Implementación de Redes Generativas Adversarias (GANs) para la generación de imágenes de tejido humano.

Pedro Juan Segura Cabrera
Máster Universitario en Bioinformática y Bioestadística
Estadística y Bioinformática 5

Ferran Reverter Comes
Alexandre Sánchez Pla
02/01/2019

FICHA TRABAJO FINAL

Título:	*Implementation of a Generative Adversarial Network (GAN) for the generation of images of human tissue.*
Name of author:	*Pedro Juan Segura Cabrera*
Nombre del consultor/a:	Ferran Reverter Comes
Nombre del PRA:	Alexandre Sánchez Pla
Fecha de entrega (mm/aaaa):	01/2019
Titulación:	*Máster Universitario en Bioinformática y Bioestadística*
Área del Trabajo Final:	*Bioinformática y Bioestadística Área 5*
Idioma del trabajo:	*Inglés*
Palabras clave	*GAN, IA, Disciminator, Generator, MVP, CNN, Deep Learning*

Abstract:

Artificial intelligence (AI) is increasingly collaborating in medicine. Generative Adversarial Networks (GANs) constitute one of the most interesting algorithms or techniques of AI, concretely of Deep Learning, having great applications to medicine, such as the generation of medical images.

This project is based on the generation of images of human tissue, specifically, skin lesions such as melanoma or seborrheic keratosis, using a dataset of 2000 real images.

Despite the difficulties involved in GANs models in terms of definition, architecture, programming and training, the added value that they present justifies the above.

The main application of the project in the field of medicine is to increase the global database of images of human tissue and thus, contribute to medical studies in the dermatological area.

It has been possible to obtain images with a resolution of 64x64 and through a representation of reduced dimensions (PCA and tSNE) of the whole set of images (real and generated); the variability of the generated images has been visually validated against the real ones.

Index

List of figures

1. Introduction

1.1 Context and justification of the project

This project deals with the generation of images of human tissue through Generative Adversarial Networks (GANs), a well-known technique of Deep Learning. A GAN is based on a system of two artificial neural networks that compete with each other in a zero-sum game. It was introduced by Ian Goodfellow et al. in 2014 [1] and consists of a network that generates candidates (generator) based on a certain distribution of data and another evaluates them (discriminator) based on a real database. This iterative process of generation-discrimination is known as training and is carried out until the discriminating network does not know how to distinguish between real and generated data.
Some examples of GANs in medical and other fields are:

- Unsupervised anomaly detection with GANs to guide marker discovery [2]: the GAN learns a manifold of normal anatomical variability and combined with a anomaly scoring scheme helps identifying and labelling anomalies.
- Generating videos with scene dynamics [3]
- GAIN: missing data imputation using Generative Adversarial Networks [4]

This work has been chosen mainly due to the technical development involved; the generation of data through GANs is an interesting and novel tool that provides a wide range of possibilities. The practical application in the field of bioinformatics in general and the value that the project can provide was the main reason for choosing it.

The amount of images of human tissue is limited, there are as many as patient's skins have been photographed. There is an inherent linkage between the image and the physical patient. Artificial Intelligence breaks this linkage and provides a solution to increase the worldwide database of images of human tissue **without** having to have new patients.

Tangibly, what is intended to achieve with this work is to obtain validated images of human tissues good enough to be used for other purposes, i.e. contrast of hypothesis in skin cancer research.

1.2 Goals of the project

General goals:

- Design and implementation of a Generative Adversarial Network (GAN).
- The obtaining of images generated through the GAN.
- Guarantee a minimum level of quality of the generated images.

Specific goals:

- Research and read documentation related to the project.

- Establish a database of images as input to the network.
- Design, program and validate the system architecture.
- Generate and validate images.
- Write documentation about the project.

1.3 Approach and methodology

Several strategies have been evaluated to address the project, such as *Agile Management* or KANBAN and, finally, an iterative strategy based on *Lean Start up* has been chosen and consists of creating, measuring and learning. In the development phase, the focus has been on the creation of a viable minimum product (MVP) to work with and, later on, functionalities have been added to converge.
I decided to use existing frameworks and libraries that are part of the day to day of Machine Learning, adapting and modifying the necessary parts to develop the application successfully.

1.4 Planning

The main resources to carry out the project would be a PC and the environment for developing Deep Learning problems. Everything needed is provided by Anaconda[1] [5] environment.

For the planning, first the tasks were identified and grouped:

1. GANs research
2. Images BBDD download
3. Image processing
   3.1. Resolution adjustment
4. Design of system architecture
5. Input data
   5.1. Coding
   5.2. Testing
   5.3. Validation
6. GAN Generator
   6.1. Coding
   6.2. Testing
   6.3. Validation
7. GAN Discriminator
   7.1. Coding
   7.2. Testing
   7.3. Validation
8. GAN Model
   8.1. Coding
   8.2. Testing

[1] Anaconda is a Python data science platform that helps providing the dependencies needed centralizedly for creating Machine Learning.

8.3. Validation
9. GAN Training
10. Output data
10.1. Validate
11. Quality output
12. Optimize code
13. Write documentation
14. Design presentation
15. Present

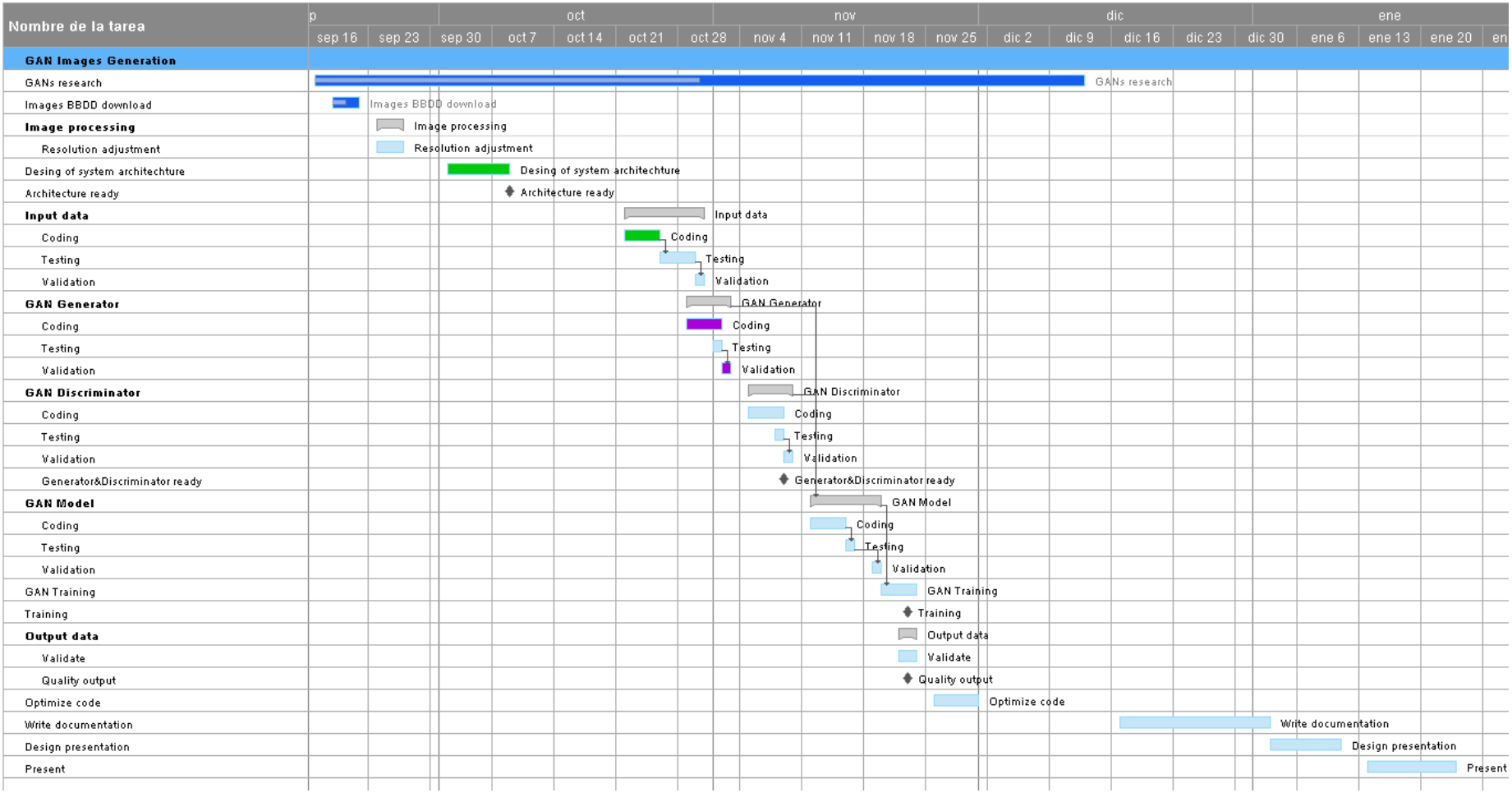

Figure 1. Gantt Diagram

Milestones
It has been established 4 milestones along the development of the project that will define its course:

- Definition of the system architecture: data I/O, GAN architecture...
  - For meeting this, both the data flow (input and output) and the network architecture (network type, number of layers and filters...) have to be defined. The project starts with a Convolutional Neural Net for the Generator and a Deconvolutional Neural Net for the discriminator (DCGAN).
- Program the generator and the discriminator:
  - Once they are defined and programmed, this milestone will be achieved.
- Training of the system:
  - Loss function defined
  - Algorithm and training variables of the general model defined:
    - Batch size
    - Number of epochs
    - Latent vector size
    - Etc.
- Obtaining a quality output

1.5 Brief summary of obtained products

As obtained products we have:
- Folder called *Code* that contains 4 programs in Python:
  - *model.py* defines the models of neural networks (generator and discriminator)
  - *GAN.py* implements the system for the generation of images
  - *PCA_representation.py* performs the PCA plotting for validate the generated images.
  - *tSNE_representation.py* performs the tSNE plotting for validate the generated images.
- Folder with tests and results
  - Inside there is a folder called *Validation*, there can be found the validation plots (PCA and tSNE) for each case.
- Folder with the dataset (ISIC_2017)
- The memory in PDF

1.6 Brief description of the other sections

The rest of the sections will explain the technical design and development deeper going into the details. The *Design* section is about what and how the things will be and the *Development* section explains how to get them done. After those chapters, the reader will find the *Tests and results* section where the experiments and their outputs are explained.

2. Theoretical Background

Artificial Intelligence (AI) is progressively getting more attention, researchers from all over the world are investigating in solutions and improvements to every-day problems applying AI, but what is AI?
Massachusetts Institute of Technology (MIT) defines it through a couple of concepts [6]:

- Computational models of human behaviour; programs that behave (externally) like humans
- Computational models of human "thought" processes; programs that operate (internally) the way humans do
- Computational systems that behave intelligently; i*ntelligently* meaning like humans
- Computational systems that behave rationally

So, Machine Learning (ML) is a tool for achieving AI. In 1959, Arthur Samuel, AI researcher defined machine learning informally as the ability to learn without being explicitly programmed. So Arthur Samuel, way back in the history of machine learning, actually did something very cool, which was he wrote a checkers program, which would play games of checkers against itself [7].

The way ML is performed depends on how the learning process is done, therefore we can distinguish among:
- **Supervised learning**: it is the task of learning a function that maps an input to an output based on example input-output pairs. It is performed over labelled data [8].
- **Unsupervised learning**: the learning is done from unlabelled, unclassified data. This type of learning tries to identify commonalities, patterns in the data and predict new data based on those patterns .
- **Reinforcement learning**: it is concerned with how software [9]. agents ought to take actions in an *environment* so as to maximize some notion of cumulative *reward* [10].

So, depending on which learning process we have in our problem, it will be address using one algorithm or another.
In the Supervised Learning, there most widely used are:
- Support Vector Machines
- Linear Regression
- Logistic Regression
- K-Nearest Neighbour
- Decision Trees
- Artificial Neural Networks

On the other hand, in the Unsupervised Learning:
- K-means
- Hierarchical Clustering

- Artificial Neural Networks

The algorithm[2] Artificial *Neural Networks* (ANN) is in both types of learning, it belongs to a subset of Machine Learning called Deep Learning, as seen in the image below.

Deep Learning uses a cascade of multiple layers of nonlinear processing units for feature extraction and transformation. Each successive layer uses the output from the previous layer as its input [11].

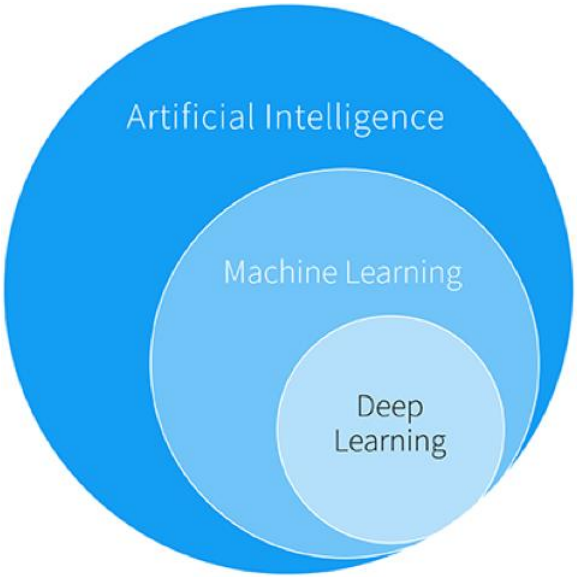

Figure 2. Artificial Intelligence breakdown

So, the Artificial Neural Networks are a computing systems inspired by the biological neural networks from animal brain. See Figure 3. They are based on a collection of connected units or nodes called artificial neurons disposed by layers. Each connection, like the synapses in a biological brain, can transmit a signal from one artificial neuron to another. An artificial neuron, receives a signal process it and then signal additional artificial neurons connected to it.

So, the signals are propagated forward according to the **activation function** until the last layer, the output layer, in which the loss function is computed.

The **loss function**, generally, calculates in the training process the difference between the obtained value from the ANN and the theoretical one. This value is called error. Then, this error is propagated backwards in order to update all of the weights (**backpropagation**) by means of the **gradient descent** algorithm.

Gradient descent is a first-order iterative optimization algorithm, mainly used for finding the minimum of a function. In ANN it performs gradient of the loss function in each neuron, therefore the weights are properly updated depending on the neuron.

[2] Artificial Neural Networks are not an algorithm, but rather a framework for many different machine learning algorithms to work together and process complex data inputs.

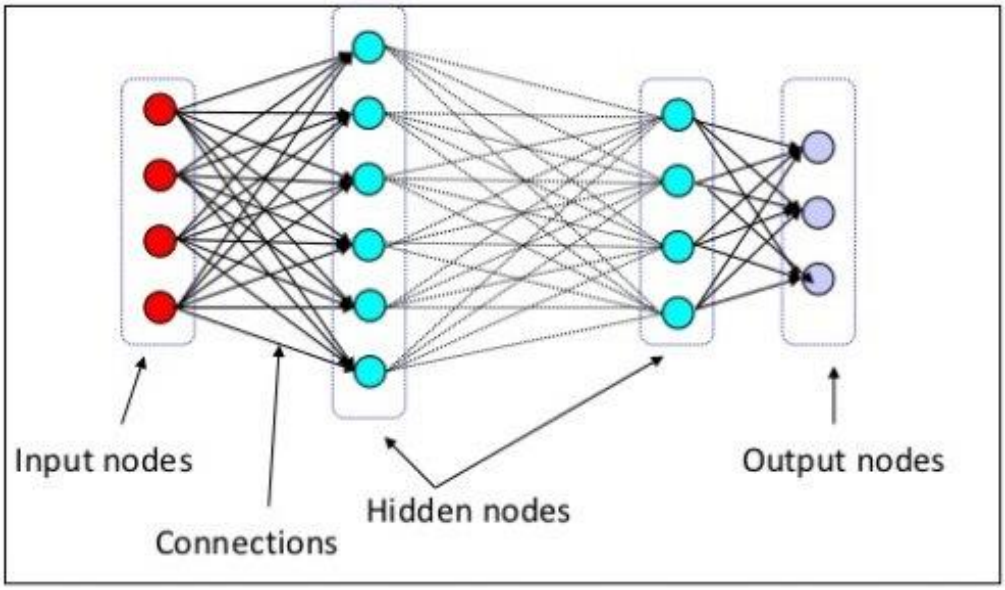

Figure 3. Artificial Neural Network

Convolutional Neural Networks (CNN) [12] are a type of ANN, used mainly for visual imagery. Typical CNN architecture is shown in the Figure 4.
It consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of convolutional layers, ReLU layer i.e. activation function, pooling layers, fully connected layers and normalization layers.
Description of the process as a convolution in neural networks is by convention. Mathematically it is a cross-correlation rather than a convolution (although cross-correlation is a related operation). This only has significance for the indices in the matrix, and thus which weights are placed at which index.

Convolutional layers apply a convolution operation to the input, passing the result to the next layer. The convolution emulates the response of an individual neuron to visual stimuli.
Each convolutional neuron processes data only for its receptive field. Although fully connected feed forward neural networks can be used to learn features as well as classify data, it is not practical to apply this architecture to images. A very high number of neurons would be necessary, even in a shallow (opposite of deep) architecture, due to the very large input sizes associated with images, where each pixel is a relevant variable. For instance, a fully connected layer for a (small) image of size 100 x 100 has 10000 weights for each neuron in the second layer. The convolution operation brings a solution to this problem as it reduces the number of free parameters, allowing the network to be deeper with fewer parameters. For instance, regardless of image size, tiling regions of size 5 x 5, each with the same shared weights, requires only 25 learnable parameters. In this way, it

resolves the vanishing or exploding gradients problem in training traditional multi-layer neural networks with many layers by using back propagation.

Pooling
Convolutional networks may include local or global pooling layers,[clarification needed] which combine the outputs of neuron clusters at one layer into a single neuron in the next layer. For example, max pooling uses the maximum value from each of a cluster of neurons at the prior layer.[12] Another example is average pooling, which uses the average value from each of a cluster of neurons at the prior layer.

Fully connected
Fully connected layers connect every neuron in one layer to every neuron in another layer. It is in principle the same as the traditional multi-layer perceptron neural network (MLP).

Receptive field
In neural networks, each neuron receives input from some number of locations in the previous layer. In a fully connected layer, each neuron receives input from every element of the previous layer. In a convolutional layer, neurons receive input from only a restricted subarea of the previous layer. Typically the subarea is of a square shape (e.g., size 5 by 5). The input area of a neuron is called its receptive field. So, in a fully connected layer, the receptive field is the entire previous layer. In a convolutional layer, the receptive area is smaller than the entire previous layer.

Weights
Each neuron in a neural network computes an output value by applying some function to the input values coming from the receptive field in the previous layer. The function that is applied to the input values is specified by a vector of weights and a bias (typically real numbers). Learning in a neural network progresses by making incremental adjustments to the biases and weights. The vector of weights and the bias are called a filter and represents some feature of the input (e.g., a particular shape). A distinguishing feature of CNNs is that many neurons share the same filter. This reduces memory footprint because a single bias and a single vector of weights is used across all receptive fields sharing that filter, rather than each receptive field having its own bias and vector of weights.

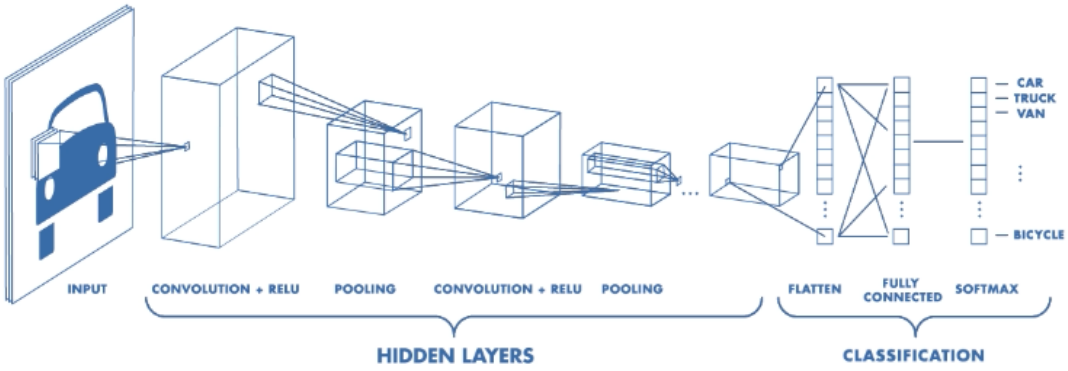

Figure 4. CNN architecture

CNN's lead us to the GANs that are a system of two neural networks contesting with each other in a zero-sum game framework. They were introduced by Ian Goodfellow et al. in 2014 [1].

Typically, the generative network learns to map from a latent space to a particular data distribution of interest, while the discriminative network discriminates between instances from the true data distribution and candidates produced by the generator. The generative network's training objective is to increase the error rate of the discriminative network (i.e., "fool" the discriminator network by producing novel synthesized instances that appear to have come from the true data distribution).

In practice, a known dataset serves as the initial training data for the discriminator. Training the discriminator involves presenting it with samples from the dataset, until it reaches some level of accuracy. Typically the generator is seeded with a randomized input that is sampled from a predefined latent space (e.g. a multivariate normal distribution). Thereafter, the discriminator evaluates samples synthesized by the generator. Back propagation is applied in both networks so that the generator produces better images, while the discriminator becomes more skilled at flagging synthetic images. The generator is typically a deconvolutional neural network, and the discriminator is a convolutional neural network [13].

The idea to infer models in a competitive setting (model versus discriminator) was proposed by Li and Gross in 2013 [14]. Their method is used for behavioural inference. It is termed Turing Learning, as the setting is akin to that of a Turing test. Turing Learning is a generalization of GANs. Models other than neural networks can be considered. Moreover, the discriminators are allowed to influence the processes from which the datasets are obtained, making them active interrogators as in the Turing test. The idea of adversarial training can also be found in earlier works, such as Schmidhuber in 1992. In the next image is shown the architecture of a GAN.

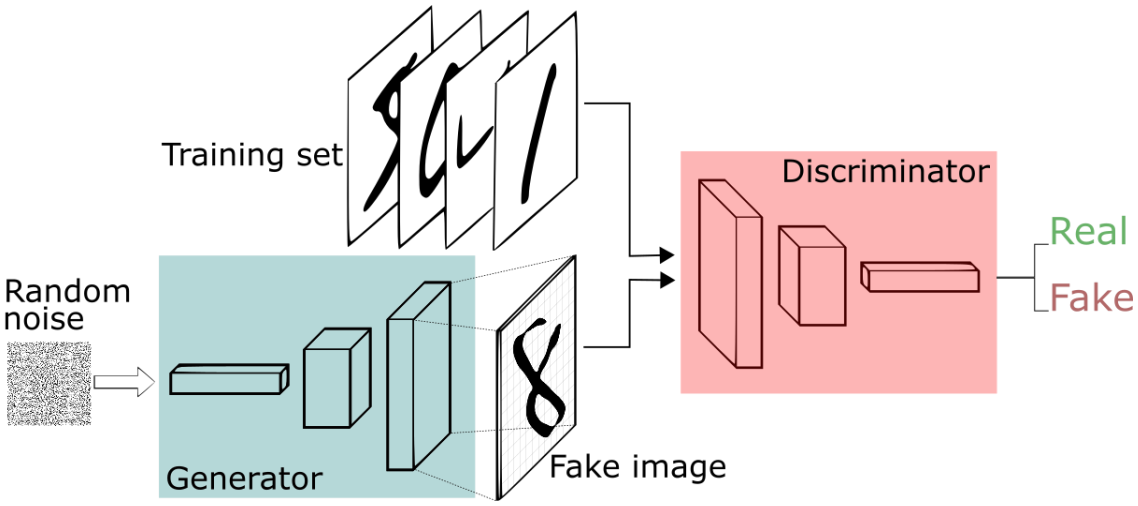

Figure 5. GAN architecture

3. Definition of requirements

Technical requirements
One of the main ones for the code is to be as modular and parameterized as possible. Thus, it is clean, organized, easy testable and easy changeable.

The code has to be able to perform the following actions:

- Set global variables as dataset path, number of epochs, batch size
- Load dataset of real images
- Image processing
  - Resizing
  - Normalization
  - Rolling axis
- Organize the images into batches of a specific size
- Define GAN model (architecture & parameters)
  - Define Generator neural network
  - Define Discriminator neural network
  - Combine them into a Generator containing Discriminator neural network
- Definition of hyperparameters:
  - Dimension of Z (latent space)
  - Which optimizer function to use? And which parameters to set? [14]
  - Loss function?
  - Intermodel margin
- Training of the complete model
  - Be able to load pre-trained models
  - Weights
  - Training and testing data
  - Definition of the labels
  - Definition of convergence
  - Save weights
- Generate images
- Validate generated images
  - tSNE
  - PCA

4. Design and methodology

There have been evaluated several work methodologies to be followed in this project, such as *Agile* Management or KANBAN and finally, it has been decided to go for an iterative strategy based on *Lean Start up*.
Lean Start up is a methodology created for developing businesses and products and consists in going through the whole process as fast as it is possible. In the next image, it can be seen the cyclic steps in a Lean Startup building process.

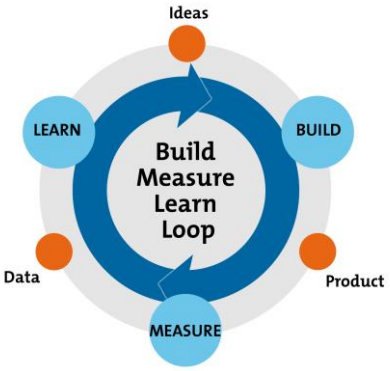

Figure 6. Lean Start up cycle

In this case, the first milestone to be reached was to develop a minimum viable product (MVP), which validates the whole end-to-end process. That means when the input/output data and the model were defined and developed and so, the model could be trained. This MVP development process involves many steps in the designing phase that we will see later on.

So, once the MVP is ready, it starts the testing phase where all the insights are gotten. The MVP consisted in the GAN with a dataset of only 10 real images in order to just generate images, even though they were only noise images with a pattern slightly similar to the real ones.

In the following paragraphs, the designing process is going to be explained, from the dataset of real images to the validation of the generated images.

DATASET COLLECTION
The dataset used [15] to evaluate the method corresponds to the one used in the ISIC 2017 challenge. It consists of 2000-coloured dermoscopic images of both benign and malignant skin lesions (images of 1372 benign lesions, 254 seborrheic keratosis samples and 374 melanoma). The resolution is not fix, therefore they were standardized to a fixed one in the code.

The dataset comes with a CSV file that classifies the images depending whether they are benign, seborrheic keratosis or melanoma following a **one hot encoding**. See image below.

image_id,melanoma,seborrheic_keratosis		
ISIC_0000000,0.0,0.0		
ISIC_0000001,0.0,0.0		
ISIC_0000002,1.0,0.0		
ISIC_0000003,0.0,0.0		
ISIC_0000004,1.0,0.0		
ISIC_0000006,0.0,0.0		
ISIC_0000007,0.0,0.0		
ISIC_0000008,0.0,0.0		
ISIC_0000009,0.0,0.0		
ISIC_0000010,0.0,0.0		

Figure 7. Real images classification

IMAGE PROCESSING

In order to have the images from the dataset as an input for the Discriminator network, they cannot be fed raw. They have to be processed and this involves resize and normalizes them.
Taking the dataset of images raw, it can be observed different resolutions like the 2 following images:

Figure 8. 2048 x 1536

Figure 9. 962 x 722

So, in order to develop quickly the MVP and due to Hardware awareness it was decided to resize all of the real images to **64 x 64** in order to be less time and resources consuming.

Another important aspect regarding the dataset is the image normalization. It modifies the range of pixel intensity values with the aim to achieve consistency in dynamic range for the dataset itself.
Initially, the images are in range [0,255], so the normalization done here is basically divide by the mean (127.5), now the range becomes [0,2] and then subtract by 1 in order to make it [-1,1] [16].

MODEL DEFINITION

The model is composed by 2-stacked Neural Networks, the Generator and the Discriminator. In this phase it was needed to define the architecture of the GAN, several types were initially evaluated, such as DCGAN, LAPGAN and Conditional GAN. It was decided to go for the DCGAN as a first approach, it works just fine with images.
So, remembering what a DCGAN looks like:

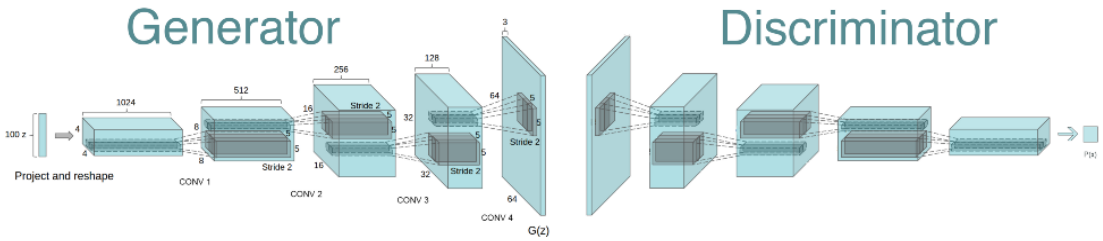

Figure 10. Deep Convolutional Generative Adversarial Network (DCGAN)

Here we have several variables to define, for building the Generator we have:

Input dimension of the Generator

The input of the Generator is a series of randomly generated numbers called latent sample that works as a noise vector (Z).

Output dimension of the Generator

The output of the Generator works as the input for the Discriminator, together with the dataset of real images. So, the dimension has to match the size of our images. As a first approach, 64 x 64.

Number of convolutions

This depends on how much information the images have, for example if the images that are trying to be generated are supposed to have many features, it is recommended to use several convolutions.
In our case, we use 4 Convolutional layers, specifying the number of output filters in each convolution, kernel size and number of strides. Taking into account that the desirable generated image size is 64x64 and the length of the noise vector is 100, therefore in order not to have superposition of the moving kernel the following parameters are to:

- kernel size = 5
- number of strides = 5

In the image below it is shown how it looks:

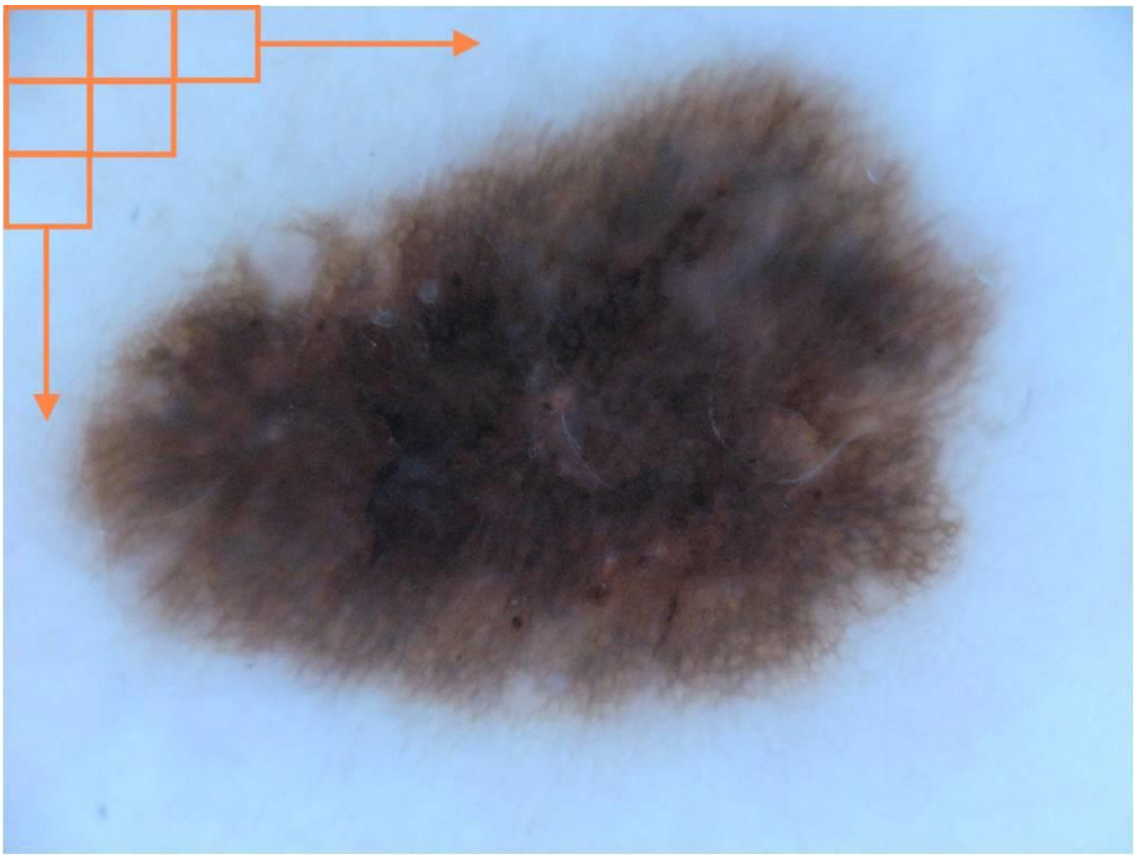

Figure 11. Convolution sliding filters

Let's assume this image has a size of 64x64, so the filters allocated in the upper left corner have a size of 5x5 and they move along the image with a stride of 5, that means there is no superposition, all the filter .
Padding is set to *same*, therefore it is assured the size of output feature map is the same as the input.

Batch normalization
In order to reduce the amount by what the hidden unit values shift around, so called covariance shift, we apply Batch Normalization after each layer in the Generator [17].

Activation layer
Also, after each conv layer, it is convention to apply a nonlinear layer (or **activation layer**) immediately afterward. The purpose of this layer is to introduce nonlinearity to a system that basically has just been computing linear operations during the convolutional layers (just element wise multiplications and summations). In the past, nonlinear functions like *tanh*

and *sigmoid* were used, but researchers found out that **ReLU layers** work far better because the network is able to train a lot faster (because of the computational efficiency) without making a significant difference to the accuracy. It also helps to alleviate the vanishing gradient problem, which is the issue where the lower layers of the network train very slowly because the gradient decreases exponentially through the layers.
ReLU layer applies the function f(x) = max(0, x) to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to 0. It increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the convolutional layer [18].

Upsampling
So, the idea behind the upsampling process is to reconstruct the continuous signal from the original one and resample it using more samples. We use a 2D upsampling layer before any convolutional layer in the generator in order to increase the sampling rate of the input data.

Output layer
The output layer is a tanh activation function, which helps mapping the resulting values into the [-1, 1] range, which is the same range as our processed real images. The output from the generator and the real images will be fed into the discriminator for the training process.

In the case of the Discriminator, regarding its structure we have:

Number of convolutions
It will have 4 convolutional hidden layers like the Generator. The first one will receive the 64x 64 sized images as input. In every convolutional layer, subsampling is performed in order to combat unbalanced classes. This is a common practice in classification problems solved by CNNs and that is what we are doing here, trying to classify what images are good enough to be considered “real”.

Leaky-ReLU activation layer
Every convolutional layer is followed by a Leaky-ReLU activation function. Many activation functions will work fine with this basic GAN architecture. However, leaky ReLUs are very popular because they help the gradients flow easier through the architecture.
A regular ReLU function works by truncating negative values to 0. This has the effect of blocking the gradients to flow through the network. Instead of the function being zero, leaky RELUs allow a small negative value to pass through. That is, the function computes the greatest value between the features and a small factor.
Leaky RELUs represent an attempt to solve the *dying ReLU”* problem. This situation occurs when the neurons get stuck in a state in which RELU units always output 0s for all inputs. For these cases, the gradients are completely shut to flow back through the network. This is especially

important for GANs since the only way the generator has to learn is by receiving the gradients from the discriminator [19].
In our case we will go for a slope value of 0.2.

Dropout
Right after every LeakyReLU activation layer, a Dropout layer is needed for several reasons.
GANs are likely to get stuck, therefore it is recommended to add some randomness in the training process. We do this by adding dropout that basically shuts down several units in certain passes in order not the develop co-dependencies among each other.
Dropout also helps preventing overfitting.
The dropout function takes a parameter called rate that is a fraction of the input units to drop, 0.2 in this case.

Flatten
A Flatten layer is needed for converting the 2d image into vector representation.

Sigmoid activation layer
Finally, the discriminator needs to output probabilities. For that, we use the *Logistic Sigmoid* activation function on the final logits.
A sigmoid function is applied to the real-valued output to obtain a value in the open-range [0, 1].

Once the generator and the discriminator are designed, they will be combined in a model called *generator-containing-discriminator*. Basically this is a sequential model that puts the generator and right after the discriminator. For training purposes is easier to work with this rather than working with the 2 models separated.

5. Development

In this chapter, we are going through the code giving explanations to what is needed.

DEPENDENCIES

I used Keras [20] running with a Tensorflow backend for the Machine Learning (ML) part of the code, i.e. definition of the neural networks, training...

Keras is a high-level ML API that allows fast prototyping and experimentation. Keras seems to fit perfectly in the context of this work taking into account the time restriction and difficulty level.

Tensorflow [21] is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

I used Anaconda [5] for install dependencies and libraries, it is a data science platform.

So, the code is structured in 2 modules:

- *model.py;* Defines the Generator and Discriminator neural networks and also the model based on both of them combined.
- *GAN.py;* This is the main code. From here the module *model.py* is imported in order to instantiate the models. In this code happens from the dataset loading and processing to the training and images generation.

Let's go through the code starting from the *model.py*:

```
3  from keras.models import Sequential, Model
4  from keras.layers import Dense, Dropout
5  from keras.layers import Reshape
6  from keras.layers.core import Activation
7  from keras.layers.advanced_activations import LeakyReLU
8  from keras.layers.normalization import BatchNormalization
9  from keras.layers.convolutional import UpSampling2D
10 from keras.layers.convolutional import Convolution2D
11 from keras.layers import Input, LSTM, RepeatVector, Lambda
12 from keras.layers.core import Flatten
13 from keras.optimizers import Adam
14 from keras import backend as K
```

Figure 12. *model.py* imported libraries

In the Design section, it has been explained the structure of the generator and discriminator. So in order to develop the code, it is needed to import mostly all the functions and libraries from above. Once they all imported, they can be used straightly.

The Keras architecture basically starts defining a model, Sequential in this case, that stacks layers linearly. Then, layers like Convolutional, Dropout, LeakyReLU or Dense (fully connected) have to be also imported.

The following image corresponds to the generator model:

```
def generator_model(inputdim = 100, xdim = 8, ydim = 8):
    # xdim = 2, ydim = 2 results in prediction shape of (1, 3, 32, 32)
    # xdim = 4, ydim = 4 results in prediction shape of (1, 3, 64, 64)
    model = Sequential()
    model.add(Dense(input_dim=inputdim, output_dim=1024*xdim*ydim))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Reshape( (1024, xdim, ydim), input_shape=(inputdim,) ) )
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Convolution2D(512, 4, 4, border_mode='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Convolution2D(256, 4, 4, border_mode='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Convolution2D(128, 4, 4, border_mode='same'))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(UpSampling2D(size=(2, 2)))
    model.add(Convolution2D(3, 4, 4, border_mode='same'))
    model.add(Activation('tanh'))
    return model
```

Figure 13. Generator definition

It can be seen the procedure with 3 parameters that are specified in the header:

- *inputdim:* it refers to the dimension of the randomly generated noise vector that works as the input of the generator.

As we mentioned before, the value is 100 as per convention[3].

- *xdim, ydim;* their value depends on the size of the image. They are used to specify the output of the first fully connected layer of the generator.

Then, the discriminator is defined. See image below to see how it is developed:

[3] GANs are recent discovery in the Artificial Intelligence world [27], so a value of 100 in the latent vector is commonly used. Analysis of several values can be considered as a future linea of research, however it is out of this project's scope.

```
def discriminator_model():
    model = Sequential()
    model.add(Convolution2D(128, 4, 4, subsample=(2, 2), input_shape=(3, 64, 64), border_mode = 'same'))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.2))
    model.add(Convolution2D(256, 4, 4, subsample=(2, 2), border_mode = 'same'))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.2))
    model.add(Convolution2D(512, 4, 4, subsample=(2, 2), border_mode = 'same'))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.2))
    model.add(Convolution2D(1024, 4, 4, subsample=(2, 2), border_mode = 'same'))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(output_dim=1))
    model.add(Activation('sigmoid'))
    return model
```

Figure 14. Discriminator definition

As a highlight, it can be seen how easy the layers are stacked one following another once the basic architecture is clear.
The first design and implementation of a discriminator was done in the early research phase in order to gain some background knowledge. In that design a classifier of images of cats and dogs were developed with a quite good success rate to be a simple one. See Annex C for more info.

And the last procedure of this script *model.py* is the combination of generator and discriminator.
Setting the model not to be trained, it allows fixing the weights in order to fine-tune the mode for example layer-wise.

```
def generator_containing_discriminator(generator, discriminator):
    model = Sequential()
    model.add(generator)
    discriminator.trainable = False
    model.add(discriminator)
    #discriminator.trainable = False
    return model
```

Figure 15. Generator containing discriminator

Now the main script (GAN.py) will be shown, starting from the libraries to be imported:

```
import argparse
import cv2
import numpy as np
import glob
import os
import model
import scipy
import matplotlib.pyplot as plt
import pandas as pd
from keras.optimizers import Adam
from keras.optimizers import SGD, RMSprop
import os, struct
from array import array as pyarray
```

Figure 16. *GAN.py* imported libraries

The basic imported libraries are:

- **Numpy:** Python package for numeric and scientific computing [22].
- **cv2:** it is an image and video-processing library with bindings in C++, C, Python, and Java. OpenCV is used for all sorts of image and video

analysis like facial recognition and detection, license plate reading, photo editing… [23]

- **Scipy:** Python-based ecosystem of open-source software for mathematics, science, and engineering [24].
- **Argparse:** this module makes it easy to write user-friendly command-line interfaces. The program defines what arguments it requires, and *argparse* will figure out how to parse those out of *sys.argv*. The *argparse* module also automatically generates help and usage messages and issues errors when users give the program invalid arguments. [25]
- **OS:** this module provides a portable way of using operating system dependent functionality [26].
- **Matplotlib:** it is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

Also module *model.py* is imported in order to be able to use the neural nets previously defined.

So, I decided to organize the main code in functions that will be called when is needed. For example for loading images, there is this following code that loads, resizes and normalizes it:

```
17  def load_image(path):
18      ''' Resize image to 64x64 and shuffle axis to create 3 arrays (RGB) '''
19      img = cv2.imread(path, 1)
20      img = np.float32(cv2.resize(img, (128, 128))) / 127.5 - 1
21      img = np.rollaxis(img, 2, 0)
22      return img
```

Figure 17. Image loading and processing

Also, I created a function for creating the noise vector and for creating the chunks for the batch distribution later on:

```
62  def noise_image():
63      ''' Create noisy data that will be converted to an image
64          Note size = (total number, number in sublist, length of subsublist )
65      '''
66      zmb = np.random.uniform(-1, 1, 100)
67      #zmb = np.random.uniform(0, 1, 100)
68      return zmb
```

Figure 18. Noise vector

```
70  def chunks(l, n):
71      """Yield successive n-sized chunks from l."""
72      for i in range(0, len(l), n):
73          yield l[i:i+n]
```

Figure 19. Chunks for batches organization

In order to run the algorithm, it is mandatory to specify some parameters like:

- **path**: directory where the dataset of real images is located.
- **TYPE**: training or generate.
- **batch_size**: size of the batches of the dataset.
- **epochs**: how many times the entire dataset is passed forward and backward through the entire neural network.
- **img_num**: how many images to generate.

In the image below it can be seen the function that parses all this parameters needed to run the algorithm:

```
279  def get_args():
280      parser = argparse.ArgumentParser()
281      parser.add_argument("--path", type = str, default = "../Dataset/ISIC_2017/")
282      parser.add_argument("--TYPE", type = str, default = "generate")
283      parser.add_argument("--batch_size", type = int, default = 100)
284      parser.add_argument("--epochs", type = int, default = 20)
285      parser.add_argument("--img_num", type = int, default = 25)
286
287      args = parser.parse_args()
288      return args
289
290  if __name__ == "__main__":
291
292      args = get_args()
293
294      if args.TYPE == 'train':
295          train(path = args.path, batch_size = args.batch_size, EPOCHS = args.epochs)
296      elif args.TYPE == 'generate':
297          generate(img_num = args.img_num)
```

Figure 20. Get arguments

The training process is the most important part; it gives the network the ability to get better through every iteration (epoch).
It starts by setting a seed number for reproducibility of the whole process.
Then the real images from the dataset with JPG extension are loaded. They have to be in a certain path that is specified in the arguments form the *get_args* function.
The real images are organized in batches in order to be fed to the discriminator that way. Then the models are instantiated from the *model.py* script and the optimizer function is defined. In this case, I used Adam optimizer. It has shown good results in Conv. Nets even for models in production. Right after the models have to be compiled to use them. The loss chosen is the “binary_crossentropy”.

To set up a point where we can assure the GAN has converged, it is needed to define a variable that measures the difference between the Generator and the Discriminator errors. That variable is called **intermodel margin**. We set a value of 0.1 for it by means of trial and error.

When the training core loop begins, in the first epoch, the first thing that the algorithm does is looking for any trace from previous trainings, looking for the weights files. If it finds it, they are loaded and the training continues right

from that point on. This is an interesting and useful concept because it saves time and prevents the duplicated work; it makes the whole process more efficient.

```
def train(path, batch_size, EPOCHS):

    #reproducibility
    np.random.seed(123)

    fig = plt.figure()

    # Get image paths
    print("Loading paths..")
    paths = glob.glob(os.path.join(path, "*.jpg"))

    print("Got paths..")

    # Load images
    IMAGES = np.array([load_image(p) for p in paths])
    np.random.shuffle(IMAGES)

    # Organize the images into batches
    BATCHES = [b for b in chunks(IMAGES, batch_size)]

    # Instantiate the discriminator, the generator and the gen_containing_disc
    discriminator = model.discriminator_model()
    generator = model.generator_model()
    discriminator_on_generator = model.generator_containing_discriminator(generator, discriminator)

    # Define the optimizers
    adam_gen = Adam(lr = 0.00002, beta_1 = 0.0005, beta_2 = 0.999, epsilon = 1e-08)
    adam_dis = Adam(lr = 0.00002, beta_1 = 0.0005, beta_2 = 0.999, epsilon = 1e-08)

    # Compile the models
    generator.compile(loss = 'binary_crossentropy', optimizer = adam_gen)
    discriminator_on_generator.compile(loss = 'binary_crossentropy', optimizer = adam_gen)
    discriminator.trainable = True
    discriminator.compile(loss = 'binary_crossentropy', optimizer = adam_dis)

    print("Number of batches", len(BATCHES))
    print("Batch size is", batch_size)

    # Define intermodel margin
    inter_model_margin = 0.1

    # Training core loop
    for epoch in range(EPOCHS):
        print()
        print("Epoch", epoch)
        print()

        # Load weights on first try (i.e. if process failed previously and we are attempting
        # to recapture lost data)
        if epoch == 0:
            if os.path.exists('generator_weights') and os.path.exists('discriminator_weights'):
                print("Loading saved weights..")
                generator.load_weights('generator_weights')
                discriminator.load_weights('discriminator_weights')
                print("Finished loading")
            else:
                pass
```

Figure 21. Training process I

Now it starts the looping inside of the real images batches, it creates noise batches and generates images out of them. They are all saved for any troubleshooting or analysis later on.
So, the following steps are basically the definition of the Discriminator and Generator inputs and labels vectors.

- For Discriminator, the input is based on 2 vectors:
  - The generated images (fakes)
  - The images from the dataset (real)
- For Generator, the input is the noise vector.

Now the proper training is performed thanks to the *train_on_batch* Keras function.
For every batch and epoch it outputs a loss both for the generator and discriminator and with this losses we established the convergence condition:

- The absolute value of the difference has to be smaller than the intermodel margin.

So, what the convergence loop does is identify the model with smallest loss and keep training it until the convergence condition is reached.

```
        # Looping inside of a batch
        for index, image_batch in enumerate(BATCHES):
            print("Epoch", epoch, "Batch", index)

            # Create a batch of noise vectors
            Noise_batch = np.array([noise_image() for n in range(len(image_batch))])

            # Generate images out of the noise batch
            generated_images = generator.predict(Noise_batch)

            # Saving all the generated images
            for i, img in enumerate(generated_images):
                rolled = np.rollaxis(img, 0, 3)
                if os.path.exists('./results'):
                    cv2.imwrite('./results/' + 'Epoch_' + str(epoch) + '_' + str(i) + ".jpg", np.uint8(255 * 0.5 * (rolled + 1.0)))
                else:
                    os.mkdir('./results')
                    cv2.imwrite('./results/' + 'Epoch_' + str(epoch) + '_' + str(i) + ".jpg", np.uint8(255 * 0.5 * (rolled + 1.0)))

            # Defining input of the discriminator
            X_disc = np.concatenate((image_batch, generated_images))

            # Defining labels for the discriminator
            Y_disc = [1] * len(image_batch) + [0] * len(image_batch) # labels

            print("Training first discriminator...")

            # Discriminator training
            d_loss = discriminator.train_on_batch(X_disc, Y_disc)

            # Defining input of the generator
            X_gen = Noise_batch

            # Defining labels for the generator
            Y_gen = [1] * len(image_batch)

            print("Training first generator...")
            g_loss = discriminator_on_generator.train_on_batch(X_gen, Y_gen)

            print("Initial batch losses : ", "Generator loss", g_loss, "Discriminator loss", d_loss, "Total:", g_loss + d_loss)

            # Convergence conditions
            if g_loss < d_loss and abs(d_loss - g_loss) > inter_model_margin:
                while abs(d_loss - g_loss) > inter_model_margin:
                    print(abs(d_loss - g_loss))
                    print("Updating discriminator..")
                    d_loss = discriminator.train_on_batch(X_disc, Y_disc)
                    print("Generator loss", g_loss, "Discriminator loss", d_loss)
                    if d_loss < g_loss:
                        break
            elif d_loss < g_loss and abs(d_loss - g_loss) > inter_model_margin:
                while abs(d_loss - g_loss) > inter_model_margin:
                    print(abs(d_loss - g_loss))
                    print("Updating generator..")
                    g_loss = discriminator_on_generator.train_on_batch(X_gen, Y_gen)
                    print("Generator loss", g_loss, "Discriminator loss", d_loss)
                    if g_loss < d_loss:
                        break
            else:
                pass

            print("Final batch losses (after updates) : ", "Generator loss", g_loss, "Discriminator loss", d_loss, "Total:", g_loss + d_loss)
            print()
```

Figure 22. Training process II

At the end of every training on batch, the weights are updated and saved and some of the generated images are shown in a combined plot.

```
            # Saving weights
            print('Saving weights..')
            generator.save_weights('generator_weights', True)
            discriminator.save_weights('discriminator_weights', True)

        plt.clf()

        # Show a combined plot generated images for visual assessment
        for i, img in enumerate(generated_images[:5]):
            i = i + 1
            plt.subplot(3, 3, i)
            rolled = np.rollaxis(img, 0, 3)
            plt.imshow(rolled)
            plt.axis('off')
        fig.canvas.draw()
        plt.savefig('Epoch_' + str(epoch) + '.png')
```

Figure 23. Training process III

Once the GAN is trained, it can be straightly used to generate images by specifying in the arguments the parameter *generate* and the number of images to generate:

```
199  def generate(img_num):
200      '''
201          Generate new images based on trained model.
202      '''
203      generator = model.generator_model()
204      adam = Adam(lr = 0.00002, beta_1 = 0.0005, beta_2 = 0.999, epsilon = 1e-08)
205      generator.compile(loss = 'binary_crossentropy', optimizer = adam)
206      generator.load_weights('generator_weights')
207
208      noise = np.array([noise_image() for n in range(img_num)])
209
210      print('Generating images..')
211      generated_images = [np.rollaxis(img, 0, 3) for img in generator.predict(noise)]
212      for index, img in enumerate(generated_images):
213          cv2.imwrite("{}.jpg".format(index), np.uint8(255 * 0.5 * (img + 1.0)))
214      print(np.shape(generated_images[0]))
```

Figure 24. Generate images

6. Tests and results

For the testing, several variables have been taking into account:

- Number of epochs
- Batch size
- Resolution of images

The resolution of the images is the main limitation due to the image processing capacity of the machine. I am running this in a MacBook Pro (i5 2.7 GHz processor, RAM of 8 GB and Graphic card Intel Iris Graphics 6100); it is not a set up specific for Deep Learning computing. However reducing the resolution, it is feasible to carry out the processing, training and generation of images in a relatively normal duration.

Below is shown the experiments carried out with their parameter details and generated images in each case:

Test #1
Number of epochs = 10
Batch size = 100
Resolution of images = 64x64
Duration = 5 hours

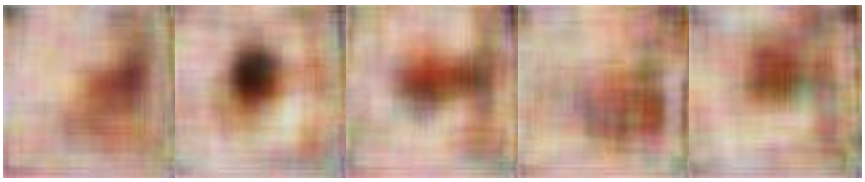

Figure 25. Test #1 - Generated images

Test #2
Number of epochs = 20
Batch size = 100
Resolution of images = 64x64
Duration = 11 hours

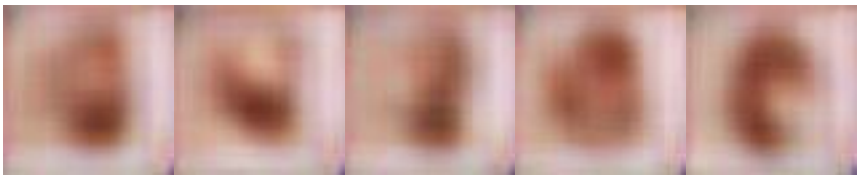

Figure 26. Test #2 - Generated images

Test #3
Number of epochs = 10
Batch size = 100
Resolution of images = 512x512
Duration = NA

Model did not converge; it remained stuck in the convergence condition.

Test #4
Number of epochs = 10
Batch size = 100
Resolution of images = 128x128
Duration = 38 hours

Figure 27. Test #4 - Generated images

As it can be seen, the poor resolution makes the images hard to be visually inspected or validated. That is why the validation was performed objectively by means of a reduced components representation (PCA and tSNE) of 1000 images from the dataset, for keeping the graph readable, including the generated images. In the graph, it is observed the distribution of the images along the axis and the clusters formed are noticeably logical. The reader can access to the code for both the PCA and the tSNE representation located in the Annex A and B, respectively.

So, Principal Components Analysis and t-Distributed Stochastic Neighbor Embedding have been chosen because both of them are techniques of dimensionality reduction and are well suited for this task. The main difference between them is that tSNE is non-linear so, it would be able to capture some trickier manifolds.

The following 2 pages are the results obtained by plotting the tSNE and the PCA for the Test #4:

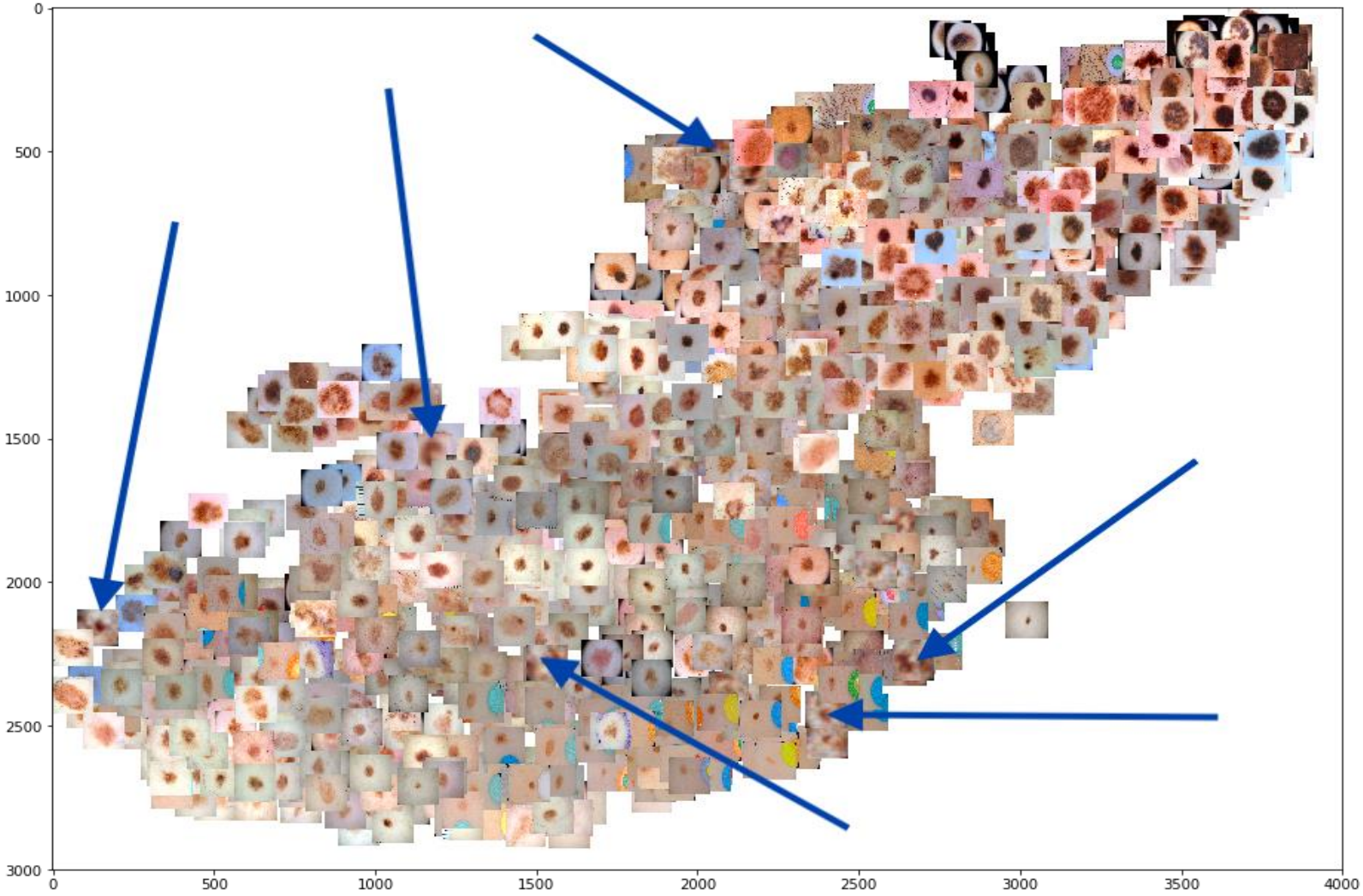

Figure 28. Test #4 - tSNE

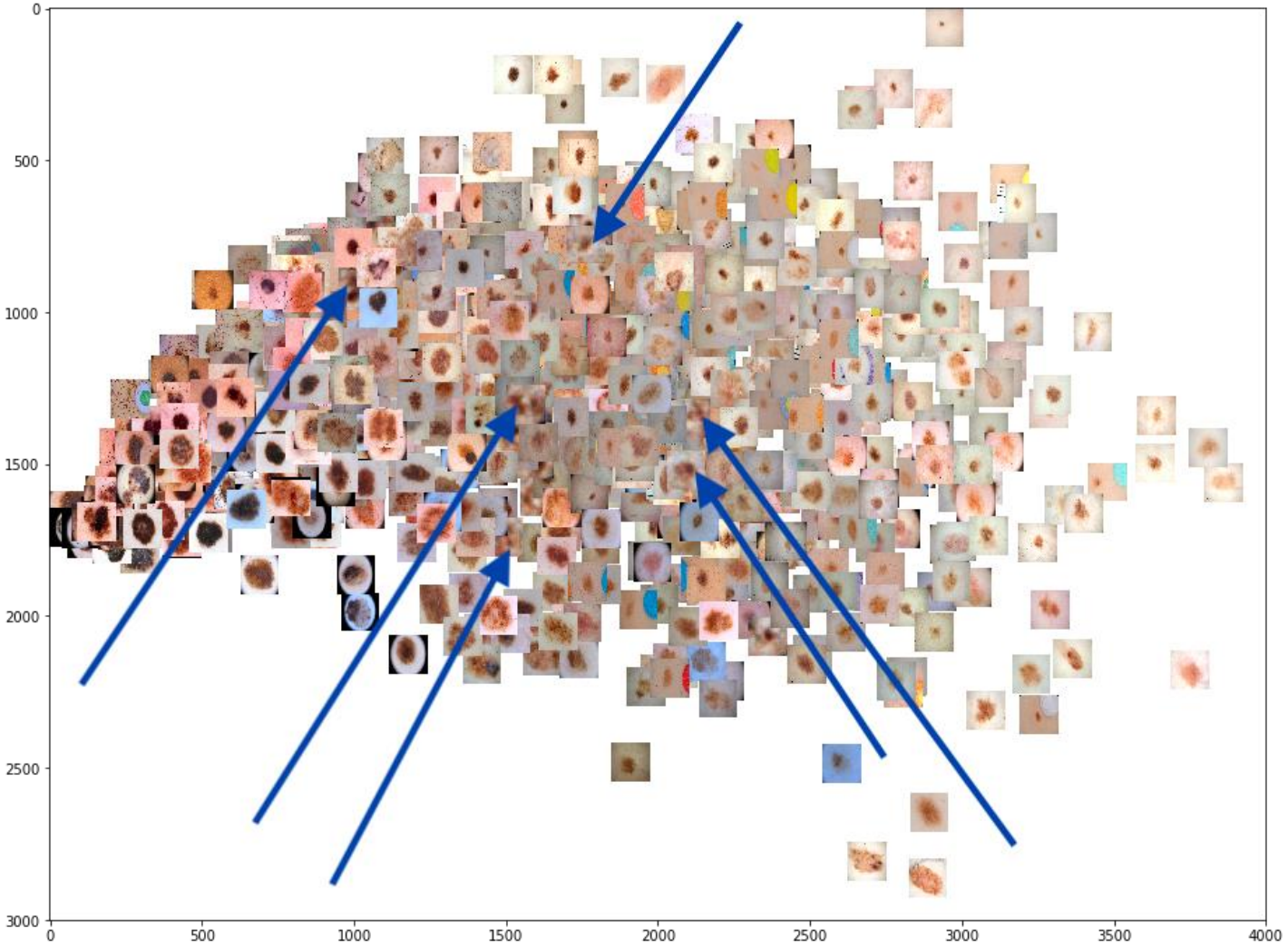

Figure 29. Test #4 - PCA

RESULTS

So, for the 2 analyses there have been added 25 generated images and the rest are real ones. Highlighted with a blue arrow are some of the fake images. It can be observed that the distribution of the fake images along the graph visually makes sense taking into account the clusters formed.
This means that the statistical variability of the generated images is similar to the variability of the dataset and could be added into the dataset for further studies as if they were real ones.
A failing scenario would be that in the graphical representation, the fake images form a clear cluster, meaning that they have their own characteristics. It is not what we are looking for with this work; instead we are trying to bring the fake images as close as we can to the real ones.

7. Conclusions and future lines of research

The general conclusions are:

- Even though the hardware limitation affecting the resolution of the output images, **the aim of the project has been reached successfully**. The technical complexity and the lack of rigorous information due to the newness of the topic have made the work tough sometimes. But it has been rewarding to meet the goals and satisfying to develop this work in the medicine field.
- Regarding the project management in terms of timing and resources, it has met the expectations. Due to the lack of seniorship in the field, I set a non-aggressive schedule with a big buffer at the end of the development phase in order to mitigate any timing deviation. It happened during the development of the training algorithm, I got stuck and spent more time than the one foresaw in the Gantt diagram. The buffer just worked fine for me in this case.

Future lines of research would be:

- Improving the algorithm to be able to distinguish among the skin lesions: Melanoma, Seborrheic keratosis... That would refine the model and make it more powerful. It can be done with a neural network as a classifier placed right after the discriminator of the GAN. As input to this classifier, the generated images those were able to “fool” the discriminator and the dataset for the training remains the same.
- Explore different GAN architectures, such as LAPGAN or Conditional GAN, in which the noise vector and a vector of labels form the input of the generator.
- With a more powerful computer set up, try to generate high-resolution images in order for them to be compared to the originals at the original resolution.

8. Glossary

AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
GAN	Generative Adversarial Network
DCGAN	Deep Convolutional Generative Adversarial Networks
MVP	Minimum Viable Product
MLP	Multi-Layer Perceptron
tSNE	t-Distributed Stochastic Neighbor Embedding
PCA	Principal Components Analysis
ReLU	Rectified Linear Unit
Tanh	Hyperbolic tangent
LAPGAN	Laplacian Generative Adversarial Network

9. Bibliography

[1] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio. (2014) Generative Adversarial Networks. [Online]. https://arxiv.org/pdf/1406.2661.pdf

[2] Thomas Schelgl, Philipp Seeböck, Sebastian M. Waldstein, Ursula Schmidt-Erfurth, and Georg Langs. (2017) Unsupervised Anomaly Detection with Generative Adversarial Networks to Guide Marker Discovery. [Online]. https://arxiv.org/pdf/1703.05921.pdf

[3] Carl Vondrick, Antonio Torralba, and Hamed Pirsiavash. (2016) Generating videos with scene dynamics.

[4] Jinsung Yoon, James Jordon, and Mihaela van der Schaar. (2018) GAIN: Missing Data Imputation using Generative Adversarial Nets.

[5] Anaconda. Anaconda. [Online]. https://www.anaconda.com

[6] Artificial Intelligence. MIT. [Online]. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-825-techniques-in-artificial-intelligence-sma-5504-fall-2002/lecture-notes/Lecture1Final.pdf

[7] Stanford University. https://see.stanford.edu/. [Online]. https://see.stanford.edu/materials/aimlcs229/transcripts/MachineLearning-Lecture01.pdf

[8] Supervised Learning. Wikipedia. [Online]. https://en.wikipedia.org/wiki/Supervised_learning

[9] Unsupervised Learning. Wikipedia. [Online]. https://en.wikipedia.org/wiki/Unsupervised_learning

[10] Reinforcement Learning. Wikipedia. [Online]. https://en.wikipedia.org/wiki/Reinforcement_learning

[11] Deep Learning. Wikipedia. [Online]. https://en.wikipedia.org/wiki/Deep_learning

[12] Convolutional NN. Wikipedia. [Online]. https://en.wikipedia.org/wiki/Convolutional_neural_network

[13] Wikipedia. https://en.wikipedia.org/wiki/Generative_adversarial_network.

[14] Gauci Li W and Gross R. M. (2013) A Coevolutionary Approach to Learn

Animal Behavior through Controlled Interaction.

[15] P. Kingma Diederik and Jimmy Lei Ba. (2015) Adam: a Method for Stochastic Optimization International Conference On Learning Representations, 2015. [Online]. https://arxiv.org/abs/1412.6980v8

[16] N.C., Gutman, D., Celebi, M.E., Helba, B., Marchetti, M.A., Dusza, S.W., Kalloo, A., Liopyris, K., Mishra, N., Kittler, H., et al. Codella. (2017) Skin lesion analysis toward melanoma detection: A challenge at the 2017 international symposium on biomedical imaging (isbi), hosted by the international skin imaging collaboration (isic). arXiv preprint arXiv:1710.05006.

[17] [Online]. https://eclass.teicrete.gr/modules/document/file.php/TP283/Lab/03.%20Lab/lesson3Notes.pdf

[18] Towards data science. [Online]. https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c

[19] Adit Deshpande. Adeshpande3. [Online]. https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

[20] Thalles Silva. sthalles.github.io. [Online]. https://sthalles.github.io/intro-to-gans/

[21] François Chollet and others. Keras. [Online]. https://keras.io

[22] G
oogle. Tensorflow. [Online]. https://www.tensorflow.org

[23] Numpy. Numpy. [Online]. http://www.numpy.org

[24] cv2. OpenCV. [Online]. https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_setup/py_intro/py_intro.html#intro

[25] SciPy. SciPy. [Online]. https://www.scipy.org/scipylib/index.html

[26] Argparse. Argparse. [Online]. https://docs.python.org/3/library/argparse.html

[27] OS. OS. [Online]. https://docs.python.org/3/library/os.html

[28] Alec Radford, Luke Metz, and Soumith Chintala. (2015) Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.

[29] Matplotlib. Matplotlib. [Online]. https://matplotlib.org

10. Annexes

A. Code for implementing representation in reduced dimensions (PCA)[4].

```
import os
#%matplotlib inline
import random
#import cPickle as pickle
import numpy as np
#from ggplot import *
import keras
from keras.preprocessing import image
from keras.applications.imagenet_utils import decode_predictions, preprocess_input
from keras.models import Model
import matplotlib.pyplot
from matplotlib.pyplot import imshow
from keras.applications.imagenet_utils import decode_predictions, preprocess_input
from keras import applications
from skimage.util.shape import view_as_windows
from skimage.transform import resize
from scipy.spatial import distance
from tqdm import tqdm
import pandas as pd
from sklearn.preprocessing import StandardScaler
from PIL import Image
from sklearn.decomposition import PCA


PATH = '../../../Dataset/ISIC_2017_split1_copy/'


model = keras.applications.Xception(weights='imagenet', include_top=True)
model.summary()

feat_extractor = Model(inputs=model.input, outputs=model.get_layer("avg_pool").output)
feat_extractor.summary()


def load_image(path):
    img = image.load_img(path, target_size=(64,64))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
```

[4] It has been implemented in Jupyter Notebook with its graphical functionalities.

```
    x = np.rollaxis(x, 1, 3)
    x = np.rollaxis(x, 3, 2)
    #print(x.shape)
    x = preprocess_input(x)
    return img, x


images_path = PATH
image_extensions = ['.jpg']   # case-insensitive (upper/lower doesn't matter)
max_num_images = 10000

images = [os.path.join(dp, f) for dp, dn, filenames in os.walk(images_path)
for f in filenames if os.path.splitext(f)[1].lower() in image_extensions]
if max_num_images < len(images):
    images = [images[i] for i in sorted(random.sample(xrange(len(images)),
max_num_images))]

print("keeping %d images to analyze" % len(images))


features = []
thumbs = []
for i, image_path in tqdm(enumerate(images)):
    if i % 10 == 0:
        print("analyzing image %d / %d" % (i, len(images)))
    img, x = load_image(image_path);

    feat = feat_extractor.predict(x)[0]
    features.append(feat)
    thumbs.append(img)

print('finished extracting features for %d images' % len(images))


features = pd.DataFrame(features)
scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)



pca = PCA(n_components=2)

_pca = pca.fit(features).transform(features)
#_pca    =    pca.fit(features_scaled).transform(features_scaled)       #scaled
version

pca_df=pd.DataFrame(_pca,columns=['pc1','pc2'])

tx, ty = pca_df.pc1, pca_df.pc2
tx = (tx-np.min(tx)) / (np.max(tx) - np.min(tx))
```

```
ty = (ty-np.min(ty)) / (np.max(ty) - np.min(ty))

width = 4000
height = 3000
max_dim = 200

full_image = Image.new('RGBA', (width, height))
for img, x, y ,thumb in tqdm(zip(images, tx, ty, thumbs)):
   tile = thumb
   rs = max(1, tile.width/max_dim, tile.height/max_dim)
   tile = tile.resize((int(tile.width/rs), int(tile.height/rs)), Image.ANTIALIAS)
   full_image.paste(tile, (int((width-max_dim)*x), int((height-max_dim)*y)),
                    mask=tile.convert('RGBA'))

matplotlib.pyplot.figure(figsize = (16,12))
imshow(full_image)
```

B. Code for implementing representation in reduced dimensions (tSNE)[5].

```
import os
#%matplotlib inline
import random
#import cPickle as pickle
import numpy as np
#from ggplot import *
import keras
from keras.preprocessing import image
from keras.applications.imagenet_utils import decode_predictions,
preprocess_input
from keras.models import Model
import matplotlib.pyplot
from matplotlib.pyplot import imshow
from keras.applications.imagenet_utils import decode_predictions,
preprocess_input
from keras import applications
from skimage.util.shape import view_as_windows
from skimage.transform import resize
from scipy.spatial import distance
from tqdm import tqdm
import pandas as pd
from sklearn.preprocessing import StandardScaler
from PIL import Image
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
```

[5] It has been implemented in Jupyter Notebook with its graphical functionalities.

```
PATH = '../../../Dataset/ISIC_2017_split1_copy/'


model = keras.applications.Xception(weights='imagenet', include_top=True)
model.summary()

feat_extractor = Model(inputs=model.input,
outputs=model.get_layer("avg_pool").output)
feat_extractor.summary()


def load_image(path):
    img = image.load_img(path, target_size=(64,64))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = np.rollaxis(x, 1, 3)
    x = np.rollaxis(x, 3, 2)
    #print(x.shape)
    x = preprocess_input(x)
    return img, x


images_path = PATH
image_extensions = ['.jpg']   # case-insensitive (upper/lower doesn't matter)
max_num_images = 10000

images = [os.path.join(dp, f) for dp, dn, filenames in os.walk(images_path)
for f in filenames if os.path.splitext(f)[1].lower() in image_extensions]
if max_num_images < len(images):
    images = [images[i] for i in sorted(random.sample(xrange(len(images)),
max_num_images))]

print("keeping %d images to analyze" % len(images))


features = []
thumbs = []
for i, image_path in tqdm(enumerate(images)):
    if i % 10 == 0:
        print("analyzing image %d / %d" % (i, len(images)))
    img, x = load_image(image_path);

    feat = feat_extractor.predict(x)[0]
    features.append(feat)
    thumbs.append(img)

print('finished extracting features for %d images' % len(images))
```

```
features = pd.DataFrame(features)
scaler = StandardScaler()
features_scaled = scaler.fit_transform(features)

print('A')

pca = PCA(n_components=200)
_pca = pca.fit(features).transform(features)

#np.random.seed(123)

X = np.array(_pca)
tsne = TSNE(n_components=2, learning_rate=150, perplexity=55,
angle=0.2, verbose=2).fit_transform(X)

tx, ty = tsne[:,0], tsne[:,1]
tx = (tx-np.min(tx)) / (np.max(tx) - np.min(tx))
ty = (ty-np.min(ty)) / (np.max(ty) - np.min(ty))

width = 4000
height = 3000
max_dim = 200

full_image = Image.new('RGBA', (width, height))
for img, x, y ,thumb in tqdm(zip(images, tx, ty, thumbs)):
    tile = thumb
    rs = max(1, tile.width/max_dim, tile.height/max_dim)
    tile = tile.resize((int(tile.width/rs), int(tile.height/rs)), Image.ANTIALIAS)
    full_image.paste(tile, (int((width-max_dim)*x), int((height-max_dim)*y)),
mask=tile.convert('RGBA'))

print('B')
matplotlib.pyplot.figure(figsize = (16,12))
imshow(full_image)

print('C')

print(type(full_image))
img_gen = []
for i, img in tqdm(enumerate(images)):
    img = image_path.split('.')[-2]
    img = img.split('/')[-1]
    if img == 'generated1':
        #print(i)
        img_gen.append(img)
    full_image.tell(img_gen)
```

C. Code for implementing classifier dogsVScats

```
import cv2                # working with, mainly resizing, images
import numpy as np        # dealing with arrays
import os                 # dealing with directories
from random import shuffle # mixing up or currently ordered data that might lead our network astray in training.
from tqdm import tqdm     # a nice pretty percentage bar for tasks. Thanks to viewer Daniel BA1/4hler for this suggestion

TRAIN_DIR = './train'
TEST_DIR = './test'
IMG_SIZE = 64
LR = 1e-3

MODEL_NAME = 'dogsvscats-{}-{}.model'.format(LR, '2conv-basic') # just so we remember which saved model is which, sizes must match

def label_img(img):
    if img == '.DS_Store':
            pass
    else:
        word_label = img.split('.')[-3]
        if word_label == 'cat': return [1, 0]
        elif word_label == 'dog': return [0, 1]


def create_train_data():
    training_data = []
    for img in tqdm(os.listdir(TRAIN_DIR)):
        label = label_img(img)
        path = os.path.join(TRAIN_DIR,img)
        if img == '.DS_Store':
            pass
        else:
            img = cv2.imread(path,cv2.IMREAD_GRAYSCALE)
            img = cv2.resize(img, (IMG_SIZE,IMG_SIZE))
            training_data.append([np.array(img),np.array(label)])
    shuffle(training_data)
    np.save('train_data.npy', training_data)
    return training_data


def create_train_data():
    training_data = []
    for img in tqdm(os.listdir(TRAIN_DIR)):
        label = label_img(img)
        path = os.path.join(TRAIN_DIR,img)
        if img == '.DS_Store':
            pass
```

```
            else:
                img = cv2.imread(path,cv2.IMREAD_GRAYSCALE)
                img = cv2.resize(img, (IMG_SIZE,IMG_SIZE))
                training_data.append([np.array(img),np.array(label)])
        shuffle(training_data)
        np.save('train_data.npy', training_data)
        return training_data


    #Now, we can run the training:
    train_data = create_train_data()
    # If you have already created the dataset:
    #train_data = np.load('train_data.npy')


    import tflearn
    from tflearn.layers.conv import conv_2d, max_pool_2d
    from tflearn.layers.core import input_data, dropout, fully_connected
    from tflearn.layers.estimator import regression

    convnet = input_data(shape=[None, IMG_SIZE, IMG_SIZE, 1],
name='input')

    convnet = conv_2d(convnet, 32, 5, activation='relu')
    convnet = max_pool_2d(convnet, 5)

    convnet = conv_2d(convnet, 64, 5, activation='relu')
    convnet = max_pool_2d(convnet, 5)

    convnet = conv_2d(convnet, 128, 5, activation='relu')
    convnet = max_pool_2d(convnet, 5)

    convnet = conv_2d(convnet, 64, 5, activation='relu')
    convnet = max_pool_2d(convnet, 5)

    convnet = conv_2d(convnet, 32, 5, activation='relu')
    convnet = max_pool_2d(convnet, 5)

    convnet = fully_connected(convnet, 1024, activation='relu')
    convnet = dropout(convnet, 0.8)

    convnet = fully_connected(convnet, 2, activation='softmax')
    convnet = regression(convnet, optimizer='adam', learning_rate=LR,
loss='categorical_crossentropy', name='targets')

    model = tflearn.DNN(convnet, tensorboard_dir='log')


    if os.path.exists('{}.meta'.format(MODEL_NAME)):
        model.load(MODEL_NAME)
```

```
    print('model loaded!')


train = train_data[:-500]
test = train_data[-500:]


X = np.array([i[0] for i in train]).reshape(-1,IMG_SIZE,IMG_SIZE,1)
Y = [i[1] for i in train]

test_x = np.array([i[0] for i in test]).reshape(-1,IMG_SIZE,IMG_SIZE,1)
test_y = [i[1] for i in test]


model.fit({'input': X}, {'targets': Y}, n_epoch=3, validation_set=({'input':
test_x}, {'targets': test_y}),
    snapshot_step=500, show_metric = True, run_id=MODEL_NAME)

model.save(MODEL_NAME)


import matplotlib.pyplot as plt

# if you need to create the data:
#test_data = process_test_data()
# if you already have some saved:
test_data = np.load('test_data.npy')

fig=plt.figure()

shuffle(test_data)

for num,data in enumerate(test_data[24:36]):
    # cat: [1, 0]
    # dog: [0, 1]

    img_num = data[1]
    img_data = data[0]
    print(img_num)
    #print(img_data)

    y = fig.add_subplot(3, 4, num + 1)
    orig = img_data
    data = img_data.reshape(IMG_SIZE,IMG_SIZE,1)
    model_out = model.predict([data])[0]

    print(model_out)

    if np.argmax(model_out) == 1: str_label='Dog'
    else: str_label='Cat'
```

```
    y.imshow(orig,cmap='gray')
    plt.title(str_label)
    y.axes.get_xaxis().set_visible(False)
    y.axes.get_yaxis().set_visible(False)
plt.show()
```

PREDICTIONS: