Deepfake Detection Using Deep Neural Networks

By

Oussama Chahboune

120225721

Supervisor: Dr. Andreas Amann

MSc in Mathematical Modelling and Self-Learning Systems

UNIVERSITY COLLEGE CORK

COLÁISTE NA HOLLSCOILE CORCAIGH

Department of Applied Mathematics

Head of Department: Prof. Sebastian Wieczorek

A MSc dissertation submitted to University College Cork in accordance with the requirements of the degree of MASTER OF SCIENCE IN MATHEMATICAL MODELLING AND SELF-LEARNING SYSTEMS in the School of Mathematical Sciences.

June 2021 – October 2021

Declaration of own work

This is to certify that the work I am submitting is my own and has not been submitted for another degree, either at University College Cork or elsewhere. All external references and sources are clearly acknowledged and identified within the contents. I have read and understood the regulations of University College Cork concerning plagiarism and intellectual property.

Oussama Chahboune

Acknowledgement

First and foremost, I would like to thank my supervisor Dr Andreas Amann who helped me, advised me and supported me during the whole project duration. I have learnt a lot while working with him and I am really grateful for it. I would also like to thank my family for encouraging me and providing me with a pleasant environment allowing me to both work on this project and rest when I needed it. Finally, I would like to thank my friends who shared their experience, their joy, and their struggles with me.

Abstract

Deepfakes are getting easier to use and getting better at imitating reality. To counter the dangers of this new technology, researchers are developing more and more accurate deepfake detection algorithm, often using deep neural networks. In this report, we tackled the idea of using the structural similarity index to detect deepfakes through flaws appearing between two consecutive frames. We also fixed the `scikit-image` implementation of this measure. Then we developed a pipeline to detect and crop the faces in given videos. Finally, we developed a deepfake detection algorithm which uses these faces. We tested it with three different models of neural networks, observed, and discussed the results of these experiments.

The code discussed in this report is fully available publicly on **GitHub**.

Contents

Page

1 Introduction **9**

1.1 What Are Deepfakes? 9

1.2 The Dangers of Deepfakes 10

1.3 The Need to Detect Deepfakes 11

1.4 Aims and Objectives 11

2 Literature Review **13**

2.1 Related Work 13

2.2 Choosing a Model 14

2.3 EfficientNet 14

2.4 RCNN 17

2.5 MesoNet (Meso-4) 17

2.6 Datasets 18

3 Assessing the Difference Between Two Consecutive Frames **20**

3.1 Mean Squared Error 20

3.2 Structural Similarity Index 21

4 Deepfake Detection Using Neural Networks **28**

4.1 Face Detection Methods 28

4.2 Proposed Pipeline for Face Detection and Extraction 30

4.3 Inspiration and Issues 34

4.4 Used Dataset 35

4.5 Data Splitting 36

4.6 Face Cropping Using MTCCN 37

4.7 Designing the Model 39
4.8 Training the Model 44
4.9 Tests and Results 45

5 Conclusion **51**

A Appendix **52**
A.1 Structural Similarity Index 52
A.2 Face Detection and Cropping Prototypes 54
A.3 Data Splitting Before Training 59

References **66**

List of Figures

1.1 This is not Tom Cruise. 9
1.2 Miles Fisher (left) impersonating Tom Cruise using deepfake (right) 9
1.3 A deepfake of Vladimir Putin (screenshot of RepresentUs ad [3]) 11

2.1 Example of face crops and segmentations used by Matern *et al.* (Source: [6]) . . . 14
2.2 Different scaling methods (Source: [13]) 15
2.3 EfficientNet-B0 architecture (Source: [15]) 16
2.4 Structure of a MBConv layer (H, W, and C stand for Height, Width, and Channels resp.) (Source: [16]) 16
2.5 Architecture of a RCNN (Source: [18]) 17
2.6 Architecture of Meso-4 network (Source: [19]) 18

3.1 Images representing a blue ball, created with *Paint* 27
3.2 Snapshots of videos. Image (a) is from an authentic video and the three others (b), (c), and (d) are from the same deepfake video 27

4.1 The four different types of Haar features used in [31] and a their possible numerical values (Source: [32] and [33] resp.) 29
4.2 Example of Haar features applied to a face (Source: [31]) 29
4.3 Pipeline of MTCNN cascaded network proposed by Zhang *et al.* (Source: [34]) . . 30
4.4 Regions detected as faces by the Haar cascade face detection function 31
4.5 Examples of returned outputs taken from [30] 33
4.6 Regions detected as faces by the MTCNN face detection function 34
4.7 Output of `03-train_cnn.py` when running A. Chong and H. Ng's code 35
4.8 Snapshot of the low contrast video used 38

4.9 A cropped face without padding (left) and the same face (and same frame) with padding (right) - (Scales do not match) 39
4.10 EfficientNet-B0 architecture in Tensorflow 41
4.11 Meso-4 architecture in Tensorflow 42
4.12 RCNN architecture in Tensorflow 43

List of Tables

4.1 Results using the EfficientNet-B0 model 47
4.2 Results using the Meso-4 model 48
4.3 Results using the RCNN model 49

1 Introduction

1.1 What Are Deepfakes?

This is not Tom Cruise.

Figure 1.1: This is not Tom Cruise.

In reality, this person is Miles Fisher. He is impersonating the famous actor using a deepfake created by visual and AI effects artist Chris Umé.

Figure 1.2: Miles Fisher (left) impersonating Tom Cruise using deepfake (right)

The concept of *deepfake* usually refers to the technology used to impersonate people using deep learning. For instance, here, Miles Fisher replaced his face by Tom Cruise's to impersonate him.

The word *deepfake* was coined from *deep learning* and *fake*. Historically, the first appearance of this word matches with the first known deepfake videos. These videos were shared by a Reddit user named *Deepfakes* in 2017. They used a software developed by a software engineer using open-source resources developed by Google and Nvidia. *Deepfakes* swapped the faces of adult performers with the face of female Hollywood actresses or singers such as Scarlett Johansson or Taylor Swift. Although this type of content was quickly banned by Reddit, it also spread very quickly across the Internet and the technology gained a lot of attention from the media and research community.

At first, the technology required technical knowledge and skills in machine learning to be used, however, new easy to use software such as Faceswap or Deepfakesweb.com are being developed and released. This allows even amateur people to create their own deepfakes. This also means that this technology can be used in a harmful way, should it be used by the wrong people.

1.2 The Dangers of Deepfakes

A knife can be used to cut food, but it can also be used to kill. Similarly, although deepfakes can be used for multiple innocent purposes such as creating fictional media or replacing dead actors in movies, its use can also be harmful to people and society. According to [1], in 2019, 96% of deepfake videos available online were non-consensual pornography. Such use can have heavy consequences such as public shaming and blackmail.

In 2019, criminals used another kind of deepfake to impersonate the voice of a businessman's boss on the phone in order to scam him [2]. The scammers used deepfake to impersonate the voice, tone and accent of the CEO of the parent company in Germany and asked the businessman in the UK to transfer €220,000 to a bogus Hungarian company.

Deepfake can also be used to spread misinformation. For example, fake videos of politics saying compromising things can be posted online to influence elections or cause instability. The videos published in 2020 by the advocacy group RepresentUs [3] are great examples of this type of use. These videos are part of a campaign aimed at sensitizing the American public to the threat posed by deepfakes to Democracy. However, this is not the only threat, doubts generated by whether a video footage is a deepfake or authentic can also create trouble. In 2019 in Gabon, after several months

without any appearance of president Ali Bongo, a video of him was released for New Year's address [4]. The issue was that the president did not look like himself, he was presumed dead and doubts about the authenticity of the video spread. The opposition party even tried to prove the video was a deepfake. This caused an attempted coup by Gabon's elite Republican Guard. In fact, the video was authentic. It turned out Bongo had suffered a stroke and probably had Botox and heavy make up, which changed his appearance.

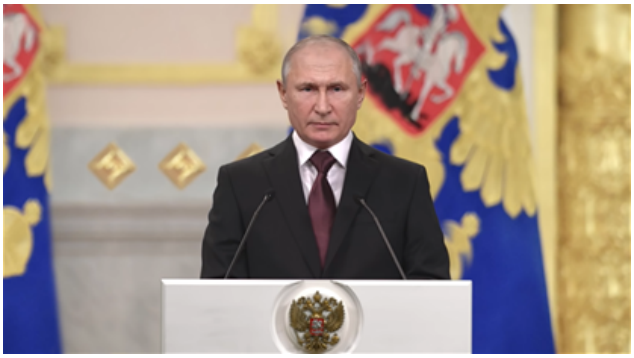


Figure 1.3: A deepfake of Vladimir Putin (screenshot of RepresentUs ad [3])

1.3 The Need to Detect Deepfakes

The dangers listed above show how big of an impact deepfakes can have on people and society. They also show the need for a response, solutions that can help tackle these issues. Apart from sensitization and education, there is a need to detect deepfakes and distinguish them from authentic videos.

As deepfakes are getting better at impersonating people and difficult to detect with human eyes, researchers are making good progress at tackling the issue, and even industry giants such as Facebook and Microsoft are investing in research and challenges aiming to improve deepfake detection software.

1.4 Aims and Objectives

In this report, we will try and understand how deefakes are detected using deep neural networks. Firstly, we will review interesting work that has been published in this field. Then, after describing

relevant neural network models and datasets, we will explain our research process, the objective of which is to produce a working proof of concept of a deepfake detection algorithm. To achieve it, we will talk about the structural similarity index and how it could be used for deepfake detection. Then, we will introduce a pipeline for face detection and extraction from videos. Finally, we will show how we trained the chosen deep neural networks and discuss the results that we got.

2 Literature Review

2.1 Related Work

There have been several attempts at finding a way to detect identity swap deepfakes. Researchers have leveraged face features such as eye blinking [5] or visual artifacts to prove such manipulations. The research results are as interesting as they are varied.

One very interesting, yet complex method was proposed by Li *et al.* [5], who use eye blinking data in videos to label them as fake or authentic. Indeed, more than 40 muscles compose the human face which make it a complex and difficult system to reproduce. A person blinks every two to ten seconds and one single blink lasts between 0.01 and 0.25 second, which make it very difficult for an algorithm to fake this physiological aspect. In many cases, deepfake creation methods fail to reproduce facial geometry and reflections. Matern *et al.* [6] exploit missing reflection details in eyes, and missing or imprecise geometry in eyes and teeth to expose manipulated videos. In Figure 2.1, we show face crops and segmentations resulting from the use of K-means clustering applied to the convex hulls of the regions of the eyes and teeth. Furthermore, deepfake techniques generate limited resolution images which are then distorted to match the dimensions of the original faces. This leaves visible artifacts, which can be easily detected using convolutional neural networks (CNNs) as demonstrated by Yi *et al.* [7]. Other visual components such as color disparities [8], or inconsistencies in lighting or shadowing [9] can also be used to detect deepfakes. Deepfakes also have flaws which can be detected using multimedia forensics such as the method proposed by Odena *et al.* [10] designed to detect artifacts created by deconvolution or the method to detect and localize resampling which make up forged images, proposed by Bunk *et al.* [11].

In general deepfake detection methods target a specific flaw of deepfakes. This raises the questions of whether considering the whole region of the face instead of a particular characteristic of the the video makes sense, whether it works and whether it is accurate enough.

Figure 2.1: Example of face crops and segmentations used by Matern *et al.* (Source: [6])

2.2 Choosing a Model

Supervised learning designates a machine learning category. A supervised learning algorithm is trained using labeled datasets (pairs of input object and correct output value) [12]. Using cross-validation, the model is fitted through the adjustment of the weights as data is fed to the algorithm. During the training a loss function is used to compute the accuracy, which improves as the algorithm adjusts and the error is minimized. Supervised learning is mainly used to solve two types of problems: classification and regression. Classification is used to classify data into given categories. During training the algorithm learns to recognize features and patterns and assign a category to the input. Regression is used for predictions (such as weather forcast) by finding the link between dependent and independent variables.

As the goal is to characterize a given video as being either a *deepfake* or *authentic*, deepfake detection can be seen as a classification problem. In the following sections we will deal with three different supervised learning models.

2.3 EfficientNet

EfficientNet was introduced by two Google researchers Mingxing Tan and Quoc V. Le in [13]. They claim that it is up to 10 times more efficient than "state-of-the-art" accuracy on several datasets including ImageNet and CIFAR-10. EfficientNet differs from other CNNs in the scaling method. Instead of focusing only on one dimension of the network such as width, depth, or resolution scaling, they use compound scaling which uniformly scales all three dimensions in the same process. In Figure 2.2, (a) is a base convolutional network used for comparison. The spatial dimension of this network is gradually decreased while the dimension of the channels is increased. The dimensions of network (b) are only scaled with respect to width (number of channels). For network (c)

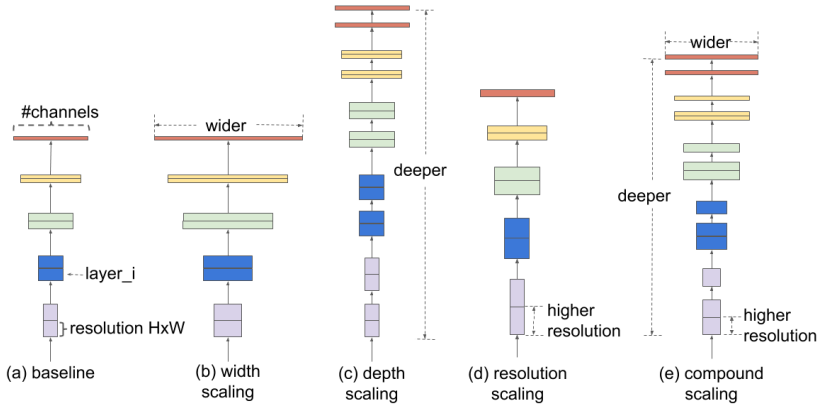


Figure 2.2: Different scaling methods (Source: [13])

it is the depth of the network (number of layers) which is increased, and for (d) layers have higher resolution. The scaling method of EfficientNet is similar to network (e) where all three dimensions are scaled up.

The architecture of the EfficientNet baseline model, EfficientNet-B0, is shown in Figure 2.3. Mingxing Tan and Quoc V. Le designed it using a Neural Architecture Search, which is a method used to automate the design of artificial neural networks. The EfficientNet-B0 is composed of *MBConv* (mobile inverted bottleneck convolution) layers. Here MBConv1 and MBConv6 have two different expansion factors [14]. MBConv6 has an expansion factor of 6, which means that when it gets c channels in input, it outputs $6c$ channels. MBConv1 has an expansion parameter of 1, so there is no expansion. There are 16 MBConv6 layers, 1 MBConv1 layer and 1 regular convolutional layer. The structure of a MBConv is shown in Figure 2.4. The output of a MBConv layer, is the sum of its input and the output of network composed of a 1×1 convolutional layer, 3×3 depth-wise convolution, Squeeze and Excitation (SE) layer, and a last 1×1 convolutional layer. Depth-wise convolutional consists of separating the channels of the image and the filter, perform convolution, then stack them back together.

This model was originally designed for image classification and it was proved by Tan and Le that it was highly accurate for this task. Therefore, it should also work with deepfake detection as it is a kind of image classification. That is why we chose to use this model for our project.

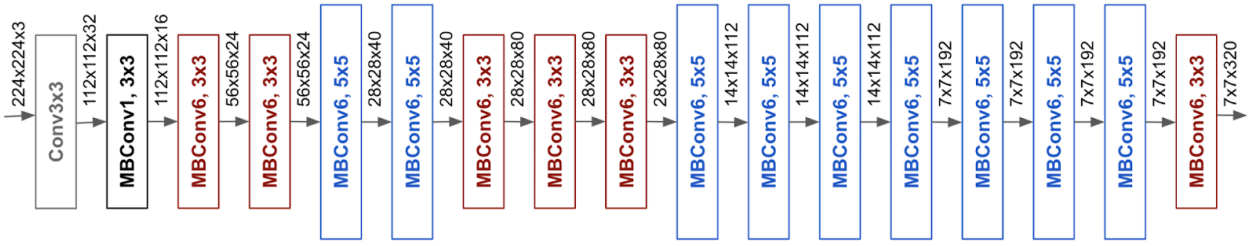


Figure 2.3: EfficientNet-B0 architecture (Source: [15])

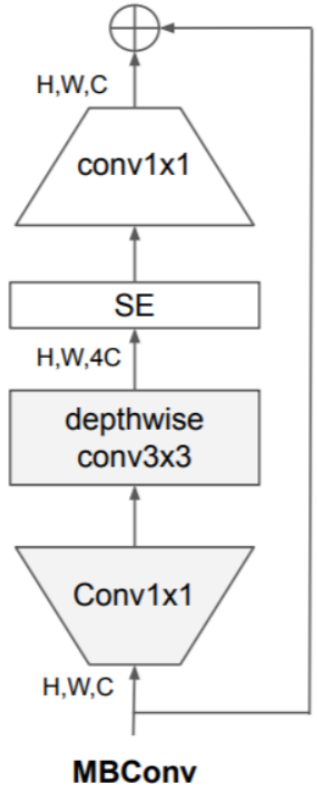


Figure 2.4: Structure of a MBConv layer (H, W, and C stand for Height, Width, and Channels resp.) (Source: [16])

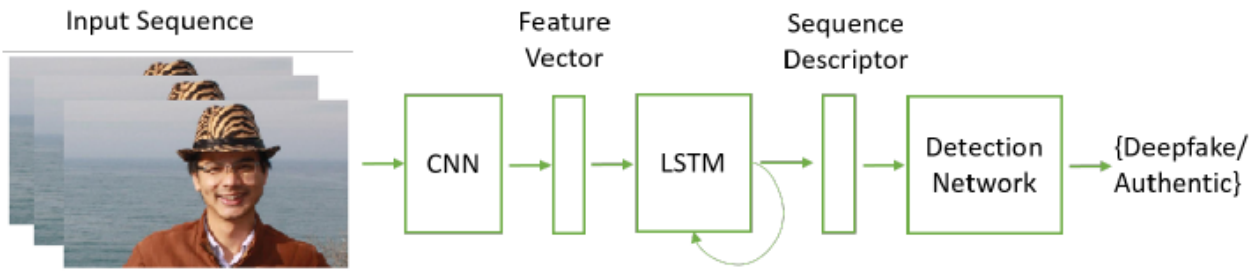


Figure 2.5: Architecture of a RCNN (Source: [18])

2.4 RCNN

Recurrent Convolutional Neural Networks (RCNN, sometimes abbreviated RCN) are another interesting type of model for deepfake detection. RCNN are a composed of two sub-networks, one CNN and one LSTM (Long Short-Term Memory) [17]. The CNN is used for feature extraction and the LSTM is used for temporal analysis. In deepfakes, inconsistencies may appear between consecutive frames and the spatiotemporal analysis of the frames can be more effective at detecting them. In a RCNN, the input sequence of images is fed into the CNN which returns a spatial feature vector. This feature vector is then fed into the LSTM layer which returns a sequence descriptor representing extracted temporal features. At the end of the RCNN, a fully connected layer to calculates probabilities in order to classify the input as "Deepfake" or "Authentic".

2.5 MesoNet (Meso-4)

In 2018, Afchar *et al.* proposed a method to detect face modifications in videos, in an automated and effective way, especially for deepfakes [19]. The term MesoNet was coined based on the words "mesoscopic" and "network". Mesoscopic means between microscopic and macroscopic, Afchar *et al.* claim to use this approach to avoid the disadvantages of microscopic analysis (image noise cannot be easily leveraged in compressed videos) and macroscopic analysis (human eyes have a hard time detecting facial forgeries). Afchar *et al.* introduced two models, Meso-4 and MesoInception-4, which they claim had the best classification scores during their experiments. We chose to use the Meso-4 network in our project. The architecture of this network is relatively

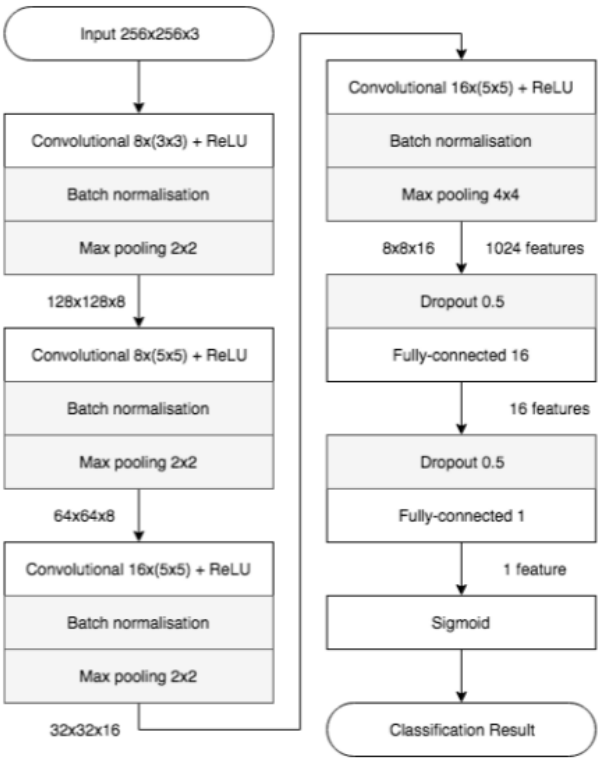


Figure 2.6: Architecture of Meso-4 network (Source: [19])

simple (Figure 2.6). It uses 4 successive layers of convolution with ReLU activation function for generalization, batch normalization for regularization, and max pooling. Then, to finish the classification, 2 sequences of a fully connected layer and a dropout layer are used before feeding their output into a Sigmoid function (for 2-class classification).

2.6 Datasets

In order to create a viable deepfake detection programme the choice of dataset used to train the model is crucial. During our research phase we came across what could be characterized as the three most popular datasets. FaceForensics++ [20] was originally an extension of the FaceForensics database (orignally centered on expression swap). It contains 1,000 original videos taken from YouTube, equating to 509,914 images. It also contains 2,000 fake videos, half of which were created using FaceSwap [21] which swaps the face of a given footage with another face provided separately. It uses face alignment, Gauss Newton optimization and image blending. The other half

were created using the opensource Deepfake FaceSwap implementation [22].

The Celeb-DF dataset [23] was published in 2019 by Li *et al.*. They claim that the visual quality of the deepfake videos is better than other popular datasets including Google's DeepFake Detection, FaceForensics++, and Facebook's DeepFake Detection Challenge datasets because it contained way less visual artifacts such as low quality synthesized faces, partially replaced faces, or colour mismatch. The Celeb-DF dataset contains 590 original videos downloaded from YouTube. These videos feature 59 celebrities of various ethnicity, gender, and age. From these authentic videos, 5,639 deepfake videos of 13 seconds at 30 FPS (on average) were created.

The Deepfake Detection Challenge (DFDC) dataset [24] was the one which retained the most our attention. It was created by Dolhansky *et al.* from Facebook AI and released in 2019. We came across this dataset because it was the one given to the Kaggle users who participated to the DFDC [25]. This dataset is extremely large and contains over 128,000 videos from 3,426 paid actors, among them 104,500 deepfake videos created with eight facial modification algorithms, making it the larger known deepfake dataset. However, such a large number of videos was equal to a ridiculous 25 TB of data. Therefore, Facebook processed those videos to provide a smaller and more usable subsample of 5,000 videos (approximately 470 GB). It was also impossible for us to use this dataset, but fortunately, Kaggle provided a subsample of this subsample containing only 800 videos (10 seconds long at a rate of 29.97 FPS) for a total of 4.44 GB which was way easier to download. Thus, we decided to use this reduced dataset for our project.

3 Assessing the Difference Between Two Consecutive Frames

In order to detect a deepfake, we considered two methods: comparing consecutive frames of a given video, and using deep neural networks. In this chapter we will talk about the former method.

In authentic videos, the transition between consecutive frames is smooth to some extent, meaning that two consecutive frames are not very different. But, sometimes, artefacts can appear on a deepfake video which may increase the difference between two consecutive frames too much compared to an authentic video. We thought that a good way to detect deepfakes was to compare consecutive frames by measuring the difference between them and comparing it to a threshold below which frames are too different for an authentic video. To do so we thought of two solutions: measure the mean squared error (MSE), or measure the structural similarity index (SSIM).

3.1 Mean Squared Error

The MSE between two images x and y [26] is defined by the following equation:

$$\mathrm{MSE}(x,y) = \frac{1}{MN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} \|x(i,j) - y(i,j)\|^2 \tag{3.1}$$

with

- M the number of rows of the arrays representing x and y
- N the number of columns of the arrays representing x and y
- i and j indexing rows and columns, respectively

In statistics, the MSE measures the precision of an estimator, but here it measure how far one image is from the other. The higher the MSE value, the more different the image. A MSE equal to 0 means that both images are the same.

However, as we preferred finding a more precise approach, we got interested in the following method.

3.2 Structural Similarity Index

The structural similarity index (SSIM) is a more recent formula which was defined by Wang *et al.* in [27]. For two images X and Y, the SSIM is defined by the following equation:

$$\text{SSIM}(x_j, y_j) = \frac{(2\mu_{x_j}\mu_{y_j} + C_1)(2\sigma_{x_j y_j} + C_2)}{(\mu_{x_j}^2 + \mu_{y_j}^2 + C_1)(\sigma_{x_j}^2 + \sigma_{y_j}^2 + C_2)} \tag{3.2}$$

with

- x_j and y_j are the image contents at the j^{th} local window
- μ_{x_j} and μ_{y_j} are the arithmetic means of all the pixels in x_j and y_j, respectively
- σ_{x_j} and σ_{y_j} are the standard deviations of all the pixels in x_j and y_j, respectively
- $\sigma_{x_j y_j}$ is the cross-covariance of x_j and y_j

C_1 and C_2 are two constants added to "avoid instability" when $\mu_{x_j}^2 + \mu_{y_j}^2$ and $\sigma_{x_j}^2 + \sigma_{y_j}^2$ tend to 0 (respectively).

With $K_1 \ll 1$ and $K_2 \ll 1$ two arbitrary positive constants, and L the dynamic range of the pixel values, C_1 and C_2 are defined as follows:

$$\begin{cases} C_1 = (K_1 L)^2 \\ C_2 = (K_2 L)^2 \end{cases} \tag{3.3}$$

In practice, for two images X and Y the mean SSIM (MSSIM) is measured to return a value between 0 (totally different) and 1 (same images) and quantify how similar both images are.

$$\text{MSSIM}(X, Y) = \frac{1}{M} \sum_{j=1}^{M} \text{SSIM}(x_j, y_j) \tag{3.4}$$

with M the number of local windows in the image.

3.2.1 Influence of C_1 and C_2

The introduction of the two arbitrary constants K_1 and K_2 made us doubt the necessity of C_1 and C_2. In order to asses their influence in the equation, let us introduce two factors S_1 and S_2 defined

as follows.

$$S_1(x_j,y_j) = \frac{(2\mu_{x_j}\mu_{y_j} + C_1)}{(\mu_{x_j}^2 + \mu_{y_j}^2 + C_1)} \qquad S_2(x_j,y_j) = \frac{(2\sigma_{x_jy_j} + C_2)}{(\sigma_{x_j}^2 + \sigma_{y_j}^2 + C_2)} \tag{3.5}$$

such that

$$\mathrm{SSIM}(x_j,y_j) = S_1(x_j,y_j) \times S_2(x_j,y_j) \tag{3.6}$$

C_1 is introduced to avoid instability when $\lim(\mu_{x_j}^2 + \mu_{y_j}^2) = 0$. However, $\mu_{x_j}^2 \geq 0$ and $\mu_{y_j}^2 = 0 \geq 0$. Thus:

$$\lim(\mu_{x_j}^2 + \mu_{y_j}^2) = 0 \implies \lim \mu_{x_j} = \lim \mu_{y_j} = 0 \implies \lim \mu_{x_j}\mu_{y_j} = 0 \tag{3.7}$$

So in this case, if we consider $C_1 = 0$, $\lim\limits_{\substack{\mu_x \to 0 \\ \mu_y \to 0}} S_1 = \dfrac{0}{0}$.

C_2 is introduced to avoid instability when $\lim(\sigma_{x_j}^2 + \sigma_{y_j}^2) = 0$. However, $\sigma_{x_j} \geq 0$ and $\sigma_{y_j} \geq 0$. Therefore:

$$\lim(\sigma_{x_j}^2 + \sigma_{y_j}^2) = 0 \implies \lim \sigma_{x_j} = \lim \sigma_{y_j} = 0 \tag{3.8}$$

According to the Cauchy-Schwarz inequality, we have:

$$\left| \sum_{i=1}^{N} (x_i - \mu_x)(y_i - \mu_y) \right| \leq \left(\sum_{i=1}^{N} (x_i - \mu_x)^2 \right)^{1/2} \left(\sum_{i=1}^{N} (y_i - \mu_y)^2 \right)^{1/2}$$

which leads to

$$\frac{1}{N-1} \left| \sum_{i=1}^{N} (x_i - \mu_x)(y_i - \mu_y) \right| \leq \left(\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \mu_x)^2 \right)^{1/2} \left(\frac{1}{N-1} \sum_{i=1}^{N} (y_i - \mu_y)^2 \right)^{1/2}$$

hence

$$\boxed{|\sigma_{x_jy_j}| \leq \sigma_{x_j}\sigma_{y_j}} \tag{3.9}$$

Moreover, $0 \leq |\sigma_{x_jy_j}|$. So when $\lim(\sigma_{x_j}^2 + \sigma_{y_j}^2) = 0$, we have according to (3.8):

$$0 \leq \lim_{\substack{\sigma_x \to 0 \\ \sigma_y \to 0}} |\sigma_{x_jy_j}| \leq \lim_{\substack{\sigma_x \to 0 \\ \sigma_y \to 0}} (\sigma_{x_j}\sigma_{y_j}) \leq 0$$

$$\implies \lim_{\substack{\sigma_x \to 0 \\ \sigma_y \to 0}} |\sigma_{x_jy_j}| = 0$$

$$\implies \lim_{\substack{\sigma_x \to 0 \\ \sigma_y \to 0}} \sigma_{x_jy_j} = 0$$

Therefore, as in the case of S_1, if we consider $C_2 = 0$, $\lim\limits_{\substack{\sigma_x \to 0 \\ \sigma_y \to 0}} S_2 = \dfrac{0}{0}$.

In both cases, $\lim S_1$ and $\lim S_2$ are undefined, and, when the denominator tends to 0, the numerator also tends to 0. This shows that, indeed, C_1 and C_2 do avoid instability. However, arbitrary constants are not elegant so we tried to find a way to justify removing them from the equation without adding instability.

3.2.2 Suggestions for Improvements

Now let us consider $C_1 > 0$ and $C_2 > 0$. This yields:

$$\lim_{\substack{\mu_x \to 0 \\ \mu_y \to 0}} S_1(x_j, y_j) = \frac{C_1}{C_1} = 1 \quad \text{and} \quad \lim_{\substack{\sigma_x \to 0 \\ \sigma_y \to 0}} S_2(x_j, y_j) = \frac{C_2}{C_2} = 1 \tag{3.10}$$

This means that with the existence of constants positive, non-zero constants C_1 and C_2, S_1 and S_2 will tend to 1 when $\mu_{x_j}^2 + \mu_{y_j}^2$ and $\sigma_{x_j}^2 + \sigma_{y_j}^2$ tend to 0 (respectively).

Therefore, although the limit may be undefined if C_1 or C_2 were not introduced, it is correct from an algorithmic point of view to set S_1 and S_2 to 1 when they are null.We can redefine S_1 and S_2 as follows:

$$S_1(x_j, y_j) = \begin{cases} 1 & \text{; if } \mu_x \to 0 \text{ and } \mu_y \to 0 \\ \dfrac{(2\mu_{x_j}\mu_{y_j} + C_1)}{(\mu_{x_j}^2 + \mu_{y_j}^2 + C_1)} & \text{; else} \end{cases} \tag{3.11}$$

$$S_2(x_j, y_j) = \begin{cases} 1 & \text{; if } \sigma_x \to 0 \text{ and } \sigma_y \to 0 \\ \dfrac{(2\sigma_{x_j y_j} + C_2)}{(\sigma_{x_j}^2 + \sigma_{y_j}^2 + C_2)} & \text{; else} \end{cases} \tag{3.12}$$

S_1 and S_2 are not null, C_1 and C_2 have very little influence on the final result. By extension, as C_1 and C_2 are "small" constants, they can be removed completely from the equation.

3.2.3 Suggestion of Changes in `scikit-image` Library

After looking at the issue from a theoretical point of view, it was important to see whether these changes could be applied in the implementation of of the function in Python. The most used implementation of the SSIM in Python is the function `structural_similarity()` contained in the `scikit-image` package [28]. This function is also based on [27] so it uses the same mathematical expression and the same parameters to compute and return the MSSIM. The suggested changes are

shown in Listing 3.1. `ux`, `uy`, `vx`, `vy`, and `vxy` respectively represent μ_x, μ_y, σ_x^2, σ_y^2, and σ_{xy}. The constant `R` represents L the dynamic range of the pixel values.

In the original function `K1` and `K2` can be specified by the user as arguments, by default they are set equal to `0.01` and `0.03` respectively (as suggested in [27]). A test is done to verify that `K1` and `K2` are not negative. However, a value of `0` is accepted for both constants. After running the original code with different images, and with `K1=0` and `K2=0`, the returned MSSIM is not a number (`NaN`). This is due to invalid divisions by 0 which lead to the SSIM (`S` in the code) containing `NaN` elements. The same happens when only `K1=0` or `K2=0`. To fix this issue we implemented what explained previously.

As shown in Listing 3.1, we introduced appropriate factors `S1` and `S2` such that `S = S1 * S2`. The difference here is that, instead of testing zero values for `ux`, `uy`, `vx` or `vy`, we see if the denominators `B1` and `B2` are equal to `0.0`. Every element of `S1` (resp. `S2`) where `B1` (resp. `B2`) is equal to `0.0`, is a `NaN`. Therefore, it is set to `1.0`.

3.2.4 Results

We ran different tests to observe the behaviour of our corrected structural similarity index code compared to the `scikit-image` implementation. Our goal was also to see whether the measure of the structural similarity index could be used for deepfake detection. For these tests we created simple images on *Paint* (Figure 3.1), and we took snapshots of videos (Figure 3.2) to have more complex images. Morevoer, in order to have to run both functions in the same conditions, we set `K1 = K2 = 1e-12` in both cases. Note that `K1 = K2 = 0` could not be used because the output of `scikit-image`'s algorithm would be a `NaN`, and thus not comparable.

The first test was to compute the MSSIM for 2 exact same images. We used one blue ball image (`ball1.png`) then we used one deepfake snapshot (`df1.png`). In both cases the MSSIM was equal to exactly `1.0`. This result is logical and confirms that our implementation works.

The second test was to compare two images that were slightly different from each other. We used `df1.png` and `df2.png` which are two snapshots taken from the same video, a few seconds apart. Our implementation returned `0.7664`, and the original one returned `0.6795`. These values show that the two images are indeed quite similar but not totally. Also, they have a small difference of about 0.09.

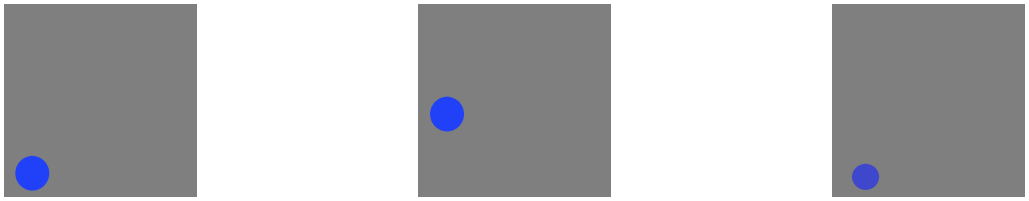
```
    R = data_range
    C1 = (K1 * R) ** 2
    C2 = (K2 * R) ** 2

    A1, A2, B1, B2 = ((2 * ux * uy + C1,
                       2 * vxy + C2,
                       ux ** 2 + uy ** 2 + C1,
                       vx + vy + C2))
    D = B1 * B2
    # [-removed] S = (A1 * A2) / D

    # [+added]
    S1 = A1 / B1
    S2 = A2 / B2

    # When K1 = 0 and/or K2 = 0, set NaN elements of S1 and S2 to 1
    S1[B1 == 0.0] = 1.0
    S2[B2 == 0.0] = 1.0

    S = S1 * S2
```

Listing 3.1: Modified part of `structural_similarity()` function in `scikit-image`

The third test was to compare snapshot from two different videos (`df4.png` and `aut1.png`). In theory the two images are completely different but can have common features, thus we would expect a low, non zero value. After running the program, we had `0.2353` and `0.2622` for our code and `scikit-image` implementation respectively. These values confirm our expectations. Moreover, this time the difference between the two outputs is of about 0.03. However, this time the result of our implementation is larger than the other one. But most importantly, this test shows that the MSSIM implementation detects brutal changes between two images.

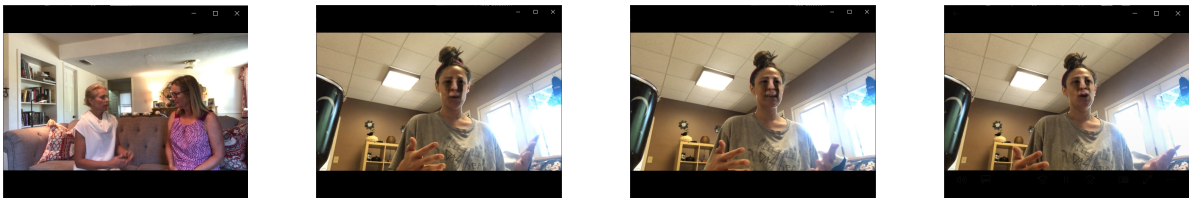
Finally, we run two more tests using simpler images. With `ball1.png` and `ball2.png` only the position of the ball changes. For our implementation we get `0.9303` and for the other implementation we get `0.9604`. There is still a slight difference of 0.03 between the two measure. With `ball1.png` and `ball4.png` the size and the colour are slightly changed. In the same order we get `0.9614` and `0.9770`. In both cases, the similarity is really high showing that the function sees the images as very similar. This is correct with respect to pixel values. Indeed, only the position of some pixels change in the first case , and a very small number of pixels change values in the second one. However, the change of position of the ball as an object should be seen as a brutal change for our measure to have a meaning.

In all the cases, both implementations have similar values. This means that our implementation of the MSSIM is legitimate. Nonetheless, this measure seems to be very effective in quantifying the difference between two very different images (pixel wise), but, it is not effective in quantifying the difference between two images which have too many pixels in common. As far as deepfakes are concerned, this would mean that if a flaw appeared around the area of the face, the MSSIM measure would not really be impacted as long as the number of impacted pixels is low enough. Therefore, given this piece of information and our experimental results we decided not to go further with using this method to detect deepfakes.



(a) `ball1.png` (b) `ball2.png` (c) `ball4.png`

Figure 3.1: Images representing a blue ball, created with *Paint*



(a) `aut1.png` (b) `df1.png` (c) `df2.png` (d) `df4.png`

Figure 3.2: Snapshots of videos. Image (a) is from an authentic video and the three others (b), (c), and (d) are from the same deepfake video

4 Deepfake Detection Using Neural Networks

The majority of the work done on deepfake detection uses deep neural networks. In this part, we will detail the process of creating a deepfake detection algorithm that uses deep neural networks.

To build an efficient deepfake detection programme it is important to have a well defined strategy. Intuitively, if one wants to look for a potential deepfake, they need to find and isolate the face in the video to be analyzed. Indeed, this reduces the amount of data to be processed, be it for a human being or an artificial neural network. Thus, one of the first things to do is to use a face detection algorithm in order to extract the faces from every frame of the videos that will be used to train the network, then save them in a a dedicated folder. The dataset must also be split into training, validation, and test sets. Then, after being designed, the model has to be trained. Finally, it is important to test the model in order to assess its accuracy and possibly improve it.

4.1 Face Detection Methods

Two face detection algorithms have retained our attention: OpenCV's Cascade Classifiers [29] and MTCNN [30].

4.1.1 Cascade classifiers

OpenCV provides pretrained cascade classifiers which can be loaded via XML files containing the feature sets. Haar cascade models can be used for various purposes such as detecting frontal faces, frontal cat faces, or Russian license plates, and for each of these features there is a specific XML file.

The method of using Haar-like features for face detection was first proposed by Paul Viola and Michael Jones in [31] which served as a basis for OpenCV's Cascade Classifier class. Viola and Jones created Haar-like features based on Haar wavelets. In Figure 4.1, the main Haar features

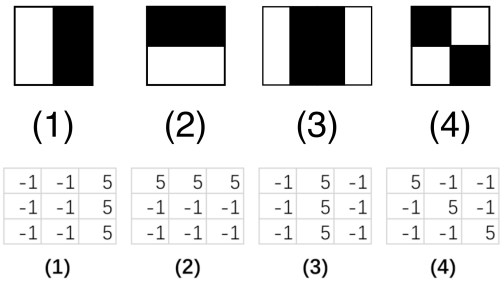


Figure 4.1: The four different types of Haar features used in [31] and a their possible numerical values (Source: [32] and [33] resp.)

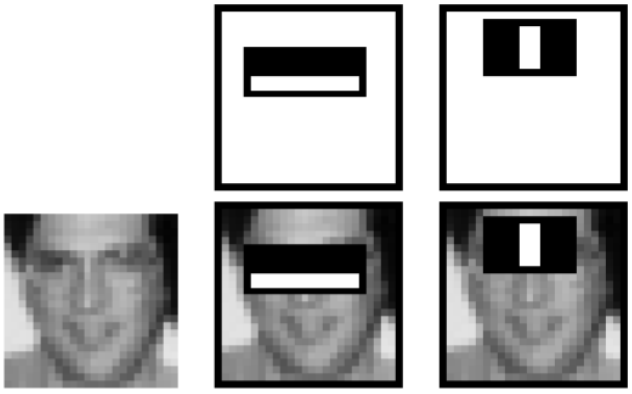


Figure 4.2: Example of Haar features applied to a face (Source: [31])

which are used by Viola and Jones are showed. In this figure, (1) and (2) are two-rectangle edge features, (3) is a three-rectangle line feature, and (4) is a four-rectangle diagonal feature.

The cascade classifier model is trained using positive and negative images (images with and without human faces respectively) and AdaBoost (Adaptive Boosting). Instead of using a large number of Haar features for each image, the images go through a number of steps, each step being a group of Haar features. When a sub-window of the image fails at one stage, it is rejected. For a sub-window to be classified as representing a face, it has to succeed at each stage of the process. When Haar features are applied they compare the intensity of the dark region with the light one, the result is a single value obtained by subtracting the sum of the of pixels in the dark region to the sum of pixels in the light region. For instance, in Figure 4.2, in the first features compares the region of the eyes and the region under them which is lighter, and the second one compares the region of the bridge of the nose to the region of the eyes which is darker.

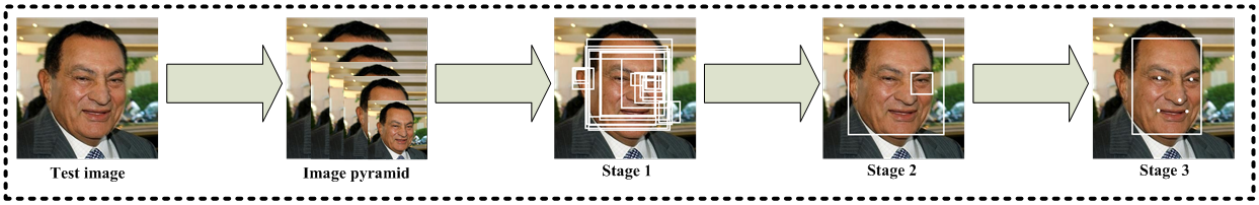


Figure 4.3: Pipeline of MTCNN cascaded network proposed by Zhang *et al.* (Source: [34])

4.1.2 MTCNN

MTCNN stands for Multi-Task Convolutional Neural Network. Zhang *et al.* proposed in [34] proposed a method that uses MTCNN to perform joint face detection and alignment. They claim that this method is more adapted and more efficient in real-world situations than the Viola-Jones framework. The proposed model consists of cascaded CNNs, composed of three stages. The process is shown in Figure 4.3. After the image is resized, the first CNN (Proposal Network, P-Net) quickly selects potential windows, then a more complex CNN (Refine Network, R-Net) discards the majority of windows which do not represent a face. Finally, the last CNN (Output Network, O-Net), which is more powerful, processes the last remaining windows to output the faces and their landmarks' positions (eyes, mouth, etc.).

In this report we used the implementation published on Github by De Paz Centeno [30]. It can be installed as a PIP package and uses TensorFlow.

4.2 Proposed Pipeline for Face Detection and Extraction

In this part, we propose a method to extract the faces which are featured in a given video. The first step is to open and read the video. This done by calling `VideoCapture(videoPath)` from OpenCV, `videoPath` being the path of video. To reduce the execution time and the amount of generated data, we only consider some of the frames of the video. With `frame_number` the 0-based index of the frame to be considered and `frame_rate` the frame rate (in FPS) , we only apply the process to the frames obeying the condition `frame_number % math.floor(frame_rate) == 0`. For example, a 10 second video (which is also the length of videos in [35]) with a frame rate of 29.97 FPS contains 299 frames. In this case, `math.floor(frame_rate)` is equal to 29, and the condition will be true for the case frame number is a multiple of 29 lower than 299, i.e. 0, 29, 58,

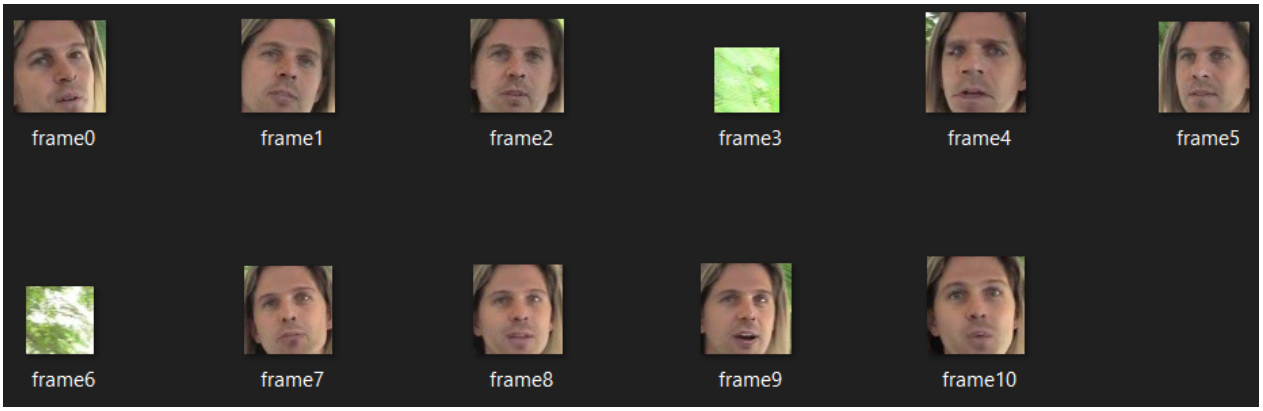


Figure 4.4: Regions detected as faces by the Haar cascade face detection function

87, 116, 145, 174, 203, 232, 261, 290. Thus, only up to 11 out of the 299 frames of the video will be processed. In a simpler way, we take 1 frame per second of video. Then, also in order to reduce the execution time, we resize the selected frame with the `imutils.resize()` function, after which the frame is ready to be processed by our face detection function (Listing 4.1). This first function uses an OpenCV cascade classifiers, it was inspired by S. Tiwari's GitHub repository [36]. It takes as arguments the image (or frame) and the path to the XML cascade classifier, and it returns a Boolean which is True when a face is found and the bounding box of the face. The returned bounding box is then cropped from the frame using the function `croppedRect()` function defined in Listing 4.2, that returns region of interest corresponding to the coordinates returned by the previous function.

```
1 def croppedRect(roi, x1, x2, y1, y2 ):
2     roi = roi[y1:y2, x1:x2]
3     return roi
```

Listing 4.2: `croppedRect()` function

Finally, after cropping the face from the frame, it is saved in a dedicated directory using OpenCV's `imwrite()` function. This prototype was tried on the video `aagfhgtpmv.mp4`. We deliberately used a deepfake video to complicate the face detection. The results are shown in Figure 4.4, only 9 out of 11 saved images are faces. The algorithm detected leaves instead of a face twice during the process.

We also used the MTCNN method to compare the results. The process is exactly the same, except that the `face_detection()` function is different (see Listing 4.3). This function, as shown

```python
def face_detection(image, cascPath):
    # Instantiate Cascade Classifier
    faceCascade = cv2.CascadeClassifier(cascPath)

    # Convert to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

    # Detect faces
    faces = faceCascade.detectMultiScale(
        gray,
        scaleFactor=1.1,
        minNeighbors=5,
        minSize=(30, 30)
    )

    # If there are faces return 'True'
    # Else return 'False'
    if len(faces) != 0:
        success = True
    else:
        success = False

    # Only return the 1st element of faces
    # Only 1 face is expected here
    face = faces[0]
    return (success, face)
```

Listing 4.1: `face_detection()` function using Haar Cascade Classifier

(a) Example of bounding box and landmarks (in orange) displayed on a face

```
>>> detector.detect_faces(img)
[
    {
        'box': [277, 90, 48, 63],
        'keypoints':
        {
            'nose': (303, 131),
            'mouth_right': (313, 141),
            'right_eye': (314, 114),
            'left_eye': (291, 117),
            'mouth_left': (296, 143)
        },
        'confidence': 0.99851983785629272
    }
]
```

(b) Example of a list of JSON objects returned by the MTCNN detector

Figure 4.5: Examples of returned outputs taken from [30]

below, takes as only input argument the frame to be processed, and returns a list of JSON objects as shown in Figure 4.5b. In our case, the most important items of this JSON object are the bounding box of the detected face `'box'` and the probability that it is a face `'confidence'`, and their respective values. In Figure 4.5a, an example of use of the returned JSON objects is shown. The orange rectangle corresponds to the bounding box, and the orange points correspond to the key points (eyes, nose and extremities of the mouth).

```
1 def face_detection(image):
2     detector = MTCNN()
3     results = detector.detect_faces(image)
4 
5     return results
```

Listing 4.3: `face_detection()` function using MTCNN face detection

We use the same video as previously to test this algorithm in order to be able to compare the results. In Figure 4.6, we see that, with the same video, a face is detected in every frame.

We observe that MTCNN face detection is more efficient than OpenCV's Haar cascade classifier. Indeed, by using convolutional neural networks MTCCN face detection is able to detect faces facing different directions where the Haar cascade classifier that we used only detects frontal faces accurately. Moreover, there is not much difference in execution times between the two techniques.

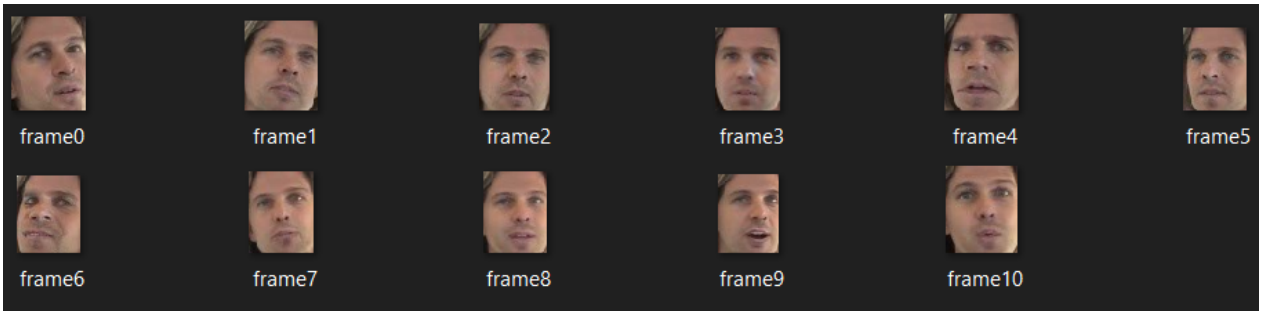


Figure 4.6: Regions detected as faces by the MTCNN face detection function

4.3 Inspiration and Issues

The code in Sections 4.5 - 4.9 is inspired by the work of A. Chong and H. Ng [37]. Their process is similar to ours. Each of the five Python programmes that they provide has a purpose. The first one `00-convert_video_to_image.py` is used to save a certain number of frames of each video into a folder. Then, one of the two files `01a-crop_faces_with_mtcnn.py` or `01b-crop_faces_with_azure-vision-api.py` is used to read the saved frames, detect the faces in them using MTCCN or the Azure Computer Vision API (respectively), crop them and save them into another directory. After this is done, by using `02-prepare_fake_real_dataset.py` the cropped faces are then split into training, validation, and test sets, each of them having *real* and *fake* subsets. These two last directories are used as labels. The last file `03-train_cnn.py` is then used to create the model, train it, and test it.

4.3.1 Issues Encountered While Testing the Code

Before doing anything it was important for us to run the code from [37] as provided and observe the results. In Figure 4.7, the final results of the process are shown. A prediction value of 1 means that the image belongs 100% to an authentic video, and a value of 0 means that it belongs 100% to a deepfake. Thus, a value above 0.5 indicates an authentic video, and a value below 0.5 indicates a deepfake. Except the first image, all the images were correctly classified yielding an impressive accuracy of $5/6$ even though only five videos were used to train the model. This seemed weird, so after reading the code more carefully, we discovered that A. Chong and H. Ng attributed 80% of the cropped faces to training, 10% to validation and 10% to testing (see Listing 4.4). In other words,

	Filename	Prediction
0	real\abarnvbtwb-000-00.png	0.197113
1	real\abarnvbtwb-002-00.png	0.868894
2	fake\aagfhgtpmv-003-00.png	0.373034
3	fake\aagfhgtpmv-009-00.png	0.234963
4	fake\aapnvogymq-001-00.png	0.126811
5	fake\aapnvogymq-004-01.png	0.016328

Figure 4.7: Output of `03-train_cnn.py` when running A. Chong and H. Ng's code

images from the same video were found in the training and validation sets, and also possibly in the testing set. This creates bias in the validation step and, to a certain extent, the model will predict that the tested image belongs to a known video in "real" or "fake" folder rather than a class. In the next sections we will suggest fixes and improvements.

```
# Split into Train/ Val/ Test folders
splitfolders.ratio(dataset_path, output='split_dataset', seed=1377, ratio
    =(.8, .1, .1))
```

Listing 4.4: Data splitting in `02-prepare_fake_real_dataset.py` in [37]

4.4 Used Dataset

Before starting to talk about programming, it is important to visualise the dataset that we used to train, validate and test our algorithm. As said before in Section 2.6, we chose the smaller dataset available for the DFDC on Kaggle. However, our goal is to have a proof of concept, we drastically reduced the number of videos and created an even smaller custom dataset. For training and validation, our dataset contained 14 videos including 4 originals and 10 deepfakes. The metadata of these videos is saved as JSON file. This `metadata.json` files contains the filenames of each videos, their label ("REAL" or "FAKE"), and, in the case of "FAKE", the filename of its original. For instance in Listing 4.5, the video `"cprhtltsjp.mp4"` is an original (thus "REAL") video so the key `"original"` have a value of `null`, and the video `"emfbhytfhc.mp4"` is a "FAKE" video created from `"cprhtltsjp.mp4"` which is its original. The item `"split"` was not used however we kept it as it was because it did not disturb the process. We also chose 6 videos for testing (3 authentic and 3 deepfakes). Moreover, we tried to keep the diversity of the dataset by choosing videos

featuring people from different gender and ethnicity, and videos with different lighting conditions and contrast.

```
"cprhtltsjp.mp4":{"label":"REAL","split":"train","original":null},
"emfbhytfhc.mp4":{"label":"FAKE","split":"train","original":"cprhtltsjp.
    mp4"}
```

Listing 4.5: Example of items contained in `metadata.json`

4.5 Data Splitting

For any machine learning model, the data has to be split into training, validation, and test sets. The training set is the set used to fit the model. The validation set is used to evaluate the model fit in an unbiased way and tune the model's hyperparameters. After training and validation, the final model fit has to be evaluated without bias using the testing set. When evaluating the accuracy of different competing models, the same testing set should be used for each of them.

For the validation and the testing steps to be unbiased and avoid *teaching* the wrong things to our model, their respective sets must be different from the training set. In order to accomplish this easily, instead of splitting the faces into three different sets, we split the videos into three folders. Manually, we select the videos that we want to use for training and validation and place them into a folder, we do the same for the videos we want to use for testing and place them in a different folder. Moreover, we add more videos from the Deepfake Detection Challenge [35] in order to have at least one video in each folder (14 videos for training and validation and 6 for testing, 20 in total).

In our method, we first run the data splitting programme. It stays almost the same except for the splitting part. Indeed, the videos used for testing are already in a separate folder so only the training and validation folders need to be created. As shown in Listing 4.6, 80% of the videos are assigned to testing and the remaining 20% are assigned to validation. Here, as in the code shown previously, the function `ratio()` from the package `splitfolders` is used. In Listing 4.6, `dataset_path` is the input path containing the videos, `output='split_dataset'` is the output folder, the random seed is set to 1377 (unchanged compared to previously), and `ratio=(.8, .2)` represents the splitting ratio.

It is also important to mention that in order to have "real" and "fake" sub-folders in both folders,

```
# Split into Train / Val folders
# With 80% / 20% ratio
splitfolders.ratio(dataset_path, output='split_dataset', seed=1377, ratio
    =(.8, .2))
```

Listing 4.6: Data splitting in `01-split_data.py`

we initially split the videos into such folders. This is done by using the `metadata.json` file which contains the names of all the videos along with data stating whether they are real or fake (see Listing 4.7).

```
"cprhtltsjp.mp4":{"label":"REAL","split":"train","original":null},
"emfbhytfhc.mp4":{"label":"FAKE","split":"train","original":"cprhtltsjp.
    mp4"}
```

Listing 4.7: Example of items contained in `metadata.json` caracterizing a deepfake and its original

4.6 Face Cropping Using MTCCN

Regarding the two first steps which A. Chong and H. Ng propose, we preferred to keep our initial method of detecting and cropping the faces directly from the videos instead of first saving frames then processing them to extract the faces.

Once the videos are saved in their dedicated folders, we perform the face detection and cropping process through all of them the same way it was done in Section 4.2. However, two things change: the way the frames are rescaled and the padding size of the cropped images.

When trying to run the programme with a low contrast video (see Figure 4.8), the results were not satisfying. A face was detected 3 times only, out of 11 possible. So we thought that it was not only because of the low contrast, but also because of the resizing of the frame which added difficulty to the detection process. After looking at the folder where the outputs of *DeepFake Detect* [37] were stored, we noticed that 5 faces were saved for the same video. Thus, we changed the rescaling function and used the one from [37].

Instead of writing it in line with the rest of the code, as it was done in [37], we preferred writing

Figure 4.8: Snapshot of the low contrast video used

it in a function called `rescale_frame()` (see Listing 4.8). `frame.shape[1]` is the width of the frame. when the frame is to small (width lower than 300), it is enlarged to double its original size. If it is too big (width higher than 1900) its size is divided by 3, when its width is between 1000 and 1900, the size of the frame is halved. If none of the previous conditions are met, then the frame remains unchanged.

```
def rescale_frame(frame):
    if frame.shape[1] < 300:
        scale_ratio = 2
    elif frame.shape[1] > 1900:
        scale_ratio = 0.33
    elif frame.shape[1] > 1000 and frame.shape[1] <= 1900 :
        scale_ratio = 0.5
    else:
        scale_ratio = 1

    width = int(frame.shape[1] * scale_ratio)
    height = int(frame.shape[0] * scale_ratio)
    dim = (width, height)
    frame = cv2.resize(frame, dim, interpolation = cv2.INTER_AREA)

    return frame
```

Listing 4.8: `rescale_frame()` function written by A. Chong and H. Ng [37]

As for the padding of the cropped images in the code from [37], we thought that it was a good idea to add also add it, as it was a way to include the data around the faces (hair, environment,

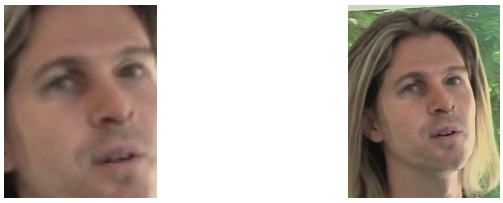


Figure 4.9: A cropped face without padding (left) and the same face (and same frame) with padding (right) - (Scales do not match)

lighting, etc.) in the dataset (see Figure 4.9). As shown in Listing 4.9, 30% of the width is added to right and the left of the image, and 30% of the height is added to the top and the bottom.

4.7 Designing the Model

The objective of the two last sections was to explain how we got our dataset. This section and the next ones until Section 4.9 explain how this dataset is used to obtain a deepfake detection programme.

The next step is to create the model that is going to be used for deep learning. In fact, we used three different models to compare the results and see which one is better suited for the task of deepfake detection. Firstly, we used an EfficientNet network as it was done originally by A. Chong and H. Ng.

4.7.1 EfficientNet-B0

We implemented the EfficientNet-B0 baseline model using the sequential method. The first "layer" is the actual EfficientNet-B0 network defined in Listing 4.10. The input shape is `(128, 128, 3)`. The weights used are those of the pretrained network with ImageNet reference dataset. The argument `include_top = False` means that no fully connected layer is added at the top of the network. Finally, global max pooling will be used for feature extraction. It is good to note that max pooling is a downsampling method which works by calculating the maximum value in patches of a feature map then using them to create a downsampled one.

The summary of the model is shown in Figure 4.10. We added one dropout and three dense layers. The dropout layer is used to randomly set a certain proportion of inputs (here 50%) to 0

```python
def padding(bounding_box, image):
    margin_x = bounding_box[2] * 0.3  # 30% as the margin
    margin_y = bounding_box[3] * 0.3  # 30% as the margin
    x1 = int(bounding_box[0] - margin_x)
    if x1 < 0:
        x1 = 0
    x2 = int(bounding_box[0] + bounding_box[2] + margin_x)
    if x2 > image.shape[1]:
        x2 = image.shape[1]
    y1 = int(bounding_box[1] - margin_y)
    if y1 < 0:
        y1 = 0
    y2 = int(bounding_box[1] + bounding_box[3] + margin_y)
    if y2 > image.shape[0]:
        y2 = image.shape[0]

    print(x1, y1, x2, y2)
    return x1, y1, x2, y2
```

Listing 4.9: `padding()` function (Source: [37])

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
efficientnet-b0 (Functional) (None, 1280)              4049564
_________________________________________________________________
dense (Dense)                (None, 512)               655872
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 129
=================================================================
Total params: 4,771,229
Trainable params: 4,729,213
Non-trainable params: 42,016
```

Figure 4.10: EfficientNet-B0 architecture in Tensorflow

```
efficient_net = EfficientNetB0(
    weights = 'imagenet',
    input_shape = (input_size, input_size, 3),
    include_top = False,
    pooling = 'max'
)
```

Listing 4.10: EfficientNet-B0 in Keras

at each step of the training. This means that at each step the model has a different "view" of the data, which prevents overfitting. The first two dense layers play a role of activation layers using the ReLU function, whereas the last one acts is a fully connected layer using the Sigmoid loss function. The latter is used for the actual classification, distinguishing images from fake videos from pristine ones.

4.7.2 MesoNet

The second model we used is MesoNet (Meso4) model. We used the official model as given by D. Afchar *et al.* in [38], the summary of which is given in Figure 4.11. The architecture matches what we described in Section 2.5.

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 128, 128, 3)]     0
_________________________________________________________________
conv2d (Conv2D)              (None, 128, 128, 8)       224
_________________________________________________________________
batch_normalization (BatchNo (None, 128, 128, 8)       32
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 64, 64, 8)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 64, 64, 8)         1608
_________________________________________________________________
batch_normalization_1 (Batch (None, 64, 64, 8)         32
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 32, 32, 8)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 16)        3216
_________________________________________________________________
batch_normalization_2 (Batch (None, 32, 32, 16)        64
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 16, 16, 16)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 16)        6416
_________________________________________________________________
batch_normalization_3 (Batch (None, 16, 16, 16)        64
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 16)          0
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 16)                4112
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 16)                0
_________________________________________________________________
dropout_1 (Dropout)          (None, 16)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 15,785
Trainable params: 15,689
Non-trainable params: 96
```

Figure 4.11: Meso-4 architecture in Tensorflow

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_1 (InputLayer)         [(None, 128, 128, 3)]     0         
_________________________________________________________________
conv2d (Conv2D)              (None, 128, 128, 10)      40        
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 128, 128, 3)       0         
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 128, 128, 10)      910       
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 128, 128, 10)      40        
_________________________________________________________________
concatenate (Concatenate)    (None, 128, 128, 20)      0         
                                                                 
_________________________________________________________________
flatten (Flatten)            (None, 327680)            0         
_________________________________________________________________
dense (Dense)                (None, 150)               49152150  
_________________________________________________________________
dense_1 (Dense)              (None, 150)               22650     
_________________________________________________________________
dense_2 (Dense)              (None, 2048)              309248    
_________________________________________________________________
reshape (Reshape)            (None, 1, 2048)           0         
_________________________________________________________________
lstm (LSTM)                  (None, 1, 512)            5244928   
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 513       
=================================================================
Total params: 54,730,479
Trainable params: 54,730,479
Non-trainable params: 0
```

Figure 4.12: RCNN architecture in Tensorflow

4.7.3 RCNN

The last model that we used is a RCNN model, combining a CNN and a LSTM (Figure 4.12). The CNN is an Inception model inspired by [39] and [40]. It takes an input of size `(128, 128, 3)`, then this input is processed in parallel by 2 groups of layers. The first one is a 2D convolutional layer (`conv2D_1`) applied to the result of another 2D convolutional layer (`conv2D`). The second one is 2D convolutional layer (`conv2D_2`) applied to the output of a 2D max pooling (`max_pooling2D`). The outputs of both groups are then concatenated, then flattened to be fed to dense layers. The output of the Inception part is then reshaped to fit the LSTM input dimensions. The LSTM model is based on [41]. We use a dropout rate of 0.5 and we set `return_sequences=True` to return the full sequence in the output sequence. The output of the LSTM network is flattened and the last last layer is a dense layer using the Sigmoid loss function to classify the images as taken from a deepfake or an authentic video.

```
val_datagen = ImageDataGenerator(
    rescale = 1/255    #rescale the tensor values to [0, 1]
)
test_datagen = ImageDataGenerator(
    rescale = 1/255    #rescale the tensor values to [0, 1]
)
```

Listing 4.12: Validation and testing data augmentation

4.8 Training the Model

Once designed, the models have to be trained using the previously preprocessed dataset. But before training we added a quick data augmentation step in order to virtually augment the size of the dataset. For example, to augment the training data we use the Tensorflow `ImageDataGenerator` which generates batches of tensor image data with real-time data augmentation following the parameters given as arguments. In Listing 4.11, the data is normalized (rescaled by a factor of 1/255). Moreover, we perform rotations, width and height shifts, random zooms, and horizontal flips. As shown in Listing 4.12, the validation and training sets are only normalized.

```
train_datagen = ImageDataGenerator(
    rescale = 1/255,     # rescale the tensor values to [0, 1]
    rotation_range = 10, # random rotation in [0deg, 10deg]
    width_shift_range = 0.1, # 10% of total width
    height_shift_range = 0.1, # 10% of total width
    shear_range = 0.2, # Shear Intensity
    zoom_range = 0.1, # range of random zoom [0.9, 1.1]
    horizontal_flip = True, # randomly flip inputs horizontally
    fill_mode = 'nearest' # filled following aaaaaaaa|abcd|dddddddd
)
```

Listing 4.11: Training data augmentation

Training the model starts with compiling it. For all three models we used Adam optimizer with a learning rate of 0.0001, and a the binary cross-entropy as loss function which computes the cross-entropy between the true class label and the predicted one. We also created custom callbacks (see

Listing 4.13) in order to set an early stopping condition and model checkpoints. We chose to stop the training when the validation loss (`val_loss`) stops improving. Also, we chose to save the best model with respect to validation loss as a `.h5` file (Hierarchical Data Format v5) so that it could be used directly for testing without necessarily retraining the model.

```
# Callback to save best model
custom_callbacks = [
    EarlyStopping(
        monitor = 'val_loss',
        mode = 'min',
        patience = 5,
        verbose = 1
    ),
    ModelCheckpoint(
        filepath = os.path.join(checkpoint_filepath, 'best_model_effnet.h5
    '),
        monitor = 'val_loss',
        mode = 'min',
        verbose = 1,
        save_best_only = True
    )
]
```

Listing 4.13: Custom callbacks for the EfficientNet model

The models are trained for 20 epochs (Listing 4.14). As said before, the training data is the augmented data returned by `train_generator`, similarly we use the validation data which is the normalized validation set `val_generator`. Each epoch contains `len(train_generator)` steps and there are `len(val_generator)` validation steps.

4.9 Tests and Results

After all of the preprocessing and model training is done, the model is ready to be tested with new data. We first used Tensorflow `load_model()` function to load the best model (as `best_model`) that was previously saved as a `.h5` file. Then we used the `best_model.predict()`

```
# Train network
num_epochs = 20
history = model.fit(
    train_generator,
    epochs = num_epochs,
    steps_per_epoch = len(train_generator),
    validation_data = val_generator,
    validation_steps = len(val_generator),
    callbacks = custom_callbacks
)
```

Listing 4.14: Model training using Tensorflow `model.fit()` method

method to classify the test data (Listing 4.15). Finally, we saved the results in a `.csv` file using the Pandas package. The line `np.where(preds.flatten()<0.5, 'Deepfake', 'Authentic')` means that predictions below the threshold of 0.5 are to be classified as 'Deepfakes' and prediction above as 'Authentic'.

The test results for the EfficientNet-B0 are shown in Table 4.1. Out of 21 images, 14 are correctly classified: 4/10 authentic ones and 10/11 fake ones. Although the accuracy seems pretty high, it seems that this model is better at detecting images taken from deepfake videos than from pristine ones.

The results using the Meso-4 network (see Table 4.2) are of a completely different nature. We observe a surprising similarity between the prediction values. Indeed, they all are approximately equal to 0.46 or 0.47. This also means that every images are classifed as 'Deepfake'. However, in this case it is not correct to say that the fake images are correctly classified because it is impossible for us to see any difference between the prediction of real images and fake ones.

In Table 4.3 the predictions are even more similar. Indeed, they are all the same until the fourth decimal (0.4607). Ultimately, all the images are seen as generating from deepfakes. As with the Meso-4 model, the accuracy has no real meaning here either.

The fact the results are not accurate and difficult to qualify can be caused by different reasons. The first issue that we thought of is the size of the dataset. In this project we used a very small quantity of videos, thus a small quantity of images to train and validate the model. Therefore,

	Filename	Prediction	Class
0	real\bcvheslzrq-00.png	0.35089687	Deepfake
1	real\bcvheslzrq-01.png	0.2679402	Deepfake
2	real\bcvheslzrq-02.png	0.56117463	Authentic
3	real\bcvheslzrq-03.png	0.3401838	Deepfake
4	real\bfdopzvxbi-00.png	0.49837095	Deepfake
5	real\bfdopzvxbi-01.png	0.36454496	Deepfake
6	real\keioymnobc-00.png	0.5286546	Authentic
7	real\keioymnobc-01.png	0.5812175	Authentic
8	real\keioymnobc-02.png	0.46915683	Deepfake
9	real\keioymnobc-03.png	0.5140669	Authentic
10	fake\bofrwgeyjo-00.png	0.43339753	Deepfake
11	fake\bofrwgeyjo-01.png	0.46117812	Deepfake
12	fake\bofrwgeyjo-02.png	0.4217006	Deepfake
13	fake\bofrwgeyjo-03.png	0.39647335	Deepfake
14	fake\etdliwticv-00.png	0.49017093	Deepfake
15	fake\etdliwticv-01.png	0.607801	Authentic
16	fake\etdliwticv-02.png	0.40214688	Deepfake
17	fake\kcjvhgvhpt-00.png	0.2619552	Deepfake
18	fake\kcjvhgvhpt-01.png	0.18961942	Deepfake
19	fake\kcjvhgvhpt-02.png	0.17028978	Deepfake
20	fake\kcjvhgvhpt-03.png	0.25352162	Deepfake

Table 4.1: Results using the EfficientNet-B0 model

	Filename	Prediction	Class
0	real\bcvheslzrq-00.png	0.46565488	Deepfake
1	real\bcvheslzrq-01.png	0.46909395	Deepfake
2	real\bcvheslzrq-02.png	0.46754164	Deepfake
3	real\bcvheslzrq-03.png	0.46489334	Deepfake
4	real\bfdopzvxbi-00.png	0.46473968	Deepfake
5	real\bfdopzvxbi-01.png	0.46944752	Deepfake
6	real\keioymnobc-00.png	0.46823287	Deepfake
7	real\keioymnobc-01.png	0.46416536	Deepfake
8	real\keioymnobc-02.png	0.46738422	Deepfake
9	real\keioymnobc-03.png	0.46338183	Deepfake
10	fake\bofrwgeyjo-00.png	0.46693966	Deepfake
11	fake\bofrwgeyjo-01.png	0.46728775	Deepfake
12	fake\bofrwgeyjo-02.png	0.46659684	Deepfake
13	fake\bofrwgeyjo-03.png	0.46727055	Deepfake
14	fake\etdliwticv-00.png	0.47450745	Deepfake
15	fake\etdliwticv-01.png	0.47150385	Deepfake
16	fake\etdliwticv-02.png	0.47302276	Deepfake
17	fake\kcjvhgvhpt-00.png	0.4665686	Deepfake
18	fake\kcjvhgvhpt-01.png	0.46921664	Deepfake
19	fake\kcjvhgvhpt-02.png	0.46997583	Deepfake
20	fake\kcjvhgvhpt-03.png	0.46881142	Deepfake

Table 4.2: Results using the Meso-4 model

	Filename	Prediction	Class
0	real\bcvheslzrq-00.png	0.46076813	Deepfake
1	real\bcvheslzrq-01.png	0.46076822	Deepfake
2	real\bcvheslzrq-02.png	0.46077085	Deepfake
3	real\bcvheslzrq-03.png	0.4607668	Deepfake
4	real\bfdopzvxbi-00.png	0.46076542	Deepfake
5	real\bfdopzvxbi-01.png	0.46076542	Deepfake
6	real\keioymnobc-00.png	0.46076542	Deepfake
7	real\keioymnobc-01.png	0.46076542	Deepfake
8	real\keioymnobc-02.png	0.46076542	Deepfake
9	real\keioymnobc-03.png	0.46076542	Deepfake
10	fake\bofrwgeyjo-00.png	0.4607655	Deepfake
11	fake\bofrwgeyjo-01.png	0.46076554	Deepfake
12	fake\bofrwgeyjo-02.png	0.46076554	Deepfake
13	fake\bofrwgeyjo-03.png	0.46076548	Deepfake
14	fake\etdliwticv-00.png	0.46076542	Deepfake
15	fake\etdliwticv-01.png	0.46076542	Deepfake
16	fake\etdliwticv-02.png	0.46076542	Deepfake
17	fake\kcjvhgvhpt-00.png	0.46076545	Deepfake
18	fake\kcjvhgvhpt-01.png	0.46076545	Deepfake
19	fake\kcjvhgvhpt-02.png	0.46076545	Deepfake
20	fake\kcjvhgvhpt-03.png	0.46076542	Deepfake

Table 4.3: Results using the RCNN model

```
preds = best_model.predict(
    test_generator,
    verbose = 1
)

test_results = pd.DataFrame({
    'Filename': test_generator.filenames,
    'Prediction': preds.flatten(),
    'Class': np.where(preds.flatten()<0.5, 'Deepfake', 'Authentic')
})

print(test_results)

# Export results to .csv format
test_results.to_csv('effnet_test_results.csv')
```

Listing 4.15: Classify test data and save the results in a `.csv` file (example for EfficientNet)

the networks could have lacked data to be able to produce accurate models. The solution for this issue is simply to use a bigger dataset, if possible the nearly 500 GB of videos DeepFake Detection Challenge dataset. Using such a large dataset also means that it could be necessary to use cloud computing solutions such as Microsoft Azure or Google Colab to train the models.

Another suggestion could be to create a "controlled dataset" in order to avoid using too much data. Creating a "controlled dataset" would mean to assemble videos containing only pristine videos and deepfakes derived from them. In this case, every deeepfake video should have its original and every original video should have at least one (preferably multiple) derived deepfake videos.

Finally, the nature of the data could also be one of the reasons our results were not accurate. Indeed, we only used single images to train, validate, and test the model. Because of this, the context of each image is lost. Indeed, the models see the images as unrelated entities even when they are consecutive to one another. To fix this issue and still use images instead of entire videos, it could be possible to use sequences of consecutive images in order for the model to detect possible flaws appearing between them.

5 Conclusion

Deepfakes are still new but are also improving at a fast pace. As researchers aim to tackle this issue, their work contributes to quick evolution of deepfake detection. The majority of published research uses deep neural network to solve the issue of deepfakes. In line with this method, we tried to develop a proof-of-concept of such a deepfake detection technique. Firstly, we studied the possibility of using the structural similarity index to measure the similarity or difference between two consecutive images. We came across a possible issue in the `scikit-image` implementation of the measure and fixed it. Even though the fixed code worked, we thought that the measures were not appropriate enough for our task of detecting deepfakes. Thus, we then started to think about the use of deep neural networks without focusing on a particular measure. For this task, the choice of dataset was important, and among various datasets, we chose to create our own, smaller one from the DeepFake Detection Challenge dataset. Then, we chose to use MTCNN face detection in order to detect and crop faces from the videos because it was more accurate than the Haar Cascade Classifiers without being slower. Furthermore, after finding data splitting error in [37], we fixed it and based the rest of our work on this repository. We modified the code to suit our needs and implemented three different neural networks to compare the results and find out which one was better suited for the task. Only the EfficientNet-B0 seemed to have coherent results. Unfortunately, Meso-4 and RCNN models did not perform as expected and it was difficult to draw a conclusion about their efficiency. It is highly likely that it is due to the nature and size of the dataset. Indeed, our dataset contained too few videos and the original videos did not have enough deepfake ones that were derived from them. Future work would involve solving this issue and trying to retrain the same models with a larger dataset using cloud computing.

A Appendix

In this section only a few Python files are copied. The rest (including these files) along the dataset and the outputs are available on **GitHub**.

A.1 Structural Similarity Index

```
import matplotlib.pyplot as plt
import numpy as np
import cv2
from numpy.core.defchararray import not_equal
from scipy.ndimage import gaussian_filter
from skimage.util.dtype import dtype_range
from skimage._shared.utils import check_shape_equality, warn
from skimage.util.arraycrop import crop
from _structural_similarity import stsim


def check_nan(**kwargs):
    arr_name = list(kwargs.keys())[0]
    arr_val = list(kwargs.values())[0]
    nan_nb = np.where(np.isnan(arr_val))[0].shape[0]

    if (nan_nb):
        print(f'{arr_name} contains {nan_nb} nans')

    else:
        print(f'{arr_name} does not contain nan')

# load images
# 1st test => same image
```

```
# img1 = img2 = im1 = im2 = cv2.imread("ball1.png")
# img1 = img2 = im1 = im2 = cv2.imread("df1.png")

# 2nd test => slightly different images
# img1 = im1 = cv2.imread("df1.png")
# img2 = im2 = cv2.imread("df2.png")

# 3rd test => completely different images
# img1 = im1 = cv2.imread("df4.png")
# img2 = im2 = cv2.imread("aut1.png")

# 4th test => different ball images (positions)
# img1 = im1 = cv2.imread("ball1.png")
# img2 = im2 = cv2.imread("ball2.png")

# 5th test => different ball images (size)
img1 = im1 = cv2.imread("ball1.png")
img2 = im2 = cv2.imread("ball4.png")

#-----------------------------------------------------------------
from skimage.metrics import structural_similarity as ssim

# im1 = cv2.cvtColor(im1, cv2.COLOR_BGR2GRAY)
# im2 = cv2.cvtColor(im2, cv2.COLOR_BGR2GRAY)

k1 = 1e-12
k2 = 1e-12
# s = ssim(im1, im2, K1=k1, K2=k2)
# k1 = k2 = 'default'
s = ssim(im1, im2, multichannel=True, use_sample_covariance=False,
    gaussian_weights=True, K1=k1, K2=k2)

print(f'For K1 = {k1} and K2 = {k2}\nskimage ssim = {s}')

s = stsim(im1, im2, multichannel=True, use_sample_covariance=False,
    gaussian_weights=True, K1=k1, K2=k2)
```

```
59
60 print(f'For K1 = {k1} and K2 = {k2}\nstsim = {s}')
```

Listing A.1: `ssim_clean.py` used to test the SSIM implementations

A.2 Face Detection and Cropping Prototypes

```
1  import cv2
2  import os
3  import imutils
4  from imutils.video import FPS
5  import math
6
7  def croppedRect(roi, x1, x2, y1, y2 ):
8      roi = roi[y1:y2, x1:x2]
9      return roi
10
11 def face_detection(image, cascPath):
12
13     # Create the haar cascade
14     faceCascade = cv2.CascadeClassifier(cascPath)
15
16     #Convert image to grayscale
17     gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
18
19     # Detect faces in the image
20     faces = faceCascade.detectMultiScale(
21         gray,
22         scaleFactor=1.1,
23         minNeighbors=5,
24         minSize=(30, 30)
25     )
26
27     if len(faces) != 0:
28         success = True
```

```
    else:
        success = False
    print(faces)
    face = faces[0]
    print(face)
    return (success, face)

def extract_faces(videoPath):
    # Check if directory exists, otherwise create it
    if not os.path.exists('./savedFrames'):
        os.mkdir('./savedFrames')

    haarCascade = './haarcascade_frontalface_default.xml'

    vs = cv2.VideoCapture(videoPath)

    # cv2.CAP_PROP_FRAME_COUNT
    frame_count = int(vs.get(7))
    frame_number = 0
    frame_rate = vs.get(5) # fps

    try:
        count = 0
        test_count = 0
        # Loop over frames from the video stream
        while(vs.isOpened()):
            frame_number = vs.get(1) # current frame number
            # Grab the current frame, then handle if we are using a
            # VideoStream or VideoCapture object
            ret, frame = vs.read()
            # print(np.shape(frame))
            # print(f'frame = {frame}')
            # print(frame[1].shape)

            if ret != True:
                break
```

```
65
66              if frame_number % math.floor(frame_rate) == 0:
67                  test_count = test_count + 1
68
69              if (frame_number % math.floor(frame_rate) == 0):
70                  # Resize the frame (so we can process it faster) and grab
    the
71                  # frame dimensions
72                  frame = imutils.resize(frame, width=500)
73
74                  # Grab the new bounding box coordinates of the object
75                  (success, box) = face_detection(frame, haarCascade)
76                  # Check to see if the tracking was a success
77                  # if success:
78                  (x, y, w, h) = [int(v) for v in box]
79
80                  # cv2.CAP_PROP_POS_FRAMES
81                  frame_number = int(vs.get(1))
82
83                  cv2.imwrite('./savedFrames/frame%d.jpg' % count,
    croppedRect(frame, x, x + w, y, y + h))
84                  count = count + 1
85
86              print(f'test_count = {test_count}')
87          vs.release()
88
89      except AttributeError:
90          print('Video read')
91
92
93 videoPath = './test_vid2.mp4'
94 extract_faces(videoPath)
```

Listing A.2: `detect_faces_and_save.py` Face detection using Haar Cascade Classifier

```
1 import cv2
2 import os
```

```
import imutils
import math
from mtcnn import MTCNN
import tensorflow as tf
print(tf.__version__)
tf.compat.v1.logging.set_verbosity(tf.compat.v1.logging.ERROR)

def croppedRect(roi, x1, x2, y1, y2 ):
    roi = roi[y1:y2, x1:x2]
    return roi

def face_detection(image):
    detector = MTCNN()
    results = detector.detect_faces(image)

    return results

def extract_faces(videoPath):
    # Check if directory exists, otherwise create it
    if not os.path.exists('./savedFrames_mtcnn'):
        os.mkdir('./savedFrames_mtcnn')

    vs = cv2.VideoCapture(videoPath)

    # cv2.CAP_PROP_FRAME_COUNT
    frame_count = int(vs.get(7))
    frame_number = 0
    frame_rate = vs.get(5) # fps

    try:
        count = 0
        test_count = 0
        # Loop over frames from the video stream
        while(vs.isOpened()):
            frame_number = vs.get(1) # current frame number
            # Grab the current frame, then handle if we are using a
```

```
            # VideoStream or VideoCapture object
            ret, frame = vs.read()
            # print(np.shape(frame))
            # print(f'frame = {frame}')
            # print(frame[1].shape)

            if ret != True:
                break


            if frame_number % math.floor(frame_rate) == 0:
                test_count = test_count + 1

            if (frame_number % math.floor(frame_rate) == 0):
                # Resize the frame (so we can process it faster) and grab
    the
                # frame dimensions
                frame = imutils.resize(frame, width=500)

                # Grab the new bounding box coordinates of the object
                res = face_detection(frame)
                # Bounding box
                bounding_box = res[0]['box']
                # Confidence
                confidence = res[0]['confidence']

                if confidence > 0.95:
                    # Check to see if the tracking was a success
                    x, y, w, h = bounding_box
                    # rect = cv2.rectangle(frame, (x, y), (x + w, y + h),
                    #     (0, 255, 0), 2)

                    cv2.imwrite('./savedFrames_mtcnn/frame%d.jpg' % count,
     croppedRect(frame, x, x + w, y, y + h))
                    count += 1
                else:
```

```
                    print('Skipped a face...')

            print(f'test_count = {test_count}')
        vs.release()

    except AttributeError:
        print('Video read')


videoPath = './test_vid2.mp4'
extract_faces(videoPath)
```

Listing A.3: `mtcnn_and_save.py` Face detection using MTCNN

A.3 Data Splitting Before Training

```
import json
import os
from distutils.file_util import copy_file
import shutil
import numpy as np
import splitfolders

base_path = '.\\train_sample_videos\\'
dataset_path = '.\\z_prepared_dataset\\'
print('Creating Directory: ' + dataset_path)
os.makedirs(dataset_path, exist_ok=True)

def get_filename_only(file_path):
    file_basename = os.path.basename(file_path)
    filename_only = file_basename.split('.')[0]
    return filename_only

with open(os.path.join(base_path, 'metadata.json')) as metadata_json:
    metadata = json.load(metadata_json)
```

```
    print(len(metadata))

real_path = os.path.join(dataset_path, 'real')
print('Creating Directory: ' + real_path)
os.makedirs(real_path, exist_ok=True)

fake_path = os.path.join(dataset_path, 'fake')
print('Creating Directory: ' + fake_path)
os.makedirs(fake_path, exist_ok=True)

for filename in metadata.keys():
    print(filename)
    print(metadata[filename]['label'])
    tmp_path = os.path.join(base_path, filename)
    if metadata[filename]['label'] == 'REAL':
        print('Copying to :' + real_path)
        # Copy entire source tree ('tmp_path') to 'real_path'
        copy_file(tmp_path, real_path)
    elif metadata[filename]['label'] == 'FAKE':
        print('Copying to :' + fake_path)
        # Copy entire source tree ('tmp_path') to 'fake_path'
        copy_file(tmp_path, fake_path)
    else:
        print('Ignored..')

all_real_faces = [f for f in os.listdir(real_path) if os.path.isfile(os.
    path.join(real_path, f))]
print('Total Number of Real videos: ', len(all_real_faces))

all_fake_faces = [f for f in os.listdir(fake_path) if os.path.isfile(os.
    path.join(fake_path, f))]
print('Total Number of Fake videos: ', len(all_fake_faces))

# Split into Train / Val folders
# With 80% / 20% ratio
splitfolders.ratio(dataset_path, output='z_split_dataset', seed=1377,
```

```
    ratio=(.8, .2))
print('Train / Val Split Done!')
```

Listing A.4: 01-split_data.py

References

[1] H. Ajder, G. Patrini, F. Cavalli, and L. Cullen, The state of deepfakes: Landscape, threats, and impact, *Amsterdam: Deeptrace* 2019, 2019.

[2] C. Stupp, Fraudsters Used AI to Mimic CEO's Voice in Unusual Cybercrime Case, *The Wall Street Journal* 2019, 2019.

[3] RepresentUs, *First Ever Use of Deepfake Technology in a Major Ad Campaign*, `https://act.represent.us/sign/deepfake-release/`, 2020.

[4] S. Cahlan, How misinformation helped spark an attempted coup in Gabon, *The Wahsington Post* 2020, 2020.

[5] Y. Li, M.-C. Chang, and S. Lyu, "In ictu oculi: Exposing ai created fake videos by detecting eye blinking," in *2018 IEEE International Workshop on Information Forensics and Security (WIFS)*, IEEE, 2018, pp. 1–7.

[6] F. Matern, C. Riess, and M. Stamminger, "Exploiting visual artifacts to expose deepfakes and face manipulations," in *2019 IEEE Winter Applications of Computer Vision Workshops (WACVW)*, IEEE, 2019, pp. 83–92.

[7] Y. Li and S. Lyu, Exposing deepfake videos by detecting face warping artifacts, *arXiv preprint arXiv:1811.00656* 2018, 2018.

[8] H. Li, B. Li, S. Tan, and J. Huang, Identification of deep network generated images using disparities in color components, *Signal Processing*, vol. 174 2020, p. 107 616, 2020.

[9] E. Kee and H. Farid, "Exposing digital forgeries from 3-d lighting environments," in *2010 IEEE International Workshop on Information Forensics and Security*, IEEE, 2010, pp. 1–6.

[10] A. Odena, V. Dumoulin, and C. Olah, Deconvolution and checkerboard artifacts, *Distill* [online] 2016, 2016. DOI: `10.23915/distill.00003`. available from: `http://distill.pub/2016/deconv-checkerboard`.

[11] J. Bunk, J. H. Bappy, T. M. Mohammed, L. Nataraj, A. Flenner, B. Manjunath, S. Chandrasekaran, A. K. Roy-Chowdhury, and L. Peterson, "Detection and localization of image forgeries using resampling features and deep learning," in *2017 IEEE conference on computer vision and pattern recognition workshops (CVPRW)*, IEEE, 2017, pp. 1881–1889.

[12] IBM, *Supervised Learning*, https://www.ibm.com/cloud/learn/supervised-learning, 2020.

[13] M. Tan and Q. V. Le, Efficientnet: Rethinking model scaling for convolutional neural networks, *CoRR* [online], vol. abs/1905.11946 2019, 2019. arXiv: 1905.11946. available from: http://arxiv.org/abs/1905.11946.

[14] B. Ahmadzadeh, *Implementing EfficientNet in PyTorch Part 3: MBConv, Squeeze-and-Excitation, and More*, https://python.plainenglish.io/implementing-efficientnet-in-pytorch-part-3-mbconv-squeeze-and-excitation-and-more-4ca9fd62d302, 2021.

[15] M. Tan and Q. V. Le, *EfficientNet: Improving Accuracy and Efficiency through AutoML and Model Scaling*, https://ai.googleblog.com/2019/05/efficientnet-improving-accuracy-and.html, 2019.

[16] A. Arora, *EfficientNetV2*, https://wandb.ai/wandb_fc/pytorch-image-models/reports/EfficientNetV2--Vmlldzo2NTkwNTQ.

[17] O. de Lima, S. Franklin, S. Basu, B. Karwoski, and A. George, Deepfake detection using spatiotemporal convolutional networks, *CoRR* [online], vol. abs/2006.14749 2020, 2020. arXiv: 2006.14749. available from: https://arxiv.org/abs/2006.14749.

[18] T. T. Nguyen, C. M. Nguyen, D. T. Nguyen, D. T. Nguyen, and S. Nahavandi, Deep learning for deepfakes creation and detection, *CoRR* [online], vol. abs/1909.11573 2019, 2019. arXiv: 1909.11573. available from: http://arxiv.org/abs/1909.11573.

[19] D. Afchar, V. Nozick, J. Yamagishi, and I. Echizen, Mesonet: A compact facial video forgery detection network, *CoRR* [online], vol. abs/1809.00888 2018, 2018. arXiv: 1809.00888. available from: http://arxiv.org/abs/1809.00888.

[20] A. Rössler, D. Cozzolino, L. Verdoliva, C. Riess, J. Thies, and M. Nießner, "FaceForensics++: Learning to detect manipulated facial images," in *International Conference on Computer Vision (ICCV)*, 2019.

[21] M. Kowalski, *FaceSwap, GitHub repository*, https://github.com/MarekKowalski/FaceSwap, 2016.

[22] V. contributors, *FaceSwap - Deepfakes Software For All, GitHub repository*, https://github.com/deepfakes/faceswap, 2018.

[23] Y. Li, X. Yang, P. Sun, H. Qi, and S. Lyu, Celeb-df: A new dataset for deepfake forensics, *CoRR* [online], vol. abs/1909.12962 2019, 2019. arXiv: 1909.12962. available from: http://arxiv.org/abs/1909.12962.

[24] B. Dolhansky, J. Bitton, B. Pflaum, J. Lu, R. Howes, M. Wang, and C. Canton-Ferrer, The DeepFake Detection Challenge Dataset, *CoRR* [online], vol. abs/2006.07397 2020, 2020. arXiv: 2006.07397. available from: https://arxiv.org/abs/2006.07397.

[25] Kaggle, *Deepfake Detection Challenge*, https://www.kaggle.com/c/deepfake-detection-challenge/, 2020.

[26] A. Dimech, *Structural Similarity Index (SSIM) in Python*, https://code.adonline.id.au/structural-similarity-index-ssim-in-python/, 2021.

[27] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, Image quality assessment: From error visibility to structural similarity, *IEEE transactions on image processing*, vol. 13 2004, pp. 600–612, 2004.

[28] S. Van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu, Scikit-image: Image processing in python, *PeerJ*, vol. 2 2014, e453, 2014.

[29] OpenCV, *Cascade Classifier*, https://docs.opencv.org/4.5.1/db/d28/tutorial_cascade_classifier.html.

[30] I. De Paz Centeno, *MTCNN Face Detection Package*, https://github.com/ipazc/mtcnn, 2018.

[31] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, vol. 1, 2001, pp. I–I. DOI: 10.1109/CVPR.2001.990517.

[32] Wikipédia, *Méthode de viola et jones — wikipédia, l'encyclopédie libre*, [En ligne; Page disponible le 12-octobre-2020], 2020. available from: http://fr.wikipedia.org/w/index.php?title=M%C3%A9thode_de_Viola_et_Jones&oldid=175497864.

[33] C.-F. Wang, *What's the Difference Between Haar-Feature Classifiers and Convolutional Neural Networks?* https://towardsdatascience.com/whats-the-difference-between-haar-feature-classifiers-and-convolutional-neural-networks-ce6828343aeb, 2018.

[34] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, Joint face detection and alignment using multitask cascaded convolutional networks, *IEEE Signal Processing Letters* [online], vol. 23 2016, pp. 1499–1503, 2016. DOI: 10.1109/LSP.2016.2603342.

[35] B. Dolhansky, R. Howes, B. Pflaum, N. Baram, and C. C. Ferrer, The deepfake detection challenge (dfdc) preview dataset, *arXiv preprint arXiv:1910.08854* 2019, 2019.

[36] S. Tiwari, *FaceDetect, GitHub repository*, https://github.com/shantnu/FaceDetect, 2014.

[37] A. Chong and H. Ng, *DeepFake-Detect, GitHub repository*, https://github.com/aaronchong888/DeepFake-Detect, 2020.

[38] D. Afchar, *MesoNet, GitHub repository*, https://github.com/DariusAf/MesoNet, 2018.

[39] J. Brownlee, *How to Develop VGG, Inception and ResNet Modules from Scratch in Keras*, https://machinelearningmastery.com/how-to-implement-major-architecture-innovations-for-convolutional-neural-networks/, 2019.

[40] M. Fabien, *A guide to Inception Model in Keras*, https://maelfabien.github.io/deeplearning/inception/#, 2019.

[41] N. Sinha, *Understanding LSTM and its Quick Implementation in Keras for Sentiment Analysis*, https://towardsdatascience.com/understanding-lstm-and-its-quick-implementation-in-keras-for-sentiment-analysis-af410fd85b47, 2018.