
plt.axis('equal')
plt.show()
```

```python
# Dataset balance per Gender
dataframe['Volume_Over_BSA'] = [data_dictionary[i]['Volume_Over_BSA'] for i in data_dictionary.keys()]

dataframe['Label'] = [data_dictionary[i]['Label'] for i in data_dictionary.keys()]

dataframe

# Atria enlargment on male gender
male_data = dataframe[dataframe['gender'] == 'Male' ]

male_data

younger_male = current_year - male_data['birth'].max()
younger_male

older_male = current_year - male_data['birth'].min()
older_male

male_data['height'].mean()

male_data['weight'].mean()

male_data['Volume_Over_BSA'].mean()

dict(Counter(male_data['Label']))

labels_classes = list(dict(Counter(male_data['Label'])).keys())
labels_classes_counts = list(dict(Counter(male_data['Label'])).values())
```

```
plt.figure(figsize = (8,8))
plt.pie(labels_classes_counts, labels=labels_classes, autopct='%1.1f%%', textprops={"fontsize":15}, colors=['red', 'lightgreen'])
plt.title('Cardiac Left Atria Enlargement\nClassification Distribution \nof Male Population\n', size=20)
plt.axis('equal')
plt.show()

# Atria enlargment on female gender
female_data = dataframe[dataframe['gender'] == 'Female' ]
female_data

younger_female = current_year - female_data['birth'].max()
younger_female

older_female = current_year - female_data['birth'].min()
older_female

female_data['height'].mean()

female_data['weight'].mean()

female_data['Volume_Over_BSA'].mean()

dict(Counter(female_data['Label']))

labels_classes = list(dict(Counter(female_data['Label'])).keys())
labels_classes_counts = list(dict(Counter(female_data['Label'])).values())

plt.figure(figsize = (8,8))
```

```
plt.pie(labels_classes_counts, labels=labels_classes, autopct='%1.1f%%', textprops={"fontsize":15}, colors=['red', 'lightgreen'])
plt.title('Cardiac Left Atria Enlargement\nClassification Distribution\n of Female Population\n', size=20)
plt.axis('equal')
plt.show()
```