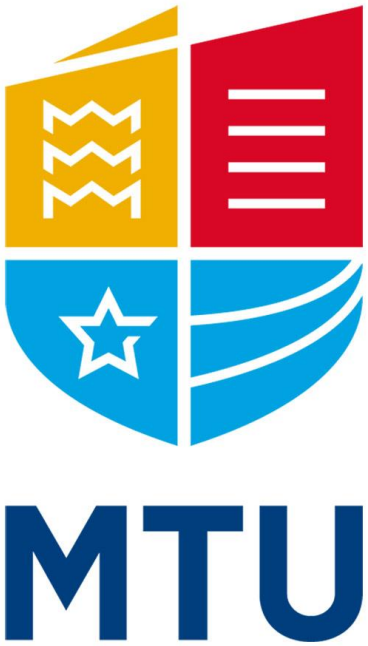

Identifying Photoshop in Facial Images with Deep Learning

Dimitrios Bikoulis

Student No: R00209268

Supervisor: Mark Hodnett

For the module DATA9003 – Research Project as part of the

Master of Science in Data Science and Analytics, Department of Mathematics, 30/08/2022

Declaration of Authorship

I, Dimitrios Bikoulis, declare that this thesis titled ‘Identifying Photoshop in Facial Images with Deep Learning’ and the work presented in it are my own. I confirm that,

- This work was done wholly or mainly while in candidature for the Masters’ degree at Munster Technological University
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Munster Technological University or any other institution, this has been clearly stated
- Where I have consulted the published work of others, this is always clearly attributed
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work
- I have acknowledged all main sources of help
- I understand that my project documentation may be stored in the library at MTU, and may be referenced by others in the future

Signed: ~~Dimitrios Bikoulis~~

Date: ~~30/08/2022~~

Acknowledgment

I would like to thank my supervisor Mr. Mark Hodnett for helping, guiding and

supporting me all this period to tackle this project and come up with great results.

Furthermore, I would like to thank each lecturer individually for successfully conveying their

knowledge and passion to reach my potentials and become a passionate data scientist.

Table of Contents

Table of Figures

Abstract (300 words)

Banks are the main institutions which are being targeted by fraudsters, and as a result, the amount of money lost annually is remarkably high. Detecting the fraudsters is the most important goal for a bank, since it provides reliability and security. The last 10 years, machine learning and deep learning perform immensely well in different fields. Big data is easily handled by stronger hardware systems than they used to be in the past or equivalently more information is available for use to tackle different problems which humans cannot. The influence of technology has created a world that allows several processes to be done well remotely. When a new account is created, the bank come up against a risk of fraud, unless the regulations are satisfied. Some of the documents which should be submitted in order for a new account to be created are a photograph of customer's ID card and an image displaying their face. The documents submitted by the prospective customer are checked by the bank and either get accepted or rejected. Fraudsters tend to submit fake ID cards or photoshopped images to hide their identity. In this research, the study of photoshopped images to identify potential frauds will be proposed. ConvNeXts have been shown to achieve great performance in image classification tasks and for this research they will be included in a voting classifier and trained on an imbalanced dataset of facial images. An imbalanced dataset is used to approach a real-world problem and the voting classifier of 4 ConvNeXt models and an ensemble model managed to correctly classify 91.67% of the fake images. The reference style which will be used in this paper is IEEE which stands for the Institute of Electrical and Electronics Engineers.

1. Introduction

Nowadays, technology is one of the most important tools to optimize and speed up any process. Year by year, hardware systems become stronger and stronger and therefore computers can be exploited to accomplish tasks, either complicated or not, that humans need significantly much more time to do so. Especially machine learning has accomplished great achievements in terms of performance & speed; and for many organizations, it is an important and irreplaceable tool with which their goals are easily achieved. Like in any other organization, machine learning is an extremely useful tool for banks, and it is used to overcome many difficulties and avoid risks. Some of the difficulties and risks that the banks come up against in every day life are fraud attacks where machine learning is trying to prevent any suspicious transactions or reject the creation of new bank accounts from suspicious customers, document processing, where the submitted and any relevant documents are classified, organized and checked for validity with a view of being retrieved in the near feature, etc.

Every day, hundreds thousands of people go to the bank in order to pay bills, withdraw some money, make a deposit, get loans, open bank accounts and so forth. Although, the last 10 years, when the smartphones have become stronger and stronger and many applications are available to download, banks have offered online services to speed up the whole process and facilitate their customers. In light of this, clients can easily open a bank account, ask for a loan or a credit card without necessarily should go to the bank and waiting on long queues for many hours. However, the aforementioned automated procedure comes with many risks and banks, which deal with an extremely large amount of money, are continuously being targeted by many fraudsters. Here is where deep learning comes into play and help banks defend fraud attacks with a view of minimizing the annual loss. In this project, we will be explicitly focusing on identification of fraudsters who try to create a new account.

More and more fraud attacks in banks happen which significantly increases the annual loss. In 2020, it was found that the annual total fraud loss stands at $56 billion. Identity fraud scams resulted $43 billion out of the total fraud loss according to [1]. When a new account is created an ID card and facial image are required for authentication. New account fraud implies the phenomenon when a fraudster uses a stolen or fake ID accompanied by a facial image to create a new bank account. There are three different types of fraud; first, second and third-party fraud. The first one occurs when a fraudster is intentionally misrepresenting his ID card and facial image to hide their identity with the purpose to deceive the algorithm used for authentication. The second-party fraud occurs when a fake ID and fake facial image are submitted to the bank for authentication. Lastly, the third-party fraud describes the submitted image of a stolen ID card. In each case, when a new account is successfully created by a fraudster, the sole purpose is to commit fraud, typically by asking for a credit card and then disappear. This paper will propose a method to eliminate the second-party fraud, and specifically to detect fake facial images. Deep learning has addressed anti-fraud detection techniques to identify fraud attacks by handling large amount of data, some of them in real time, when humans cannot. Specific to this paper, a model which will accurately identify preprocessed and photoshopped images from potential fraudsters will be built. The research question which is generated is the following:

"*How accurately can deep learning detect photoshop in facial images?*"

1.1 Project Background

Research on the pre-existing literature was conducted to find potential gaps and build the research question stated above. There has been limited literature around this area and none of them has been linked to fraud detection for banks. ConvNeXt which is a state-of-the-art architecture has been used in **[2]** for photoshop detection. The authors created an artificial dataset using Generative Adversarial Network (GAN), which is a model to create an artificial dataset and will follow the distribution of the original one. ConvNeXt was trained on it and the resulted accuracy was poor. In this project, how ConvNeXt performs in photoshop detection will be re-examined using the novel approach of data extension. In many domains, the imbalance between the classes is not something unusual. For instance, in medical domain, it is common to deal with an imbalanced dataset. In **[3]**, the authors try to deal with the severe imbalance in different tumor cases. It is worth mentioning that fraudsters continuously try to trick banks, however the ratio between suspicious and trustworthy customers is approximately 1:1000. In light of this, an imbalanced dataset would approach a real-world problem significantly better. Since, fraudsters try to hide their identify on submitted facial images is equivalent to a non-subtle photoshop which can easily be identified by humans. Although, due to the great number of images submitted every day, the fact that each image needs to be checked by humans is time consuming. In light of this, we will try to reduce the number of images that a human should check. We will try to build a model that can classify the fake images correctly and simultaneously reduce the misclassifications of the real images. Recall is a crucial metric for this project which needs optimizing and measures the performance of the model in identifying fake images. Focusing on achieving a great score for the aforementioned metric and simultaneously significantly reduce the real images predicted as fake is a novel problem that should be addressed.

1.2 Aims & Objectives

It has already been mentioned that one of the most challenging tasks for banks is eliminating new account fraud and addressing it, the annual loss will reduce. Recall and Precision are the most important metrics for this project. The former measures the performance of the model in identifying fake images by calculating the proportion of the correctly fake classified images over the total fake images. The latter describes how accurate the model is when it predicts fake. To measure the latter, the number of correctly fake predicted images is divided by the predicted fake images, and the resulted percentage will show how accurate the model is, when it predicts fake. The optimal recall is the main objective of this research and simultaneously, we will be trying to improve the precision as much as possible. To approach a real-world problem, there is a need to consider that the ratio between real and fake submitted images is approximately 1000:1 and therefore, it is irrelevant to address this task by using a balanced dataset. In light of this, the main objective of this research is to optimize the aforementioned metrics on an imbalanced dataset. A question that is generated is: "Why accuracy is not used as a metric and why recall is not enough, and precision should be optimized as well?". To answer these questions, we must think of a model which predicts only real or only fake. The former will result in a great accuracy (more than 99%) but it is considered as useless since it cannot catch any fake images. The latter will result in 100% recall and again the model is completely useless since it cannot recognize real images. Another question that is generated is "Why recall is more important than precision?". To answer this question, we must define that the model will be used as a filter. There is a trade-off between the two metrics. By maximizing the Recall, the fake images will be spotted by the model which is the most important factor to reduce the annual loss. Each fake predicted image will be checked by an individual to define whether it is indeed fake or real. To reduce the amount of work required by a human to check every fake predicted image leads to a need of maximizing the precision as well. By achieving high recall and simultaneously a relevant score for the precision, the model can be successfully used as a

filter and only suspicious images will be checked by the human. The process that a human intervenes to define the outcome of the model is called human-in-the-loop (HITL) and refers to the human intervention in the training and testing process in order to optimize the performance of the model. Excluding the main objective which is the optimization of Recall with a sufficient Precision score, another objective of this research is to address how an extension of the data can positively influence the performance of the model in the metrics stated above.

2. Literature Review (2240/3000)

There has been some research around this subject, and in this part, key concepts to answer the research question will be outlined, starting off with Computer Vision term and moving on to how machine learning, and more precisely, deep learning has reached a significantly good performance over these years on this field. Moreover, the key tools of deep learning used in computer vision will be outlined and how it helps banks overcome and avoid risks that negatively affects the annual incomes will be demonstrated. Lastly, related work around this field will be reviewed and described.

2.1 Computer Vision

2.1.1 Definition & History of Computer Vision

Computer Vision is a scientific field where computers try to acquire high-level understanding from images, videos or other visual inputs performing as good as or even better than humans can do **[4]**, **[5]**. In 1950s and 1960s, it was observed that human vision is hierarchical and different neurons identify simple features and progressively feed into more complex ones. Acquiring this knowledge, scientists came up with an idea of recreating neurons into a digital form. Although for four decades, computer vision had been considered as one of the most difficult tasks. Later on when internet came into play, and therefore, more data was available, as well as, hardware systems became stronger and stronger, computer vision started being more realistic goal to be accomplished than it had used to be in the past. In 2012, Artificial Intelligence had its breakthrough moment in computer vision tasks when a team from the University of Toronto participated in ImageNet Large Scale Visual Recognition Challenge (ILSVRC) where the teams evaluated their algorithms on a given dataset. They won by achieving an accuracy of 84% with the deep neural network called AlexNet where the best score since then hovered around 74% **[6]**. Nowadays, the amount of data that is generated around the globe is enormously large, and as a result, individuals face difficulties in handling and getting meaningful understanding in such large information. In this part is where computers come into play through neural networks which, not only can they handle more data, but also they take advantage and improve their performance by being trained on large datasets **[7]**. In this research, we will focus on image data for which deep learning performs immensely well.

2.2 Machine learning and Deep Learning

2.2.1 Machine Learning

Machine learning is a subset of Artificial Intelligence and it is the study of algorithms and statistical models with which a system can become accurate in predicting outcomes without being explicitly programmed to do so, by using historical data to train itself **[8]**, **[9]**. Machine learning can be divided into 4 parts, Supervised Learning, Unsupervised Learning, Semi-supervised Learning and Reinforcement Learning. The first approach implies a task where the dataset is labelled. A supervised learning can be further divided into two different tasks named Regression and Classification. The first implies the task where the dependent variable Y for which the model will be trained to predict, is a continuous variable. Classification refers to the task where the Y is categorical variable and each category is called class **[10]**. In this research, we will be tackling a Binary Supervised Classification task where the dataset is labelled and the independent variable comprises of two classes. There are plenty of machine learning algorithms that can be used to accurately predict outcomes, although in this research, deep learning which is a subset of machine learning will be used.

2.2.2 Deep Learning

Deep Learning describes a set of algorithms that uses logic, like a human brain, to analyse data, train themselves, and lastly, predict outcomes **[11]**. Over the last 10 years, several deep learning models have been developed which perform well in different tasks like computer vision, Natural Language Processing (NLP) and so forth. Furthermore, different techniques have been examined and proposed to improve the performance of the model when some difficulties or restrictions with the data occur, such as limited data **[12]**, **[13]**. Some of the best deep learning algorithms are described and proposed on **[14]** and each one is focusing on different tasks. These includes Convolutional Neural Networks (CNNs) for computer vision, Long Short Term Memory Networks (LSTMs) and Recurrent Neural Networks (RNNs) for speech recognition and NLP. For this research, we will be focusing on algorithms that perform immensely well on Computer Vision and more precisely on image classification tasks.

2.3 Linking Deep learning with Computer Vision

2.3.1 Convolution layers in Computer Vision

Computer Vision tasks can be tackled by different machine learning algorithms, but the performance of the model will be poor and result in an unreliable model. However, deep learning can handle these tasks by achieving significantly better performance than traditional machine learning algorithms. This was proven firstly by the ImageNet competition that we referred to above where a deep learning algorithm won it in 2012. Since then, deep learning algorithms have won it every year and traditional ML algorithms are no longer competitive in the competition. Secondly, the author in **[15]** compares different traditional machine learning algorithms to each other, in an image classification task with a dataset of 7 different classes. It is concluded that even if the traditional machine learning algorithms reached a good performance (more than 80% accuracy) Convolutional Neural Networks significantly outperformed the other models reaching 93% accuracy. Furthermore, as a human can identify simple features using the neurons in their brain, Neural Networks implement the same concept to achieve almost perfect performance in identifying and classifying images. The most effective models in computer vision are Neural Networks which include a convolution layer, and it will be further explained in methodology part. Briefly, a convolution layer consists of a kernel which hovers over an image and extracts important features from it. The extracted features are then passed into a fully connected layer and the final outcome is derived **[16]**.

2.3.2 Relevant Work in Computer Vision

There has been some literature around computer vision using Convolutional Neural Networks (CNN). A simple CNN was proposed in **[17]** where the author achieved a great accuracy minimizing the computational cost and memory usage of the model. The simple CNN was trained on MNIST dataset which consists of 60000 grayscale 24x24 images displaying digits from 0 to 9. The model comprises of 3 Convolutional Layers with activation function ReLU followed by max polling with kernel size 2x2 and stride 2. After the convolutional layers, a fully connected layer has been created followed by a dropout. Different learning rates and optimizers were used to come up with the best model training strategy. It was found that stochastic gradient descent with momentum performs better than the other strategies when the number of iterations is bigger and the best learning rate method in training and test set is multistep. Compared to the pre-existing models, the simple CNN performs slightly worse than the other models (0.66% error rate), although the computational cost and the memory usage have significantly reduced.

CNN seems to perform well in areas that available data is not much. CNN was used by the authors in **[18]** where they tried to classify images into 5 classes. The image dataset that the CNN model has been trained on, consists of lung images classified into 5 classes (Normal, Emphysema, Ground glass, Fibrosis and Micronodules). In the paper **[18]**, it has been observed that the lung images patches are more texture-like that have no distinct structures and a single convolutional layer has been used. The authors support that more convolutional layers would lead in overfitting since more parameters would have to be trained. After the convolutional layer, a max pooling layer has been included and reduces the dimensionality of the feature map. The reduced feature map is then passed into 3 fully connected layers. The CNN model has outperformed the models that it is compared with, achieving higher recall and precision in most cases.

A breakthrough moment in computer vision tasks took place in 2022 when ConvNeXt was introduced in **[19]**. The concept of Transformers, which was initially used in Natural Language Processing (NLP) tasks, was implemented in computer vision and outperformed CNN. However, the concept of the sliding window over patches of an image was still in use. The authors in **[19]** tried to fill the gap between the two eras by progressively improving a modernized ResNet-50. Many alternative approaches were implemented in micro and macro design, as well as, the usage of inverted bottleneck, a different training procedure and many other changes were used and are clearly explained on **[19]**. The model performs immensely well in different tasks in computer vision including image classification. For the subfield of the computer vision, Image Classification, the model was trained on ImageNet 1K and 22K datasets. For the former, the accuracy of the ConvNeXt-Large was 85.5% which is 0.2% less than the EffNetV2-Large which was trained with larger images. For the latter, ConvNeXt-XL outperformed any other pre-existing model reaching an 87.8% accuracy, which is 0.5% greater than EffNetV2-XL.

Reviewing the existing literature, we saw that Convolutional Layers seem to be the best approach for computer vision tasks and for the moment, ConvNeXt is the best model on this field.

2.4 Computer Vision in Banks

2.4.1 Banks and deep learning

Over the years, more and more information is available and sparse data are necessary to be used by banks. There are different parts that deep learning contributes in order to solve different problems and overcome difficulties. Deep learning is a powerful tool for banks and it is frequently used for financial forecasting, customer churn, risk assessment, business performance and fraud detection **[20]**. In this paper, we have already mentioned that we will be focusing on fraud detection to minimize the annual loss caused from fraudsters. Facial image classification into fake and real can prevent many fraudsters to become customers and increase the risk of a potential financial loss. Searching for relevant research on face recognition and Photoshop detection in facial images, the following literature has been found and is outlined below.

2.4.2 Related Work

There has been limited research around this subject, however, ConvNeXt has been used in **[2]** which specializes in computer vision and especially in image classification. Initially, different pieces of software for image editing were used to create a dataset with photoshopped images focusing on different parts of the face (whitening teeth, reshaping the head and so forth) than previous papers proposed. A dataset was used to train a Generative Adversarial Network (GAN) which generates different artificial images. The artificial dataset was used as a training set and ConvNeXt was the architecture of the model through which the artificial dataset was passed. The accuracy on the GAN dataset was approximately 62% and 10% improvement was achieved. This accuracy could have been improved by applying data augmentation techniques such as CutMix and MixUp.

To recognize face swapping photoshop between two individuals displayed in an image or two separated images is the objective of the paper **[21]**. The paper proposed a two-stream network, the first stream is a 'based face classification stream' using GoogLeNet which generally classifies images into objects, and the second stream is based on steganalysis features. Steganalysis implies the extraction of hidden information, and it has been adapted to images. Two different applications called SwapMe and FaceSwap have been used to create images included in the training set adjoined with a significant number of untouched images. Although a good accuracy was achieved in this paper, face swapping photoshop occurred with images including in the dataset. In this proposed study, photoshopped image created from face swapping application will be examined and a model will be built to detect this.

A different approach was used to identify videos and images which have been subjected to photoshopping. The author **[22]** uses DenseNet169 to accurately classify videos to fake or real. In a dataset with 1000 videos, each video was split into images (one image per frame); if at least 1% of the images was classified as fake then the whole video was classified as fake. The videos on the dataset have been created by the deepfake algorithm which consists of two algorithms called generator and discriminator. The former creates an image which is passed to the discriminator to guess if the image is fake or real. To measure the performance of the algorithm the area under the receiver operating characteristics (ROC) curve (AUC) was used and it was concluded that the DenseNet169 outperforms the rest of the algorithm that it was compared with. However, the performance of the AUC is 60.1% and the dataset was not described well. This leads to an unclear understanding of the dataset. Lastly, the author **[22]** took advantage of the weaknesses of the deepfake model to reach an accuracy which may lead to instability in time.

A different implementation on detecting subtle editing in facial images and find the local warping was used in the paper **[23]**. The aim of the paper is to detect facial manipulations and two models were presented and used. The first of the two models, global classification model, was used to detect warped faces. The second model, local warp predictor model, was used to detect where the manipulation occurred. The dataset consists of 1.295 M faces from which 185K are unmodified. 10K images were hold out for validation purposes. The test set comprises of 100 images created by a professional, half of which are beautified subtly, and the other half have had their facial expressions modified. Dilated Residual Network (DRN-C-26) has been used for the global classification model trained in different resolutions augmented and non-augmented datasets. It performed well with the lowest accuracy being 93.7%. When the test set was passed into the models, even if the accuracy decreased, it remained high, (the lowest was 90% and the highest 99.7%). However, as there are more modified images than unmodified the training set is not balanced. The validation set is balanced (equal number of modified and unmodified images) and test set consists of 100 modified images. The aforementioned issue with the dataset might lead to low precision in a completely unseen dataset.

3. Methodology (5397/4000 words)

In this part, different methods that were used to acquire the results will be analysed and explained, as well as, how the dataset was gathered and manipulated.

Fig 1: We see 4 different facial images of the dataset. The first two (from left to right) are fake and the last two are real

3.1 Data

One of the most important steps to every research is the acquisition of a good quality dataset, as well as, a sufficient number of instances. In light of this, a dataset found on Kaggle **[24]** was used to accomplish the aims of the research. The name of the dataset is Real and Fake Face and from now on, we will be using the acronym RFF to refer to this dataset. The RFF consists of 960 fake & 1081 real images and some instances are displayed on figure 1. Additionally, a different dataset was also found on Kaggle **[25]** named CelebA and 202,599 real facial images are included in it. With the aforementioned datasets, 3 approaches will be used to answer the research question. For the first one, only the balanced RFF will be used to build a reliable model using accuracy as the most important metric to measure the performance. For the second approach, the RFF dataset will be used taking 1081 real images and only 90 fake images to achieve the imbalance which approaches a realistic problem. Lastly, we will extent the RFF by concatenating it with the CelebA and it will be examined whether an extension of a dataset results in a better performance of the model. For the last approach the dataset will consist of 203680 real and 960 fake images. For cost reduction, Google Colab was only used to train the models, and therefore the concatenated dataset was passed into the local machine and resulted issues with the RAM. To overcome the aforementioned problem, the concatenated dataset was passed in batches of 2558 125x125 RGB image using the h5py library in python. Each batch comprises of 2546 real and 12 fake images. For each approach, the dataset was split into training, validation and test set using the proportion of 80:10:10 respectively.

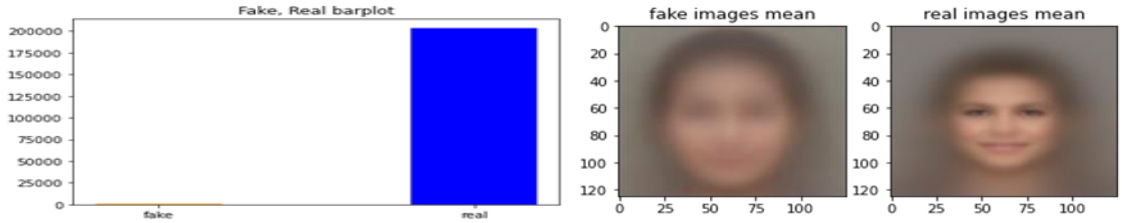

Fig 2: At the left, we see the barplot for the fake and real images. At the right, we see the facial images resulted by the calculation of the mean for the fake and real images respectively.

3.2 Preliminary Analysis

After passing the dataset into the local machine, preliminary analysis will be conducted to extract useful insights from the dataset. We initially count the number of real and fake images and a barplot is shown on figure 2. Since 203680 real and 960 fake image are found on the dataset, the ratio between the two classes is 212:1 (for every 212 real, there is 1 fake image). Each image on the dataset is a 3 dimensional matrix, the rows of the matrix are equal to the length of the image, the columns are equal to the width of the image and lastly, the third dimension is the channels of the image and is equal to 3 since it is RGB. For analysis purposes we will be flattening each image and it will be represented by a vector with 125*125*3 elements. Since each image has been converted to a vector, the dataset is a two-dimensional matrix, its rows are the number of instances (each row represents a different image) and the columns are equal to the number of elements of the vectors. For the fake and real images, we will calculate the mean, and the standard deviation of each column, and the resulted images are demonstrated on figure 2 and 3 respectively. We also calculate the absolute value of the difference in the mean of each column for the fake and real images, and the resulted picture is displayed on figure 3. We see that the face in the fake images is much more zoomed and blurry than the real images. In light of this, we will use stride on the convolutional layer which adds an x number of black pixels (R = 0, G=0, B=0) all around the perimeter of the image. On figure 3, we see a great difference in the mean at the areas of nose, mouth and eyes which indicates the areas that the photoshop occurred. Finishing with the preliminary analysis we will start off building a model for the first approach.

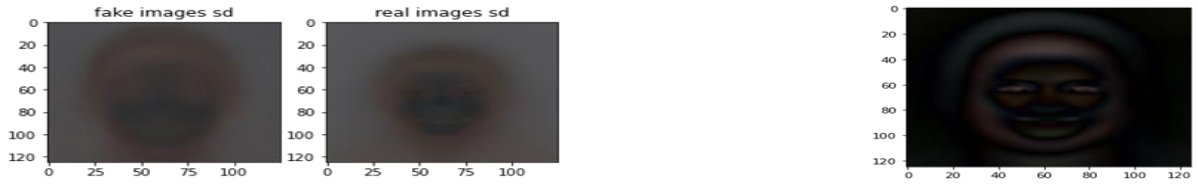

Fig 3: At the left, we see the resulted fake and real images by the calculation of the standard deviation. At the right, we see the resulted image calculated by the absolute value of the difference in "mean" images displayed on figure 2.

3.3 Convolution Layer

In order to build the model, we should introduce the most important terms that will be used. Convolutional layer is frequently used for image data and a kernel (window) with lower size than the imported image hovers over it. As we mentioned above the image is a 3 dimensional matrix and each pixel is represented by 3 numbers, one for red, one for green, and one for blue. A 2D convolution is a bunch (filter) of 3 dimensional matrices (kernel), with X rows, Y columns and 3 channels (frequently X=Y). Its elements (called weights) are random numbers from the Normal Distribution ($\mu = 0$, $\sigma = 1$) and the dot product between the kernel and a specific area of the image (initially the kernel will be placed at the upper left corner of the image) is calculated. The result is saved into a matrix called feature map and the kernel moves K pixels at the right in order to do the same procedure again. When the feature map is completed, max pooling is performed and it is a layer where a window with X rows and Y columns (not necessarily equal to the kernel of the convolution but frequently X=Y) hovers over the feature map and the maximum element is kept. Then the window is moved at the right and the same procedure takes place until the kernel scans the whole image **[16]**. It is worth mentioning that the kernel in the max-pooling layer has no weights since dot product is not required. Lastly, the activation function term should be initialized and it is a step which is either applied onto the feature map before or after max-pooling. The most well-known activation function is ReLU and its formula is shown below.

$$Relu(X) = \max(0, X)$$

X represents the elements of the feature map and if ReLU is chosen as an activation function it will simply replace any negative element on the feature map with zero. It is worth mentioning that ReLU(max-pooling(X)) = max-pooling(ReLU(X)), but if another activation function is used the activation function should be placed before max-pooling **[26]**.

3.4 Fully Connected Layer

CNN consists of convolutional layers and frequently followed by at least one fully connected layer. The extracted features acquired by the convolutional layers are flattened (Input Layer) and passed into the fully connected layer (1st Hidden layer) which comprises of number of nodes in which a random value from the normal distribution has been assigned and called bias. The extracted features are connected with each node by straight lines on which a random number from the normal distribution has been assigned and called weights. The dot product between the extracted features and the weights is calculated and the bias is added to the result. The result is passed as an input in the activation function and the image of the function will result the final value for a node. The weights and the bias is simultaneously tuned along with the convolutional layer through back propagation which will be explained in the next paragraphs. The same concept applies to the next fully connected layers (2nd hidden layer) (if exist) until we reach the output layer which will consist of a number of nodes equal to the number of classes (in our case only 2). The architecture of a convolution and fully connected layer is demonstrated in figure 4.

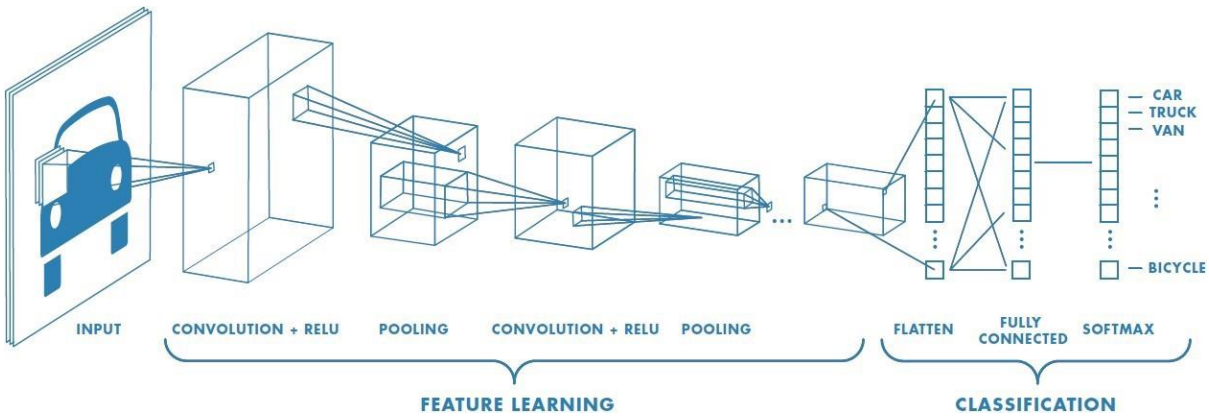

Fig 4: The architecture of Convolutional layers is displayed. We see how the kernel hovers over the image and useful features are extracted. The extracted features are then passed into the next convolutional layer until it reaches the fully connected layers. The flatten features are passed into the fully connected layer and the final prediction is acquired.

3.5 Training The CNN

To explain the concept of back propagation we firstly have to introduce the terms of Loss Function and Optimizer.

3.5.1 Loss Functions

There are a great number of loss functions and each one has a different formula, however, it is differentiable. Our purpose is to find the optimal weights for the neural network, although, it is impossible to do that due to the great number of unknowns. In light of this, the problem is converted into an optimization problem and an algorithm is used to navigate the space of possible sets of weights **[27]**. To evaluate a set of weights, we use the loss function and our goal is to minimize or maximize it which is equivalent to seek for the global minimum. The aforementioned makes the loss function extremely important for the training phase and it must be carefully chosen. Each loss function addresses different problems and in our case categorical cross entropy and focal loss are the ones that will be used and explained below.

- *Categorical cross entropy*

To introduce the categorical cross entropy we should first explain what the output layer of the neural network does. Since 2 classes are found on the dataset the activation function of the output layer is sigmoid. The domain of the sigmoid function is the set of real numbers $\mathbb{R}$ and the range is $(0,1) \subset \mathbb{R}$. The image of the sigmoid function is calculated by the formula $\sigma(x) = \frac{e^x}{e^x+1}$ and represents probabilities [28]. Therefore, if the resulted value is less than 0.5 the prediction of the model is 0, otherwise it is 1. In our case, 0 represents the fake images and 1 represents the real images. The closer to one or zero the output of the sigmoid function is, the more confident the model is, to predict real or fake respectively. Going back to the categorical cross entropy, its formula is shown below.

$$L = -\frac{1}{N}[\sum_{j=1}^{N}[t_j \log(p_j) + (1 - t_j)\log(1 - p_j)]$$

Where N is the number of instances, t_j is the ground truth class {0, 1} and p_j is the sigmoid probability for the j^{th} instance [29]. Categorical cross entropy is used for classification problems and it is considered the best loss function to use for training the model.

- *Focal loss*

Since we deal with an imbalanced dataset, the categorical cross entropy will result in some problems with its value and therefore, there is a need of using different loss function. We mentioned above that when we deal with an imbalanced dataset, a model that would predict only real is completely useless. If categorical cross entropy is used the resulted loss would be extremely low, and as a result, the weights of the deep learning model would slightly change and be evaluated as good. In general, in an imbalanced dataset, the minority class describes the one with the low number of observations (Fake). Alternatively, the majority class is the one with a large number of observations (Real). Focal loss (FL) is preferred with imbalanced datasets **[30]** and was used in **[31]**. FL was compared to binary cross entropy (BCE) and the former outperformed the later, due to the fact that, when the ground truth class is confidently predicted by the model, the FL down-weights and its value approaches zero. Although, when the model completely misclassifies an observation, the loss goes to 1. When BCE is used and the model confidently classifies an image correctly, the value of BCE has non-trivial magnitude and the total value of BCE will be relatively high. The formula of the focal loss is shown below.

$$FL(P) = -\alpha(1 - P)^{\gamma}\log(P)$$

The hyperparameters γ and α can be chosen by applying cross validation. When γ = 0 the FL is equivalent to BCE. According to **[31]** the best values for γ and α are 2 and 0.25 respectively, and therefore, these values will be used for this project.

3.5.2 Optimizers

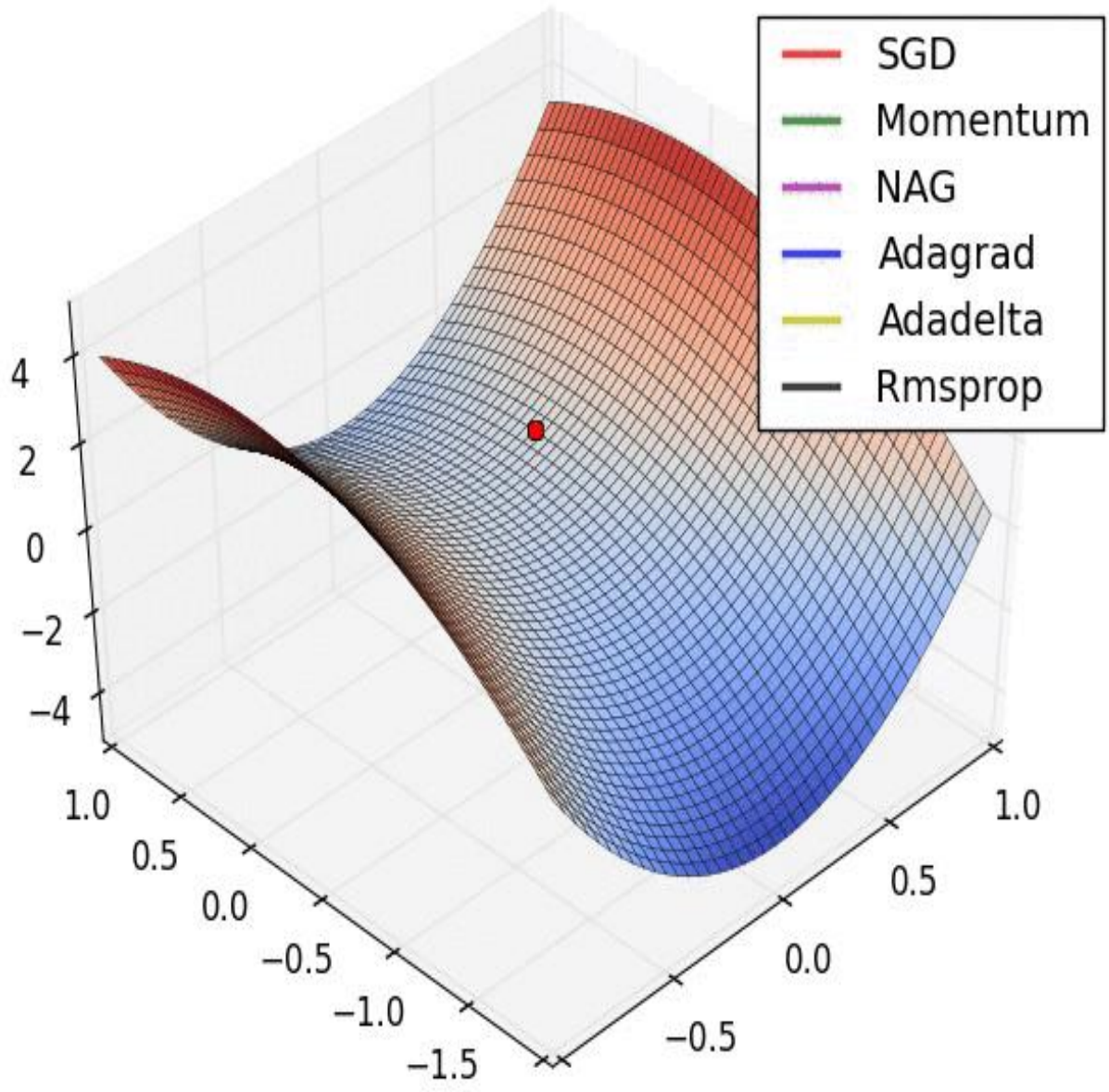

Fig 5: The surface depicted on this figure is the image of the loss function in a much simpler space. We see the importance of the optimizer and how it seeks to find the global minimum. Difference among the optimizers is observed and some of them move significantly faster than others.

In the previous paragraphs we saw how the set of weights of the model is evaluated using a loss function. However, we have not introduced or explained how we move to the next set of weights and here is where optimizer comes into play. Optimizer tunes the weights and the biases taking into consideration the resulted loss, the learning rate and potentially some more hyper parameters. The optimizer is dependent on the loss function which is dependent on the weights and the biases of the model. The formula of each optimizer calculates the slope of the tangent at a specific point on the image of the loss function to identify the way that it should move along the image of the loss function. The slope of the tangent is directly linked with the derivative (partial derivative since our space is multidimensional) and that is the reason why the loss function should be differentiable. The derivative is calculated with the chain rule which is an important term of Calculus. Each optimizer has different hyperparameters, however, all of them using the partial derivatives of the loss function and the learning rate. The learning rate along with the resulted derivative define how the weights and biases should be updated. To visualize the whole procedure is an extremely difficult task since each parameter of the neural network is an individual dimension and a model with, let's say 1M parameter, will require at least 1000001-dimensional graph to visualize only how the weights and the biases (parameters of the neural network) influence the loss function. However, on figure 5, we can see the loss function and how the optimizer seeks the global minimum in a much simpler problem. The optimizers used in this project are Stochastic Gradient Descent with momentum and Adam. Their formulas are demonstrated below.

- Stochastic Gradient Descent with Momentum

$$V_t = \beta V_{t-1} + (1-\beta)\nabla_w L(W, X, y)$$

$$W = W - aV_t$$

Where L is the loss function which is a function of the parameters of the model W (weights and biases), X are the inputs and y are the actual values. $\boldsymbol{\nabla_w}$ is $\boldsymbol{\partial L / \partial w_i}$ where w_i represents a parameter of the model (Weights and biases), α is the learning rate and β is weighting the moving average (momentum) and it is between zero and one. The closer to one the beta is, the more important the first term of the first formula is.

- Adam

$$W_t = W_{t-1} - \frac{\eta(\widehat{m_t})}{\sqrt{\widehat{V_t}} + \varepsilon}$$

Where η is the learning rate, $(\boldsymbol{\widehat{m_t}})$ and $\sqrt{\boldsymbol{\widehat{V_t}}}^2$ are calculated by the next two formulas

$$\widehat{m_t} = \frac{m_t}{1 - \beta_1^t}$$

$$\widehat{V_t} = \frac{V_t}{1 - \beta_2^t}$$

3.5.3 Back Propagation

We have introduced the terms of loss function and optimizer and in this part we will explain the back propagation with which the parameters of the model are tuned. There are different methods to train the model, but the most common one is to update the parameters after each pass of a subset of the data. The batch is a sample of the training set which is passed into the model (forward pass) and the predictions are acquired. After the forward pass, back propagation is taking place by initially calculating the gradient (derivative of the loss function), and then using it to update the parameters of the model. Then the next batch is passed into the model (forward pass) and then back propagation is happening again. When the whole dataset

is passed through the model, the first epoch is done and the model moves on with the second epoch. When the desired number of epochs has been reached, the training phase is over. Another approach that it is not as common as the one described above, is to update the model parameters after passing the entire dataset. The second approach is rationally slowing down the whole process, since the optimizer needs significantly more steps to reach the optimal loss and does many more calculations. The aforementioned will result in much higher computational cost, and therefore, in this research, the first approach will be used.

3.6 Important Metrics

In this part, the most important metrics used to measure the performance of the models on this research will be introduced and clearly explained.

3.6.1 Confusion Matrix

Confusion matrix is an important tool for classification tasks with which the correctly and incorrectly classified instances can be visualized. It is a two-dimensional matrix with x^2 elements and x (in this project x = 2) represents the number of classes in the classification task. The columns of the matrix represent the predictions of the model and the rows represent the actual values. A confusion matrix for the facial image classification task is demonstrated below.

	Predicted Fake	Predicted Real
Actual Fake	TP	FN
Actual Real	FP	TN

The four elements of the matrix displayed above are True Positives (TP) which refer to the instances that has been correctly predicted as fake, False Negatives (FN) refer to the instances that has incorrectly been classified as real, False Positives (FP) refer to the instances that has incorrectly been identified as fake, and lastly, True Negatives (TN) refer to the instances that has been correctly predicted as real. The confusion matrix is then used to calculate important predictive analytics such as, Recall, Precision, Accuracy, F1-score and so forth.

3.6.2 Accuracy – Recall – Precision – F1-Score

The aforementioned metrics measure the performance of the model in a specific task such as, how good the model is to spot real images, or how accurate the model is when it predicts fake. In light of this, accuracy measures the performance of the model in identifying the true values of the instances and counts the number of TP (fake) and TN (real) and the result is divided with the total number of observations **[32]**. It is frequently used with balanced datasets and its formula is shown below

$$Accuracy = \frac{TP + TN}{TP + FN + FP + TN}$$

Precision addresses the following question: "What proportion of fake image predictions was actually correct?" To answer the question, precision measures the TP (fake) and it is divided with the total positive (fake) predictions **[33]**; the formula is displayed below.

$$Precision = \frac{TP}{TP + FP}$$

Recall can be interpreted as "Within any observations that their actual value is real, how many has the model managed to spot?". In light of this, Recall measures the TP (fake) and the result is divided with the total number of actual true (fake) instances **[33]**. The formula of the Recall is shown below.

$$Recall = \frac{TP}{TP + FN}$$

Lastly, the F1-score is mainly used with the imbalanced data and its formula implicates the harmonic mean of the precision and recall **[32]**. Its formula is demonstrated below.

$$F1score = \frac{2 * (Precision * Recall)}{Precision + Recall}$$

By analyzing the formula of F1-score, it is easily concluded that a good score is achieved when both metrics has reached a good score. The possible values of the f1-score for the different values of precision and recall can be visualized by a 3-dimnesional graph and it is shown on figure 6.

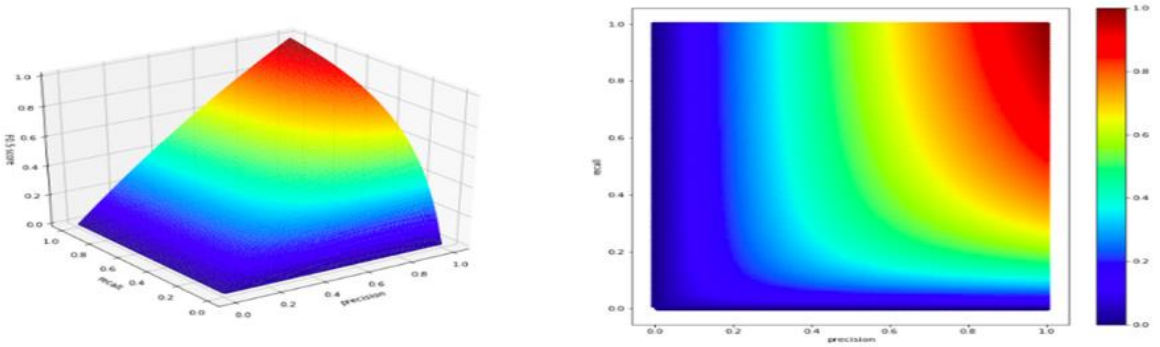

Fig. 6: We see the different values of the F1-score which are dependent on the Precision and Recall. A poor accuracy in both metrics will result a low F1-score, a bad score in Precision or Recall will result in a middle f1-score and lastly, high scores in precision and recall will result a great f1-score.

3.7 Human-In-The-Loop

Human-In-The-Loop (HITL) describes the process of combining machine and human intelligence to obtain the best results. It is frequently used with the imbalanced dataset where the model faces difficulties identifying edge cases and the cost of misclassifying an instance is relatively high **[34]**. When the model is not confident about predicting the actual value of an instance, a human intervention takes place to correct the model's prediction. This method is frequently implemented by setting a threshold. We have already mentioned that when an instance is passed into the model the final prediction is a probability. When the resulted probability is less than the threshold a human intervention takes place to guide the prediction of the model. In this project, accuracy will not be representative of the performance of the model, because of the imbalanced dataset. In light of this, Recall is considered much more important than accuracy, although, since we will be focusing on improving the Recall, will result in increasing the number of real images that have been predicted as fake (FP). HITL comes into play to correct or approve the fake model's predictions and correctly classify images into real or fake.

3.8 Data Augmentation

Data augmentation refers to the enlargement of the dataset by manipulating the existing instances. There are different techniques to augment the data, such as, position and color augmentation, as well as, CutMix and MixUp. These techniques are mainly used when the number of instances is not sufficient and they significantly improve the performance of the model. Furthermore, even if we have an adequate number of observations, data augmentation techniques can lead to an improvement of the results. In light of these, data augmentation techniques such as, MixUp and position augmentation will be used on this project. Some of the data augmented images are displayed on figure 7.

- *Position and Color augmentation*

Position and Color augmentation refers to the change of orientation and color of an image. With keras, we can include a layer in the model (Before the input layer) which will apply these transformations. When an image is passed into the model is subjected to a random rotation, scaling, flipping, cropping and padding, as well as, change of brightness, contrast and saturation. The aforementioned procedure will result in different input images in each epoch and the variety of the training instances skyrockets along with the performance of the model.

- *MixUp*

MixUp is another data augmentation technique which mixes two images from the training set. According to **[35]**, to implement this technique, we simply take one batch of the training data, and then we shuffle the rest of the batches. We take one at random, and the images of the selected batches are mixed to each other. Although, to merge the images, we need to use a formula which is shown and described below.

$$\tilde{x} = \lambda x_i + (1 - \lambda) x_j$$

$$\tilde{y} = \lambda y_i + (1 - \lambda) y_j$$

Where x_i and x_j are the images from the first and second batch respectively in the form of a vector and y_i and y_j are their labels. $\tilde{\boldsymbol{x}}$ and $\tilde{\boldsymbol{y}}$ are the new image and label resulted from the MixUp. Lastly, λ is a number from 0 to 1 and is sampled from the Beta distribution. Looking at the different cases that are generated, there are 3 different possible mixtures. The first one is when a real image is merged with a real image and the label will be real in a one-hot encoding form. The second case describes the mixture of two fake images which result in a fake image. Finally, the third case refers to the mixture of a fake and real image which will result in a one-hot-encoding-form label with values [C,D] with C+D = 1. C and D are the proportions of the fake and real images respectively, that was taken into account for creating the new instance.

Fig 7: In the first row, we see the resulted images after MixUp, and in the second row, the some of the resulted images after position and colour augmentation are displayed.

3.9 SMOTE & Undersampling

There are different techniques to deal with imbalanced data, such as, weighting the minority class, oversampling which refers to the creation of synthetic examples with a view of balancing the dataset; or undersampling which deletes instances from the majority class to bring balance between the two classes. Although, in this research, the last two techniques will be used to deal with the imbalance of the classes. SMOTE acronym stands from the Synthetic Minority Oversampling TEchnique and it works by selecting examples from the minority class that are close in the feature space. A line is drawn between the examples and a new instance is created by randomly selecting a point along that line. Since the feature space is multi-dimensional, the line is actually a hyperplane on which each point represents an instance. Specifically, a random example from the minority class is selected and then, k of the nearest neighbors are found. A random neighbor is chosen and a synthetic example is created at a random point on the hyperplane drawn between the selected instance of the minority class and the random neighbor **[36]**. A synthetic example is shown on figure 8.

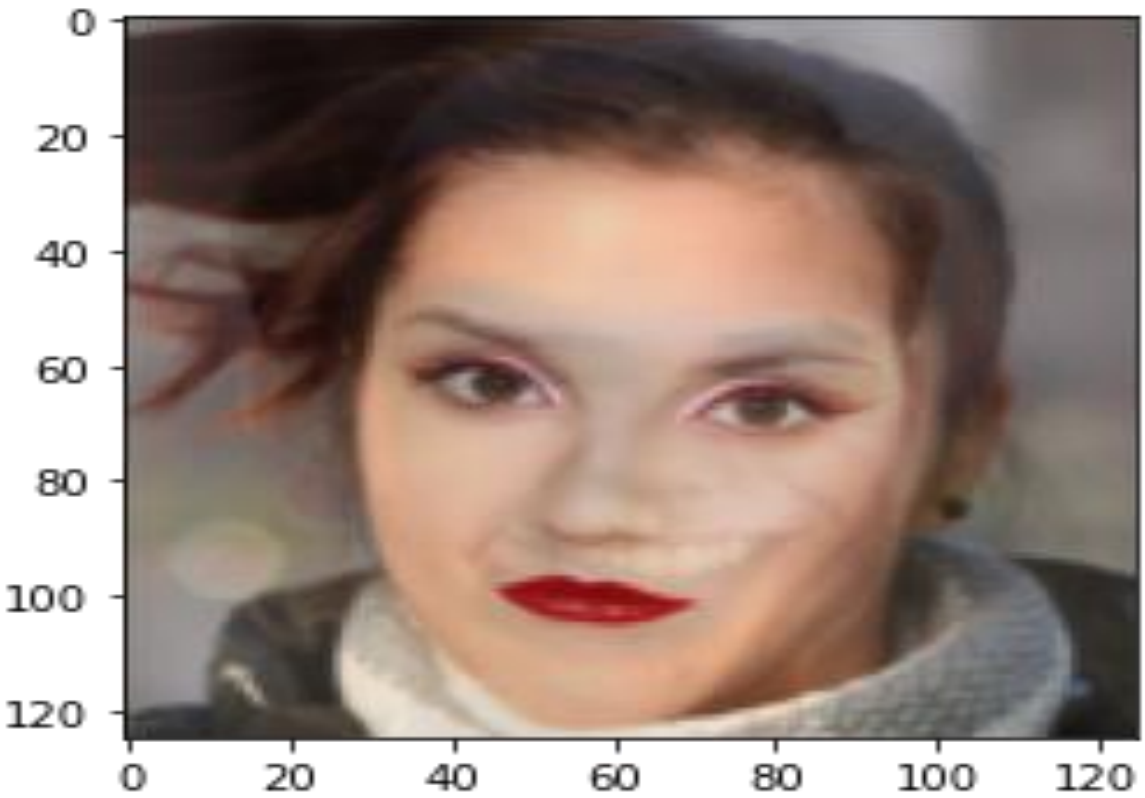

Fig 8: In this figure, a synthetic example resulted by the SMOTE method is displayed. The image above is a random point on the hyperplane between the random neighbor and the randomly selected instance of the minority class (Fake).

Alternatively, undersampling deletes random instances from the majority class until balance between the two classes is achieved, and results in losing a lot of information of the data. In light of this, undersampling is not considered as a good approach when the number of instances between the two classes is significantly large. However, according to **[36]**, when undersampling is combined with SMOTE the performance of the model is much more improved than solely using SMOTE. In this research, SMOTE will be used with and without undersampling and the best approach will be selected.

3.10 Voting Classifier & Ensemble Model

In this research, Voting Classifier and Ensemble Model will be used to further improve the results on the metrics of interest.

- *Voting Classifier*

Voting Classifier is a state-of-the-art method in which different models (frequently they perform well in different areas in the problem space) are fed into the classifier and their predictions are used to acquire the final outcome **[37]**. N number of models are trained on a dataset, the N predictions for each instance are counted (the prediction of each model will be either fake or real) and the class that has been predicted by most of the models will be the final prediction of the classifier. There are two types of voting named Hard Voting and Soft Voting. In the former, the majority predicted class is the final output of the voting classifier. For the later, we have already mentioned that the output of a neural network is probabilities, and each class is associated with one. In light of this, the average of the resulted probabilities for each class is calculated and the highest will show the prediction of the voting classifier. For instance, 5 models have been included in the voting classifier, and for the class A, 5 different probabilities are observed. Similarly, 5 probabilities are observed for the class B and the average of the 5 values for each class is calculated. The highest average will show the final prediction. In this research, hard voting will be used to acquire the final prediction.

- *Ensemble Model*

Ensemble Model is a machine learning approach to combine models in the prediction process. There are different ensemble techniques with which the ensemble model is created, such as, bagging, boosting, stacking and blending, and each one works differently, although in this research stacking will be used and refers to the creation of weak learners and a new model is trained using their predictions **[38]**. In depth, the predictions of the weak learners are passed as inputs in a new model (in this research neural network will be used) which is trained to find the optimal pattern with a view of minimizing the loss.

3.11 Transfer Learning

Transfer Learning will be one of the techniques used in this research and refers to the utilization of pre-existing knowledge acquired by a similar task. In depth, a knowledge acquired by a model trained to perform well in another task (called source domain) is transferred into the existing problem (target domain). The knowledge is actually the optimal set of weights for the source domain which will be used on the target domain. However, the pretrained model needs to be adjusted to the new task, and therefore, the parameters of the model need tuning. By training each parameter from scratch might lead to impair the optimal set of weight. In light of this, when transfer learning is used, it is common the parameters in the input and hidden layers to be kept frozen and only the last layers of the model are tuned. In this research, we are tackling an image classification task, and therefore, a pretrained model with convolutional layers will be used.

3.12 Models

As we mentioned above, three different approaches will be used according to the different datasets. The first approach implicates the balanced datasetapproach involves the imbalanced RFF with 1081 real and 90 fake images, and lastly, the third approach refers to the concatenation of the RFF with the CelebA dataset.

- *Approach One*

In the first approach, the size of each image is 225x225x3 and a pretrained ResNet-18 trained on ImageNet1K_V1 dataset with 1000 classes was used. A fully connected layer replaced the output layer of the pretrained ResNet-18. The pretrained model with the fully connected layer was the baseline, and MixUp was used to improve the results. In both cases, ADAM optimizer was used, and categorical cross entropy evaluated the sets of weights. A much simpler model is created with 6 convolutional layers with activation function ReLu and 3x3 kernel size. The first convolutional layer has 32 kernels and the remaining have 64 kernels. The size of each image has been set to 256x256. On the feature map of each convolutional layer, max pooling was performed with 2x2 kernel size. After flattening the extracted features from the convolutional layers, they were passed into two fully connected layers with 64 and 2 nodes respectively. The activation function of the first dense layer is ReLu, and for the second, Softmax has been used. The input images of the simpler model have been subjected to rotating and then passed into the model. This data augmentation technique has been included in the model, and as a result, in each epoch, a different rotation is applied to the input images. For this model, the same optimizer and loss function were used, and since the dataset is balanced, the metric to measure the performance is accuracy.

- *Approach Two*

The much simpler model using data augmentation worked well in the RFF dataset and the same model has been used for the imbalanced RFF too. Before starting the training phase, the class imbalance should be eliminated. In light of this, SMOTE has been used to bring balance between the two classes. The training set consists of 864 real and 72 fake images; after using SMOTE 864 – 72 = 792 artificial instances have been created. Categorical Cross Entropy has been used to evaluate the set of parameters, and ADAM is the optimizer for this approach. Since the validation set is imbalanced, accuracy is not a representative metric to measure the performance, and therefore, Precision and Recall have been used. The aforementioned procedure describes the baseline model. Two different approaches were used to obtain better results. For the first approach, the sampling strategy for SMOTE was set to 0.4, and therefore, 864*0.4 = 345 fake images. We also performed undersampling to reduce the number of real images setting the sampling strategy to 0.5. This value shows the ratio between the two classes (0.5 = 2:1, for 2 real, 1 fake image is found on the dataset), and as a result, since 345 fake images are found, it will result in 345*2 = 690 real images. In light of this, 1081 – 690 = 391 images were randomly discarded. For the second approach, the sampling strategy for SMOTE was set to 0.1, and therefore 86 fake images are found on the dataset from which only 14 are artificial instances. The undersampling strategy was set to 0.5 which results in 172 real images (864 – 172 = 692 real images were discarded). For the two approaches where SMOTE and Undersampling has been performed, Focal Loss has been used to evaluate the sets of parameters, since the training set is imbalanced.

- *Approach Three*

For the third approach, 125x125 image size has been used for each image due to the limited available RAM in the local machine where the image dataset was initially constructed at. As we have already mentioned, the dataset was passed using batches of 2558 images. 12 out of them are fake and the remaining are real images. The dataset was split (Stratified) into a training, validation and test set using the ratio 80:10:10. In this approach 5 models were created and then voting was conducted to get the final prediction for each instance. For the voting classifier, we have placed a threshold, and when all the models predict either fake or real (Either 5 fakes or 5 reals), the final prediction will be fake or real respectively. Otherwise, HITL is required and each image for which the models do not agree with one another, is classified by a human. The five models that constitute the voting classifier are 4 ConvNeXt trained with different hyperparameters or different over and under sampling strategies, and an ensemble model. The first ConvNeXt (model 1) was trained using SMOTE without undersampling, Stochastic Gradient Descent with momentum (learning rate = 0.05, momentum = 0.9) and Categorical Cross entropy. The second ConvNeXt (model 2) was trained using SMOTE with sampling strategy 0.4, followed by undersampling with sampling strategy 0.5. Stochastic Gradient Descent with momentum (learning rate = 0.05, momentum = 0.9) and Focal Loss were used. The third ConvNeXt (model 3) was trained using the same optimizer and loss function, however a different sampling strategy was chosen for SMOTE (0.1) and undersampling (0.5). The fourth ConvNeXt (model 4) was the same with the third, but the sets of parameters were evaluated by the Categorical Cross Entropy. Lastly, the ensemble model (model 5) was created by the 4 pretrained ConvNeXt models. The outputs of the models were passed as an input into a fully connected layer with 2 nodes (one for each class) and it was trained using the same properties as model 4. The results for each approach and model constructed for this research are demonstrated in the next section.

4. Results (2000 words)

In this part, the most important findings will be displayed and described to address the research questions. The results for the 3 main approaches used for the different datasets and described in methodology part will be demonstrated.

4.1 Approach 1 (RFF balanced)

For the first approach, it has already been mentioned that a pretrained ResNet-18 was used as a baseline model. A further improvement was achieved using the data augmentation technique called MixUp and finally, the best performance was observed using a simpler model trained on the augmented RFF dataset which consists of facial images that has been subjected to rotation. The resulted loss for the baseline model in each epoch on the training and validation set is shown in figure 9. It is observed that the model overfits after the 6th epoch where the validation loss starts increasing and simultaneously the training loss is decreasing. We see that the accuracy fluctuates along the x-axis and stabilizes after the 27th epoch. However, at the 6th epoch, the best validation accuracy is observed and stands at 68.63% on the validation and 86.33% on the training set. The performance of the model is poor and slightly better than just guessing. Therefore, improvement of the performance is required.

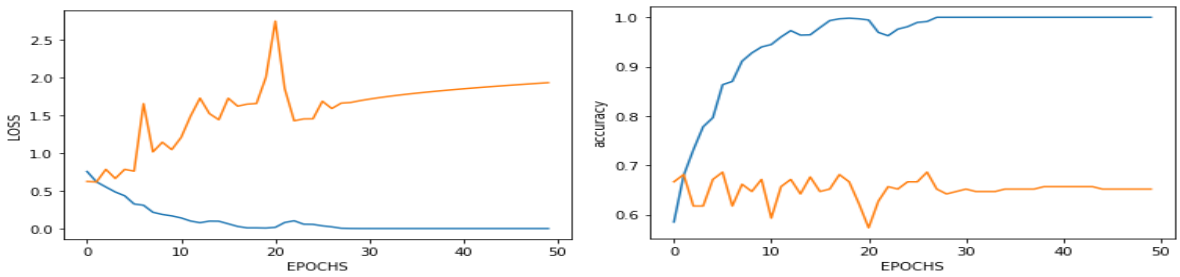

Fig 9: The resulted loss and accuracy for the training and validation set are shown on this figure. The orange graphs are the validation loss (left) and accuracy (right) for the different epochs over the training phase. The blue lines represent the training loss and accuracy.

MixUp was performed to check whether the performance of the model improves. In figure 10, we see the resulted loss for the training and validation set. The model overfits relatively early, and the loss on the validation fluctuates with an increasing pattern. The best accuracy is observed on 26th epoch where the loss is approximately 0.73. On 26th epoch, the validation accuracy stands at 73.5% which is significantly improved compared to the baseline model. The training accuracy on the 26th epoch is 95% which proofs that the model overfits. Better results were expected for both models, since a pretrained model was used, however, the training set on the source domain might be coming from completely different distribution. A pretrained model on different source domain might approach our target domain better. Probably a pretrained model on the Cats and Dogs dataset would have performed better, since the convolutional layers on it, extracts useful features, such as, eyes, edges, and so forth.

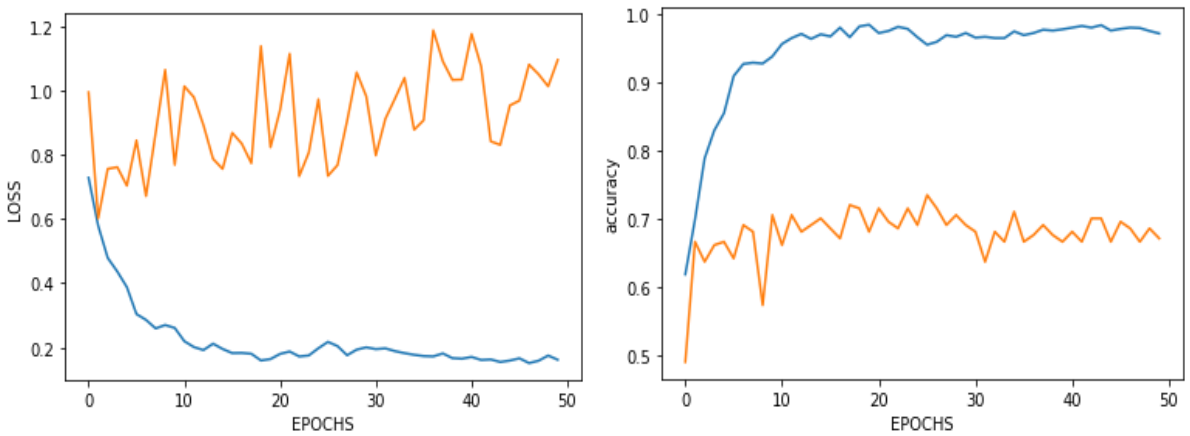

Fig 10: On the left, we see the resulted loss for the training (blue) and validation (orange) set. On the right, the resulted accuracy on the training (blue) and validation (orange) set is displayed.

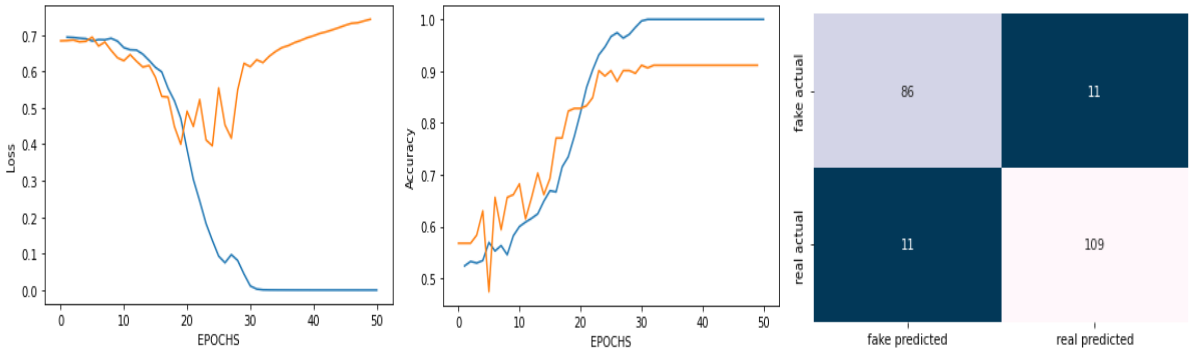

Fig 11: The resulted loss and accuracy of the simpler model on the training and validation set are displayed on this figure. On the left, we see the resulted loss for the training (blue) and validation (orange) set for the different epochs. On the middle, the accuracy on the training (blue) and validation (orange) set are depicted. Lastly, on the right we see the confusion matrix on the test set, for which the accuracy is 89.86%

A much simpler model was constructed and described on the methodology part. Data augmentation to randomly rotate each image before being passed into the input layer of the model was performed and significantly improved the performance of the model. A simpler model reduces the complexity of the model, since less parameters need tuning and relatively small dataset along with the data augmentation reduce the phenomenon of overfitting. Indeed, the accuracy was significantly better and is displayed on figure 11. It is observed that the validation accuracy has an increasing pattern and stabilizes after the 24th epoch. The best validation loss is achieved on 25th epoch and after that it significantly increases. The best accuracy on the validation set is 91.15% and observed on 31st epoch where the training accuracy is 100%. The validation loss is 0.6127, the training loss is 0.0030 and shows that the model overfits. However, the accuracy on the validation set is significantly improved, and the model will be retained and checked on the test set. The confusion matrix on the test set is also shown on figure 11 for which the accuracy is 89.86% and the table below shows the results we acquired for this approach.

Model	Training accuracy	Validation accuracy	Training loss	Validation loss	Epoch	Test Set accuracy
ResNet-18	86.33%	68.63%	0.324	0.758	6	-
ResNet-18 with MixUp	95.59%	73.53%	0.217	0.734	26	-
Simpler Model with rotation	100%	91.15%	0.003	0.613	31	89.86%

4.2 Approach 2 (RFF imbalanced)

We managed to obtain great results for the balanced RFF dataset, but as we have already mentioned, a balanced dataset is not approaching a real-world problem, and an imbalanced one will be used instead. The best model architecture which led on the best results for the first approach will be used on the second approach as well. Three different sampling strategies were used for this approach. For the first one, only SMOTE was used, for the second one, SMOTE followed by undersampling with sampling strategy 0.4 and 0.5 respectively, and lastly, for the third one, SMOTE followed by undersampling with sampling strategy 0.1 and 0.5, respectively. The model for the last two approaches was trained with the Focal Loss since the resulted dataset is imbalanced.

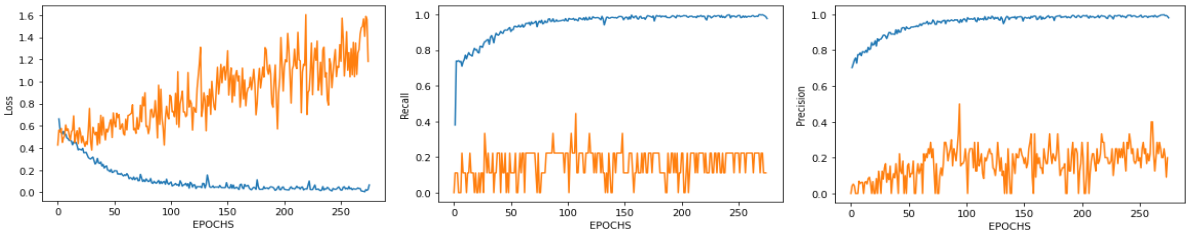

Fig 12: On this figure, the resulted loss (left), Recall (middle), and Precision (right) are displayed for the training (blue) and validation (orange) set. The sampling strategy that was used is SMOTE without Undersampling

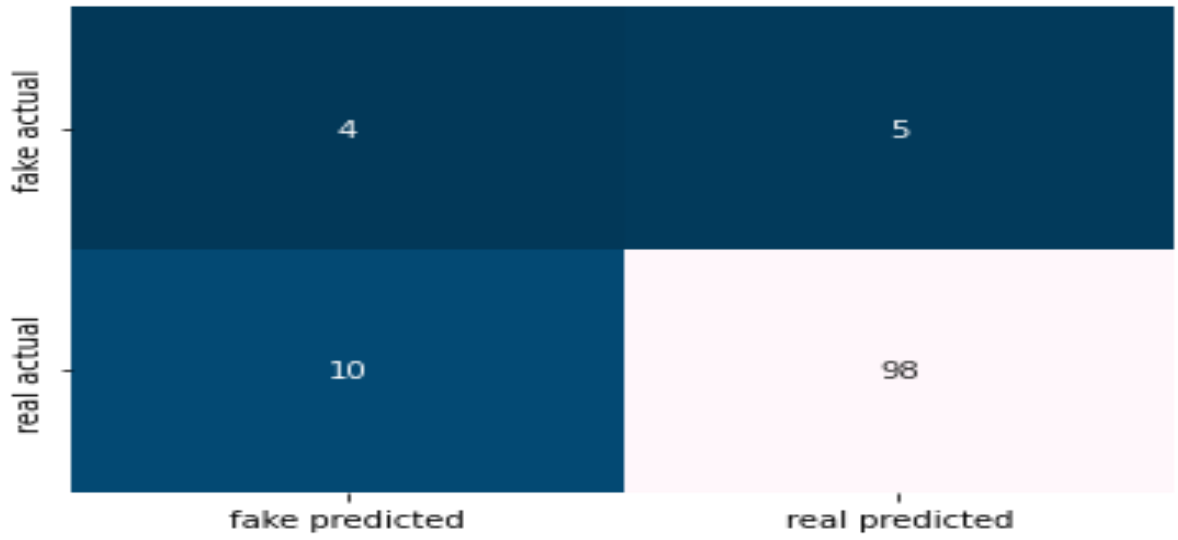

Fig 13: We see the confusion matrix on the validation set. Recall is equal to 44.4% and precision is 28.6%. Sampling strategy SMOTE without Undersampling.

In figure 12, we see the resulted loss, precision and recall for the training and validation set using the first sampling strategy (SMOTE only). After the 40th epoch the validation loss has an increasing pattern and simultaneously the training loss decreases. We have mentioned above that recall is chosen as the most important metric and in tandem, we try to maximize the precision. It is a trade-off that we must take since human intervention will take place to evaluate the fake predicted images. The best validation recall was observed on 108th and its score is 44.4%. Simultaneously, the precision is 28.6% and the confusion matrix is demonstrated on figure 13. The performance is poor, and improvement is required. We will apply different sampling strategy to check if the results improve.

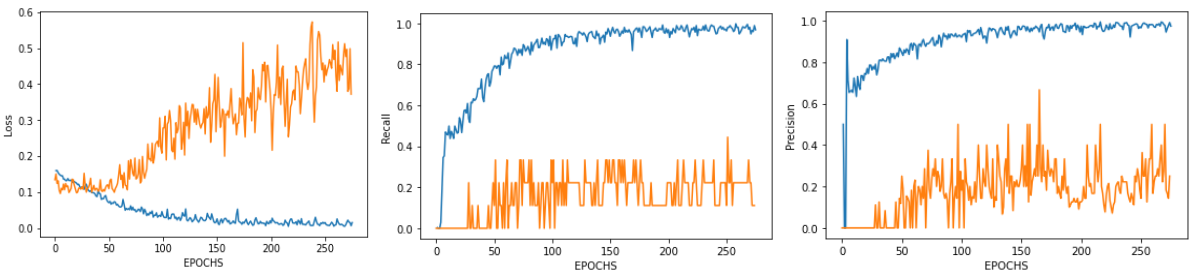

Fig 14: On this figure, the resulted loss (left), Recall (middle), and Precision (right) are demonstrated for the training (blue) and validation (orange) set. The sampling strategy is SMOTE(0.4) and Undersampling(0.5).

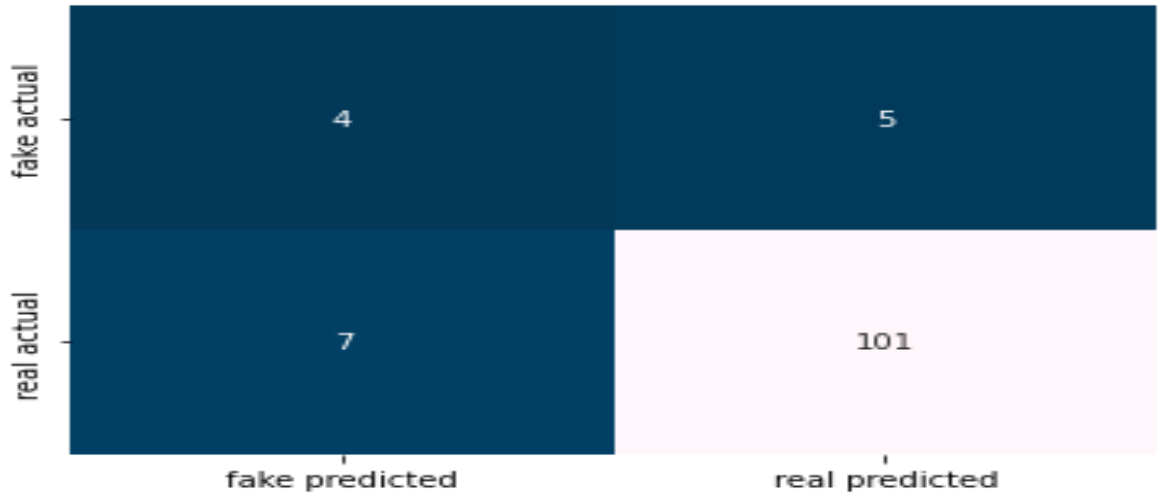

Fig 15: The resulted confusion matrix for the validation set. Recall is equal to 44.4%, and Precision is 36.4%. Sampling strategy SMOTE(0.4) with Undersampling(0.5)

We will combine SMOTE with undersampling to check whether the metrics of interest result in a better score. The sampling strategy is 0.4 for SMOTE and 0.5 for undersampling. In figure 14, the resulted loss, precision, and recall are demonstrated. It is observed that after the 55th epoch the validation loss increases, and the training loss starts converging to zero. The best recall on the validation set was achieved in epoch 252 and it is the same as in the previous approach (44.4%). However, the precision is much better (36.4%) and this model is preferred comparing it to the previous one. The confusion matrix for the validation set is displayed on figure 15. Applying different sampling strategy, we will check whether the recall can be further improved.

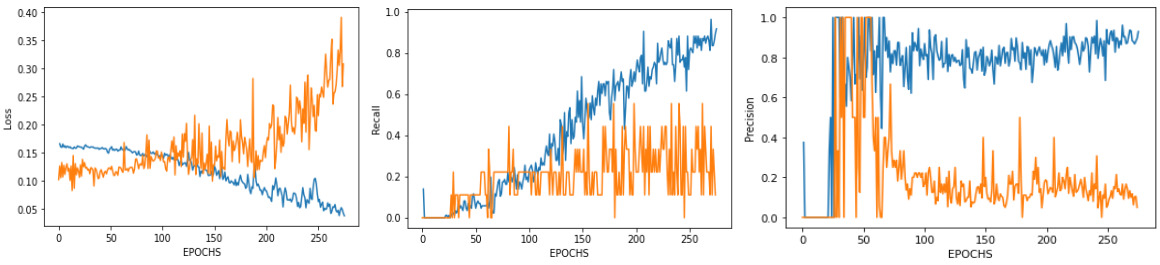

Fig 16: The resulted loss, Recall and Precision are demonstrated from the left to the right. With orange, we see the loss and the aforementioned metrics on the training set. With blue the loss and the metrics of interest on the validation set are shown. The sampling strategy is SMOTE(0.1) and Undersampling(0.5)

The third sampling approach, as we have mentioned is 0.1 for SMOTE and 0.5 for undersampling. In figure 16, we see the resulted validation loss, recall, and precision. In this approach we see that the validation loss stays close to the training loss and after 210th epoch the model starts to overfit. The best Recall is achieved in multiple epochs over the training phase, but the best precision among them is observed in 180th epoch. The validation Recall is 55.5% which is the best in comparison with the other two approaches and the Precision is 19.23%. The confusion matrices on the validation and test set for the best model are shown on figure 17, the best results for the imbalanced RFF are displayed on the next table.

Model	**SMOTE**	**SMOTE(0.4) + Undersampling (0.5)**	**SMOTE(0.1) + Undersampling(0.5)**
Training Loss	0.104	0.008	0.087
Validation Loss	0.587	0.424	0.177
Training Recall	96.88%	97.97%	56.98%
Validation Recall	44.44%	44.44%	55.56%
Training Precision	96.32%	98.83%	87.5%
Validation Precision	28.57%	36.36%	19.23%
Epoch	108	252	180
Test Recall	-	-	55.56%
Test Precision	-	-	19.23%

Despite the improvement of the Recall, the model is struggling to deal with the imbalanced RFF dataset, and we will check whether an extension of the dataset will result in a better performance.

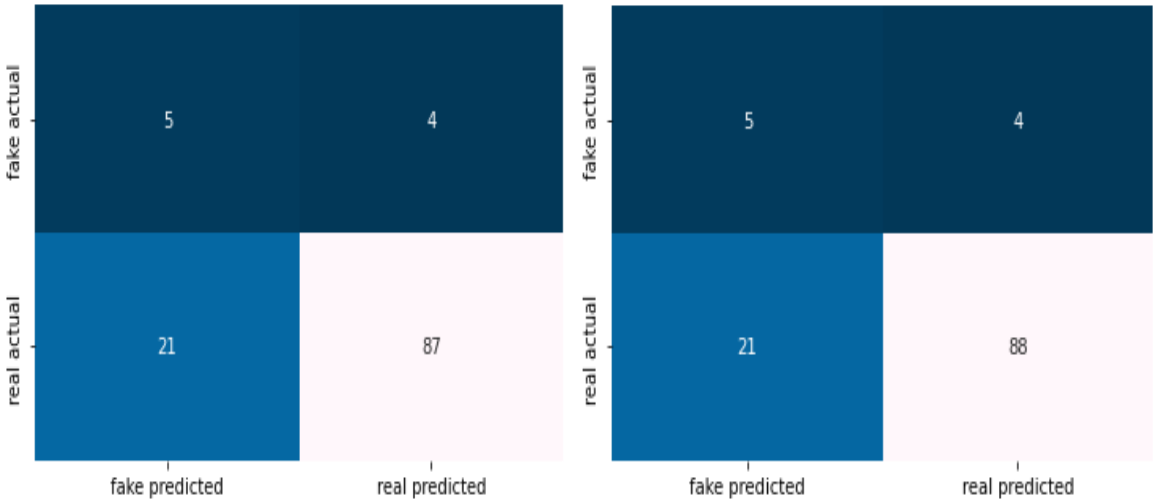

Fig 17: The confusion matrix on the left is for the validation set and on the right is for the test set. The Recall and Precision for the validation and the test set are 55.5% and 19.23%. The sampling strategy is SMOTE(0.1) and Undersampling(0.5) for which the best model was observed.

4.3 Approach 3 (RFF and CelebA)

In this approach, we will concatenate the two datasets to check whether the performance improves. We have stated above that a voting classifier will be used which consists of 5 models. The first 4 models are ConvNeXt with different hyperparameters (different loss function) and trained on a dataset with different sampling strategies to deal with the imbalance. The fifth model is an ensemble model which takes into account the predictions of the 4 ConvNeXt models. Human intervention will be required when the predictions of the 5 models in the voting classifier vary. If the 5 models agree to one another, the prediction will be kept and no human intervention is required.

For the first ConvNeXt trained on the dataset for which the imbalance was tackled by using SMOTE without undersampling, the loss function is categorical cross entropy. The model with the lowest validation loss has been kept and it stands at 0.032. The recall is 80% and the precision is 34.59%. A significantly better score was achieved in this approach comparing to the approach 2, we will try optimizing the Recall as much as possible.

For the second ConvNeXt, it was trained using focal loss and the sampling strategy for SMOTE and Undersampling was 0.4 and 0.5 respectively. The model with the lowest validation loss was kept which is equal to 0.006. The associated precision and recall are 24.49% and 75%, respectively.

For the third ConvNeXt, the sampling strategy was 0.1 for SMOTE and 0.5 for Undersampling. It was trained using focal loss and the model with the lowest value, which stands at 0.008, on the validation set was kept. The validation recall is 88.75% and the precision is 14%. A slightly better recall was observed which is the metric that we have focused on, however the precision is poor.

For the fourth ConvNeXt, the same sampling strategy was used, but the loss function with which the set of parameters of the model were evaluated, is the categorical cross entropy. Its value stands at 0.042, the Recall is equal to 81.25%, and the Precision is 25.39%.

For the ensemble model, the chosen sampling strategy for SMOTE was 0.1 and 0.5 for undersampling. Categorical cross entropy was chosen as the loss function and its lowest value over the training phase for the validation set was 0.034. The model that resulted the aforementioned loss was kept and its Precision on the validation set was 36.02%. The resulted recall stand at 83.75% and it is considered the best model comparing to the other 4. In figure 18, we see the confusion matrix in the validation set for each of the aforementioned models.

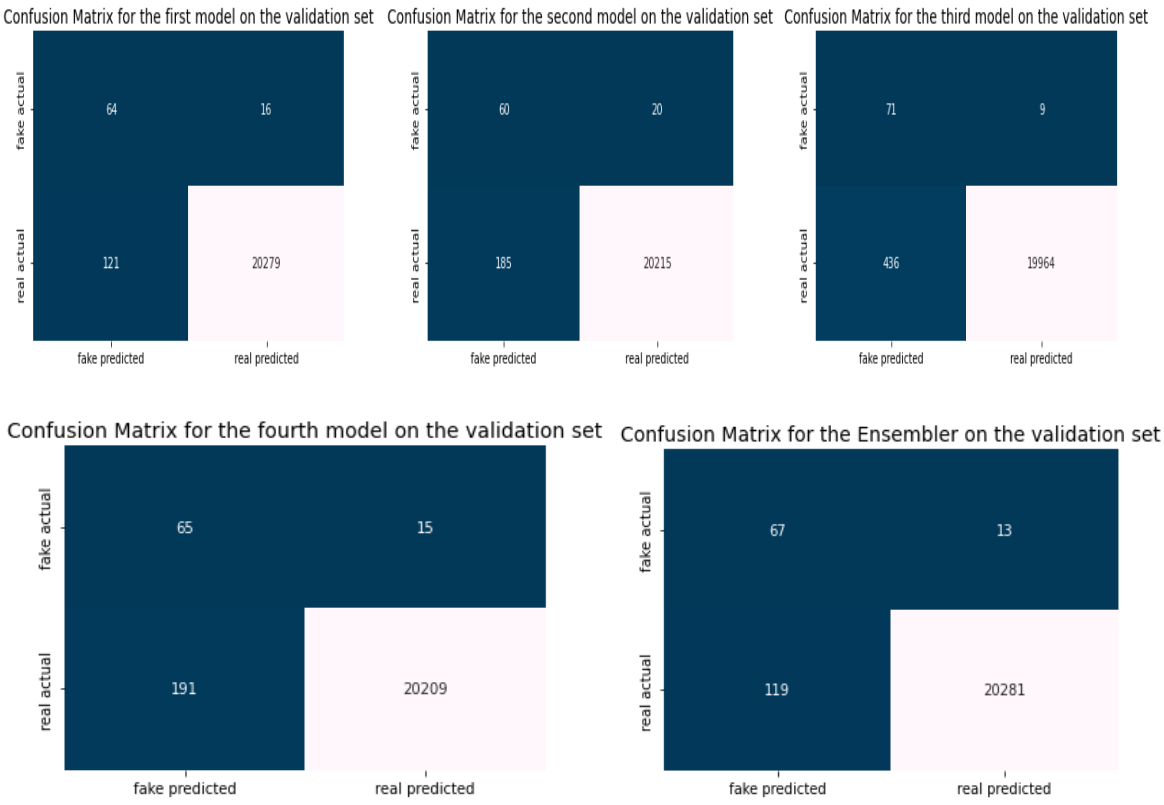

Fig 18: The resulted confusion matrices are shown on this figure for the validation set. In the first row, the confusion matrices for the first (a), second (b), and third (c) models are shown. In the second row, the confusion matrices for the fourth (d) and the ensemble model (e) are displayed. F1(a) = 48.3, F1(b) 36.9%, F1(c) = 24.2, F1(d) = 38.7% and F1(e) = 50.4%. Recall(a)=80%, Recall(b) = 75%, Recall(c) = 88.8%, Recall(d) = 81.3% and Recall(e) = 83.8%

Lastly, for the Voting Classifier, the confusion matrix for the validation set is shown on figure 19. It is worth mentioning that for validating purposes, the voting classifier was used differently than described above. When the 5 models predict real, the voting classifier will predict real. If at least, one model predicts fake, then the prediction of the voting classifier will be fake. This approach was chosen to maximize the recall since human intervention will take place to evaluate the fake predicted images. The observed recall on the validation set stands at 96.25% which is the best score in comparison with the other models. Although, the precision is equal to 13.21% which is the lowest observed. In the completely unseen test set, the model performs immensely well misclassifying only 5 images and the confusion matrix is displayed on figure 20. The third column shows the number of images that an individual needs to classify (20 of them are fake), since the models do not agree to one another (different predictions). The second column shows the real predicted images, and lastly, the first column shows the fake predicted images. Without taking into consideration the third column of the matrix, the observed precision stands at 39.28% and the Recall is equal to 91.67%. In the table below, we demonstrate the best results acquired by the 6 models.

Models	ConvNeXt1	ConvNeXt2	ConvNeXt3	ConvNeXt4	Ensemble	Voting
Validation Precision	**34.59%**	**24.49%**	**14%**	**25.39%**	**36.02%**	**13.2%**
Validation Recall	**80%**	**75%**	**88.75%**	**81.25%**	**83.75%**	**96.25%**
Validation F1-Score	**48.3%**	**36.92%**	**24.19%**	**38.69%**	**50.37%**	**23.22%**
Validation Loss	**0.032**	**0.006**	**0.008**	**0.042**	**0.03**	**-**
Test Recall	**-**	**-**	**-**	**-**	**-**	**91.67%**
Test Precision	**-**	**-**	**-**	**-**	**-**	**39.28%**
Test F1-Score	**-**	**-**	**-**	**-**	**-**	**54.99%**

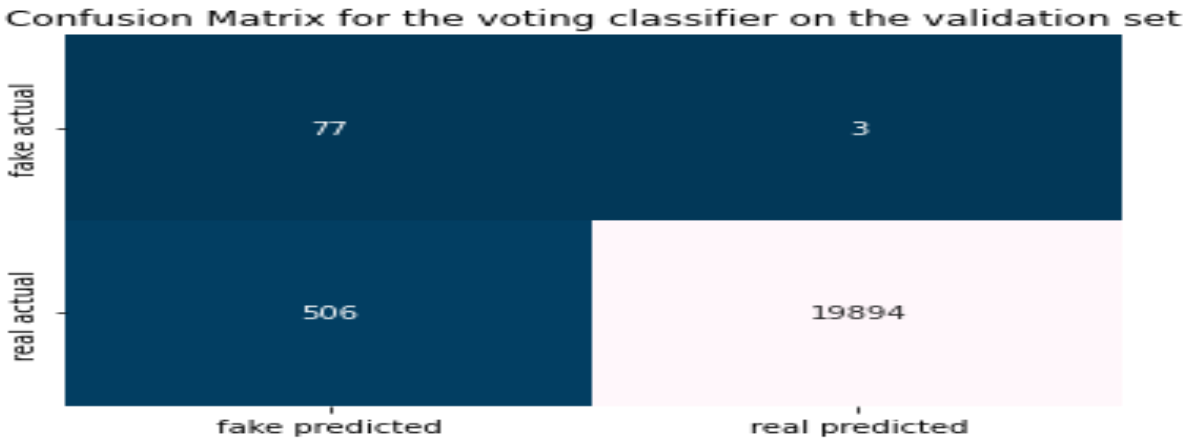

Fig. 19: The confusion matrix of the validation set for the voting classifier. Only 3 fake images are misclassified by the model and great score for the Recall was achieved (96.25%).

Fig. 20: The confusion matrix on the test set for the final model (voting classifier) is shown above. In the first column, we see the number of correctly (up) and incorrectly (down) classified fake images that will not be checked by a human, since the voting classifier was confident in predicting fake (the five models predicted fake). The second column shows the incorrectly (up) and correctly (down) classified real images that will not be checked by a human, since again the model was confident in predicting real (the five models predicted real). The third column shows the number of images that are fake (up) and real (down) and the human must classify them manually, since the voting classifier was not confident (at least one of the five models predicted fake). This approach reduces the amount of work required by a human to classify the images. Although different approach can be used for which the human must classify the resulted images on the third AND the first column. So, for the alternative approach, the total number of images that the human should check is 55+20+85+456 = 616 images which is equal to the 3% of the total images.

The best model had a recall of 92% meaning it would identify 92% of the fake images. The precision was 39%, meaning that only 39% of the records predicted as fake were actually fake. The results, we got, are outstanding, since only 5 images have been misclassified by the model. In the example above, the system identified 476 images for a person to review, where the total dataset had 20480 images. This could be used in a "human-in-the-loop" machine learning system and the voting classifier identifies 476 images of most interest, this is a massive reduction from the 20480 total images.

5. Discussion and Conclusion (1000 words)

Different methods were implemented to answer the research question and accomplish the aims of this project. We managed to achieve great accuracy on the RFF dataset by using a simple model and data augmentation techniques which was the most important action taken to increase the performance of the model. The performance on the imbalanced RFF was poor and we successfully managed to significantly improve it (mainly for the Recall) on the RFF and CelebA dataset. We conclude that the concatenation of the two datasets indeed improved the performance of the model and 91.67% recall was measured on the unseen test set. In the future, Principal Component Analysis (PCA) can be performed on the parameters of the model and obtain the kernels and nodes that explain a great proportion of the total information. This approach could potentially eliminate the phenomenon of overfitting and further increase the precision. By achieving a great score in precision and recall, human intervention would not be necessary, and the model would confidently classify facial images into fake or real.

References

[1] M. Leonhardt, "Consumers lost $56 billion to identity fraud last year—here's what to look out for," *CNBC*. https://www.cnbc.com/2021/03/23/consumers-lost-56-billion-dollars-to-identity-fraud-last-year.html (accessed Aug. 19, 2022).

[2] J. Lee, W. Lin, K. Ntalis, A. Shah, W. Tung, and M. Wulff, *Identifying Human Edited Images using a CNN*. 2021.

[3] Y. He, J. Zhou, Y. Lin, and T. Zhu, “A class imbalance-aware Relief algorithm for the classification of tumors using microarray gene expression data,” *Comput. Biol. Chem.*, vol. 80, pp. 121–127, Jun. 2019, doi: 10.1016/j.compbiolchem.2019.03.017.
[4] “What is Computer Vision? | IBM.” https://www.ibm.com/topics/computer-vision (accessed Aug. 17, 2022).
[5] R. S. Rani and P. L. Devi, “A Literature Survey on Computer Vision Towards Data Science,” vol. 8, no. 6, p. 5, 2020.
[6] “Motion Metrics | How Artificial Intelligence Revolutionized Computer Vision: A Brief History - Motion Metrics.” https://www.motionmetrics.com/how-artificial-intelligence-revolutionized-computer-vision-a-brief-history/ (accessed Aug. 17, 2022).
[7] B. Marr, “7 Amazing Examples Of Computer And Machine Vision In Practice,” *Forbes*. https://www.forbes.com/sites/bernardmarr/2019/04/08/7-amazing-examples-of-computer-and-machine-vision-in-practice/ (accessed Aug. 17, 2022).
[8] E. Burns, “What Is Machine Learning and Why Is It Important?,” *SearchEnterpriseAI*. https://www.techtarget.com/searchenterpriseai/definition/machine-learning-ML (accessed Aug. 17, 2022).
[9] “Machine Learning in Computer Vision | Full Scale,” May 08, 2019. https://fullscale.io/blog/machine-learning-computer-vision/ (accessed Aug. 17, 2022).
[10] S. Dridi, *Supervised Learning - A Systematic Literature Review*. 2021. doi: 10.13140/RG.2.2.34445.67049.
[11] A. Wolfewicz, “Deep Learning vs. Machine Learning – What’s The Difference?” https://levity.ai/blog/difference-machine-learning-deep-learning (accessed Aug. 17, 2022).
[12] J. Brownlee, *Deep Learning for Computer Vision: Image Classification, Object Detection, and Face Recognition in Python*. Machine Learning Mastery, 2019.
[13] F. Chollet, *Deep learning with Python*. Shelter Island, New York: Manning Publications Co, 2018.
[14] A. Biswal, “Top 10 Deep Learning Algorithms You Should Know in 2022,” *Simplilearn.com*. https://www.simplilearn.com/tutorials/deep-learning-tutorial/deep-learning-algorithm (accessed Aug. 17, 2022).
[15] A. Yilmaz, A. A. Demircali, S. Kocaman, and H. Uvet, “Comparison of Deep Learning and Traditional Machine Learning Techniques for Classification of Pap Smear Images.” arXiv, Sep. 11, 2020. Accessed: Aug. 18, 2022. [Online]. Available: http://arxiv.org/abs/2009.06366
[16] J. Brownlee, “How Do Convolutional Layers Work in Deep Learning Neural Networks?,” *Machine Learning Mastery*, Apr. 16, 2019. https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/ (accessed Aug. 18, 2022).
[17] T. Guo, J. Dong, H. Li, and Y. Gao, “Simple convolutional neural network on image classification,” in *2017 IEEE 2nd International Conference on Big Data Analysis (ICBDA)*, Mar. 2017, pp. 721–724. doi: 10.1109/ICBDA.2017.8078730.
[18] Q. Li, W. Cai, X. Wang, Y. Zhou, D. D. Feng, and M. Chen, “Medical image classification with convolutional neural network,” in *2014 13th International Conference on Control Automation Robotics & Vision (ICARCV)*, Dec. 2014, pp. 844–848. doi: 10.1109/ICARCV.2014.7064414.
[19] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, “A ConvNet for the 2020s.” arXiv, Mar. 02, 2022. Accessed: Aug. 18, 2022. [Online]. Available: http://arxiv.org/abs/2201.03545

[20] “Machine & Deep Learning Forecasting for Banking Industry - SDK.finance.” https://sdk.finance/machine-learning-deep-learning-forecasting-for-banking-industry/ (accessed Aug. 18, 2022).
[21] V. Kandasamy, Š. Hubálovský, and P. Trojovský, “Deep fake detection using a sparse auto encoder with a graph capsule dual graph CNN,” *PeerJ Comput. Sci.*, vol. 8, p. e953, May 2022, doi: 10.7717/peerj-cs.953.
[22] A. A. Maksutov, V. O. Morozov, A. A. Lavrenov, and A. S. Smirnov, “Methods of Deepfake Detection Based on Machine Learning,” in *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, Jan. 2020, pp. 408–411. doi: 10.1109/EIConRus49466.2020.9039057.
[23] S.-Y. Wang, O. Wang, A. Owens, R. Zhang, and A. A. Efros, “Detecting Photoshopped Faces by Scripting Photoshop.” arXiv, Sep. 05, 2019. Accessed: Aug. 18, 2022. [Online]. Available: http://arxiv.org/abs/1906.05856
[24] “Real and Fake Face Detection.” https://www.kaggle.com/datasets/ciplab/real-and-fake-face-detection (accessed Aug. 21, 2022).
[25] “img_align_celeba.” https://www.kaggle.com/datasets/yunting0123/img-align-celeba (accessed Aug. 21, 2022).
[26] K. Patel, “Convolution Neural Networks — A Beginner’s Guide [Implementing a MNIST Hand-written Digit…,” *Medium*, Oct. 18, 2020. https://towardsdatascience.com/convolution-neural-networks-a-beginners-guide-implementing-a-mnist-hand-written-digit-8aa60330d022 (accessed Aug. 21, 2022).
[27] J. Brownlee, “Loss and Loss Functions for Training Deep Learning Neural Networks,” *Machine Learning Mastery*, Jan. 27, 2019. https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/ (accessed Aug. 22, 2022).
[28] M. Saeed, “A Gentle Introduction To Sigmoid Function,” *Machine Learning Mastery*, Aug. 24, 2021. https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/ (accessed Aug. 22, 2022).
[29] K. E. Koech, “Cross-Entropy Loss Function,” *Medium*, Jul. 16, 2022. https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e (accessed Aug. 22, 2022).
[30] Y. Marathe, “How Focal Loss fixes the Class Imbalance problem in Object Detection,” *Analytics Vidhya*, Jun. 11, 2020. https://medium.com/analytics-vidhya/how-focal-loss-fixes-the-class-imbalance-problem-in-object-detection-3d2e1c4da8d7 (accessed Aug. 22, 2022).
[31] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal Loss for Dense Object Detection,” in *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017, pp. 2999–3007. doi: 10.1109/ICCV.2017.324.
[32] J. Korstanje, “The F1 score,” *Medium*, Aug. 31, 2021. https://towardsdatascience.com/the-f1-score-bec2bbc38aa6 (accessed Aug. 23, 2022).
[33] “Classification: Precision and Recall | Machine Learning | Google Developers.” https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall (accessed Aug. 23, 2022).
[34] H. in the Loop, “What is a Human in the Loop? | Humans in the Loop,” Mar. 24, 2021. https://humansintheloop.org/what-is-a-human-in-the-loop/ (accessed Aug. 23, 2022).
[35] A. Kulakov, “2 reasons to use MixUp when training your Deep Learning models,” *Medium*, Aug. 30, 2020. https://towardsdatascience.com/2-reasons-to-use-mixup-when-training-yor-deep-learning-models-58728f15c559 (accessed Aug. 23, 2022).

[36] J. Brownlee, “SMOTE for Imbalanced Classification with Python,” *Machine Learning Mastery*, Jan. 16, 2020. https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/ (accessed Aug. 23, 2022).
[37] “ML | Voting Classifier using Sklearn,” *GeeksforGeeks*, Nov. 23, 2019. https://www.geeksforgeeks.org/ml-voting-classifier-using-sklearn/ (accessed Aug. 25, 2022).
[38] M. Alhamid, “Ensemble Models,” *Medium*, Mar. 15, 2021. https://towardsdatascience.com/ensemble-models-5a62d4f4cb0c (accessed Aug. 25, 2022).

Appendix

Some of the most important code for the thesis

August 30, 2022

[1]:
```
## Creating the dataset
```

[ ]:
```
#The concept of the following piece of code is "I want to create a HDF5 file in which 80 datasets of 2558 observations are found.
#Because the fake images have been placed in one folder, we want to allocate them along each dataset, and therefore, for every 2546 real
#images we pass 12 fake. When a dataset has 2558 observations (in total), it will be saved into the hdf5 file with name/key datasetX (Where X from 1 to 80)"
f = lambda x: 0 if x.split("\\")[-1].split("_")[-1] == 'fake' else 1 #Function to extract classes from the path. fake = 0, real = 1
path = r"D:\Semester 2\Thesis\data\real_and_fake_face" #The path of the dataset. It is situated in two subdirectories (Fake and Real)
subdir = glob.glob(path + "\\*") #We get a list with the paths of the 2 subdirectories
images = {} #Initialize an empty dictionary. The images will be passed into it. length of images dictionary when a new dataset is saved into hdf5 file = 2558 and keys from 0 to 2557
#Number of images (in total) = 204640 (Fake and real), number of datasets in hdf5 file 204670/2558 = 80
#Number of fake images = 960, number of fake images in each dataset = 960/80 = 12
#Number of real images = 204640 - 960 = 203680, Number of real images in each dataset = 203680/80 = 2546
#Each dataset will have 2546 real + 12 fake = 2558 observations/images
fake = {} #Initialize an empty dictionary for the fake images only
fake_counter = 0 #The counter/key to pass each fake image into the fake dictionary
real_counter = 0 #The counter/key to pass the images into the images dictionary. Images dictionary will have fake and real images as mentioned above (2558 observations)
for i in subdir: #for loop in the path of each subdirectory
    for j in glob.glob(i + "\\*.jpg"): #for loop in the path of each image into a subdirectory
        if f(i) == 0: #f(i) returns 0 or 1 if the image is situated in the fake or real subdirectory respectively, and therefore, if fake do
```

```
            fake[fake_counter] = [np.asarray(Image.open(j).resize((125,125)),␣
 ↪"float32").flatten(), f(i)] #We read the image in a flatten array passing␣
 ↪the label simultaneously
            fake_counter += 1 #We add one to the counter
        else:#If real do
            images[real_counter] = [np.asarray(Image.open(j).resize((125,125)),␣
 ↪"float32").flatten(), f(i)] #We read the image in a flattern array passing␣
 ↪the label simultaneously
            real_counter += 1 #We add one to the counter
path1 = r"D:\Semester 2\Thesis\img_align_celeba" #the path of the second dataset
jpg = glob.glob(path1 + "\\*.jpg") #The path for each image
num = 1 #a counter which will name the dataset inside the hdf5 file
op = h5py.File(r"D:\Semester 2\Thesis\hdf5_data4.h5", "w") #We open the hdf5␣
 ↪file using mode (write)
my_list = list(range(0,12)) #This list will be used to pass the fake images in␣
 ↪the fake dictionary into the images dictionary
#At this point the real_counter = 1081 because the images on the subdirectory␣
 ↪called real has 1081 real images
#and therefore the real_counter has not reached 2546 yet. The first dataset has␣
 ↪not been created yet, and it will, inside the for loop below
for i in jpg: #For loop in the path of each image
    images[real_counter] = [np.asarray(Image.open(i).resize((125,125)),␣
 ↪"float32").flatten(), 1] #We go on adding images until the real_counter =␣
 ↪2546
    real_counter += 1 #We add one to the real_counter
    if real_counter == 2546: #When the real_counter is 2546
        for k in my_list: #For loop in my list. In the first iteration␣
 ↪my_list=[0,...,11] in the second iteration [12,...,23] and so forth
            images[real_counter + (k % 12)] = fake[k] # now the real_counter =␣
 ↪2546
#and the keys inside the images dictionary assiciated with the real images are␣
 ↪from 0 to 2545. So, the for loop will be executed 12 times
#from 0 to 11. From keys equals to 2546 to 2557 all the fake images will be␣
 ↪placed. Mod was used because when my_list = [12,...,23] or more
#We will not have concistency in keys. In each iteration we want to add the␣
 ↪numbers from 0 to 11 to the real_counter
        print(len(images)) #Print the length of the images dictionary for␣
 ↪debugging purposes
        data = pd.DataFrame.from_dict(data = images, orient = 'index',columns =␣
 ↪['image',"label"]) #We create a dataframe using the dictionary called images
        data1 = np.concatenate(data["image"][:]) #We create a flatten array␣
 ↪from the dataframe above
        data1 = data1.reshape((2558,125*125*3))/255 #We reshape it
        data3 = np.concatenate((data1,np.array(data['label']).
 ↪reshape((2558,1))),axis = 1) #We concatenate the labels to the array along␣
 ↪axis 1 (We add a column)
```

```
        ddf = pd.DataFrame(data3).sample(frac=1).reset_index(drop=True) #We␣
↪create a shuffled dataframe from the array data3
        print(ddf.shape)#We print the shape for debugging purposes
        op.create_dataset("dataset" + str(num),
                          data = ddf)#We finally create a dataset in the hdf5␣
↪file. First dataset with name/key dataset1, Second->dataset2 and so forth
        num += 1 #Since the first dataset has been created we add one to num in␣
↪order to name the second dataset (dataset2)
        real_counter = 0 #We want to reset the dictionary images and␣
↪real_counter to start creating the next dataset
        images = {} #We reset real_counter and images
        del data #We delete the datasets that we do not need anymore to save␣
↪RAM space
        del data1
        del data3
        del ddf
        my_list = [x+12 for x in my_list] #We update my_list [12,...,23], [24,..
↪.,35] and so forth
        #We update my_list because if we do not, we will get the same fake␣
↪images with keys from 0 to 11 due to the command fake[k] where k is updated␣
↪by the for loop in my_list
        print(my_list[-1]) #We print the last element of my list for debugging␣
↪purposes
op.close() #We close the file when the code is done
#It takes 42 minutes and 25 second
data = h5py.File(r"D:\Semester 2\Thesis\hdf5_data4.h5", "r") #Read the file
```

[ ]:
```
keys = list(data.keys()) #A list with the keys/names of the dataset
avg_fake = np.zeros(shape = (1,46875), dtype = "float") #Create a numpy array␣
↪with zeros and certain shape
avg_real = np.zeros(shape = (1,46875), dtype = "float") #Create a numpy array␣
↪with zeros and certain shape, they will be used to calbulate the average
for i in keys:#A for loop in the list with the keys/names
    arr_X, arr_Y = np.array(data.get(i),"float32")[:,:-1], np.array(data.
↪get(i),"int8")[:,-1] #We pass the dataset with only the independent␣
↪variables to arr_X and the lebels to arr_Y
    avg_fake = avg_fake + np.sum(arr_X[arr_Y == 0], axis = 0) #We calculate the␣
↪sum of the elements situated in the same column Σ(xij), j constant and i=1,..
↪.,n
    #We add the result to the avg_fake array
    avg_real = avg_real + np.sum(arr_X[arr_Y == 1], axis = 0) #Shame concept␣
↪for the real images
avg_fake /= 960 #Since we have the sum, we devide it with 960 which is the␣
↪total fake images
avg_real /= 203680 #Since we have the sum, we devide it with 203680 which is␣
↪the total real images
```

```
f, plot = plt.subplots(1,2)
plot[0].imshow(avg_fake.reshape(125,125,3))
plot[0].set_title("fake images mean")
plot[1].imshow(avg_real.reshape(125,125,3))
plot[1].set_title("real images mean")
```

[ ]:
```
#We calculate the difference in means and plot the resulted image
diff = abs(avg_real - avg_fake)
plt.imshow(diff.reshape(125,125,3))
```

[ ]:
```
#Split the data into training, validation and test for the 3rd approach
```

[ ]:
```
#Create train file 80% and test file 20%
op1 = h5py.File(r"D:\Semester 2\Thesis\train.h5", "w")
op2 = h5py.File(r"D:\Semester 2\Thesis\test.h5", "w")
num = 0
for i in keys:
    arr_data, arr_label = np.array(data.get(i),"float32"), np.array(data.
 ↪get(i),"int8")[:,-1]
    train, test= train_test_split(arr_data, test_size = 0.2, stratify=arr_label)
    op1.create_dataset("train" + str(num),
                           data = train)
    op2.create_dataset("test" + str(num),
                           data = test)
    num += 1
    del train
    del test
    del arr_data
    del arr_label
op1.close()
op2.close()
#validation 10% test 10%
test1 = h5py.File(r"D:\Semester 2\Thesis\test.h5", "r")
op3 = h5py.File(r"D:\Semester 2\Thesis\validation.h5", "w")
op4 = h5py.File(r"D:\Semester 2\Thesis\test_val_approach.h5", "w")
num = 0
keys_test = list(test1.keys())
for i in keys_test:
    arr_data, arr_label = np.array(test1.get(i),"float32"), np.array(test1.
 ↪get(i),"int8")[:,-1]
    val, test= train_test_split(arr_data, test_size = 0.5, stratify=arr_label)
    op3.create_dataset("validation" + str(num),
                           data = val)
    op4.create_dataset("test" + str(num),
                           data = test)
    num += 1
    del val
```

```
    del test
    del arr_data
    del arr_label
op3.close()
op4.close()
```

[ ]:
```
##Aproach 1 (RFF)
```

[ ]:
```
#Important functions
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.utils import to_categorical
from sklearn.metrics import confusion_matrix

def from_numpy_2_tensor(batch,Y,size = 4072, val = False):
    datagen = ImageDataGenerator(samplewise_center=True,␣
 ↪samplewise_std_normalization=True)
    batch = batch.reshape(size,225,225,3)
    it = datagen.flow(batch, Y, batch_size = size)
    batch, Y = it.next()
    plt.imshow(batch[0])
    plt.show()
    batch = torch.from_numpy(batch).type(torch.DoubleTensor)
    batch = batch.permute(0,3,1,2)
    plt.imshow(batch[0].permute(1,2,0).numpy())
    Y = torch.from_numpy(to_categorical(Y))
    return batch, Y

def position(ar):
    l = []
    for i in range(len(ar)):
        if ar[i,0] > ar[i,1]:
            l.append(0)
        else:
            l.append(1)
    return(np.array(l))

class metrics:
    def __init__(self, conf_mat):
        (self.TN, self.FP), (self.FN, self.TP) = conf_mat
        self.sum = np.sum(conf_mat)

    def accuracy(self):
        acc = (self.TN + self.TP) / self.sum
        return(acc)

    def precision(self):
        prec = self.TN / (self.TN + self.FN)
```

```
        return(prec)

    def recall(self):
        rec = self.TN / (self.TN + self.FP)
        return(rec)

    def F1(self):
        f1 = 2 * (self.precision() * self.recall()) / (self.precision() + self.
 ↪recall())
        return(f1)

def one_epoch(model, training_dataloader, loss_fun, optimizer):
    running_loss = 0
    last_loss = 0
    device = "cuda" if torch.cuda.is_available() else "cpu"
    cm = np.array([[0,0],[0,0]])
    lenl = len(training_dataloader)
    for i, (X,Y) in enumerate(tqdm(training_dataloader)):
        #X = torch.from_numpy(np.array(X)).type(torch.FloatTensor)
        X = normalize(X).type(torch.FloatTensor)
        X = X.to(device)
        #Y = torch.from_numpy(np.array(Y))
        Y = Y.to(device)
        optimizer.zero_grad()
       # print(X)
        out = model(X)
        #print("b")
        cm += confusion_matrix(position(Y),position(out),labels = [0,1])
        loss = loss_fun(out,Y)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % lenl == lenl - 1:
            last_loss = running_loss/(lenl)
            print("batch {} loss: {}".format(i+1, last_loss))
            running_loss = 0
    met = metrics(cm)
    print("Accuracy for the training set: {}".format(met.accuracy()))
    accur_tr.append(float(met.accuracy()))
    print("Precision for the training set: {}".format(met.precision()))
    print("Recall for the training set: {}".format(met.recall()))
    print("F1_score for the training set: {}".format(met.F1()))
    sns.heatmap(cm ,annot = True, cbar = False, cmap = 'PuBu_r', fmt = '.10g')
    plt.show()
    print(np.sum(cm))
    return last_loss
```

[ ]:
```
#Passing the dataset
f = lambda x: 0 if x.split("\\")[-1].split("_")[-1] == 'fake' else 1 #Function␣
 ↪to extract classes from the path. fake = 0, real = 1
path = "../content/drive/MyDrive/Colab Notebooks/real_and_fake_face" #The path␣
 ↪of the dataset. It is situated in two subdirectories (Fake and Real)
subdir = glob.glob(path + "/*") #We get a list with the paths of the 2␣
 ↪subdirectories
images = {} #Initialize an empty dictionary.
real_counter = 0 #The counter/key to pass the images into the images dictionary.
 ↪ Images dictionary will have fake and real images
for i in subdir: #for loop in the path of each subdirectory
    for j in glob.glob(i + "/*.jpg"): #for loop in the path of each image into␣
 ↪a subdirectory
        images[real_counter] = [np.asarray(PIL.Image.open(j).resize((225,225)),␣
 ↪"float32").flatten(), f(i)] #We read the image in a flattern array passing␣
 ↪the label simultaneously
        real_counter += 1 #We add one to the counter
data = pd.DataFrame.from_dict(data = images, orient = 'index',columns =␣
 ↪['image',"label"]).sample(frac=1).reset_index(drop=True) #We create a␣
 ↪dataframe using the dictionary called images
data1 = np.concatenate(data["image"][:]) #We create a flatten array from the␣
 ↪dataframe above
data1 = data1.reshape((2041,225*225*3))#We reshape it
data3 = np.concatenate((data1,np.array(data['label']).reshape((2041,1))),axis =␣
 ↪1) #We concatenate the labels to the array along axis 1 (We add a column)
del data
del data1
train, test1 = train_test_split(data3,test_size = 0.2)
validation, test = train_test_split(test1, test_size = 0.5)
X_t = train[:,:-1]
Y_t = train[:,-1]
X_v = validation[:,:-1]
Y_v = validation[:,-1]
X_te = test[:,:-1]
Y_te = test[:,-1]
X, Y = from_numpy_2_tensor(X_t, Y_t, 1632)
Xval, Yval = from_numpy_2_tensor(X_v, Y_v, 204, val = True)
Xtes, Ytes = from_numpy_2_tensor(X_te,Y_te, 205, val = True)
training_data = torch.utils.data.TensorDataset(X, Y)
training_dataloader = torch.utils.data.DataLoader(training_data, batch_size =␣
 ↪32)
validation_data = torch.utils.data.TensorDataset(Xval, Yval)
validation_dataloader = torch.utils.data.DataLoader(validation_data, batch_size␣
 ↪= 32)
```

[ ]:
```
#Pretrained ResNet-18 (Baseline)
```

[ ]:
```
#We freeze the parameters of the pretrained resnet for the input and hidden␣
↪layers
model = models.resnet18(weights="IMAGENET1K_V1")
k = 1
for param in model.parameters():
    if k >= 50:
        pass
    else:
        param.requires_grad = False
    k += 1
model.fc = nn.Linear(in_features = 512, out_features = 2) #The model is trained␣
↪for 1000 classes. So we change the output layer.
```

[ ]:
```
optimizer = torch.optim.Adam(model.parameters())
loss_fun = torch.nn.CrossEntropyLoss()
best_loss = 1_000_000
EPOCHS = 50
accur = []
accur_tr = []
import torchvision.transforms as transforms
normalize = transforms.Normalize(mean=[0.5, 0.5, 0.5],
                                 std=[0.5, 0.5, 0.5])
model.to(device)

avg = []
avg_val_loss = []
for epoch_number in range(EPOCHS):
    print("----------------------EPOCH " + str(epoch_number + 1)␣
↪+"----------------------")
    model.train(True)
    confusion_mat = 0
    counter = 1
    #print("-------------------------" + str(counter) + "/
↪80-------------------------")
    v = one_epoch(model, training_dataloader, loss_fun, optimizer)
    avg.append(v)
    model.train(False)
    print("Train Loss: " + str(v))
    cm = 0
    vloss = 0
    for ke, (X_V, Y_V) in enumerate(tqdm(validation_dataloader)):
        X_V = X_V.type(torch.FloatTensor)
        X_V = normalize(X_V)
        X_V = X_V.to(device)
        Y_V = Y_V.to(device)
        pred = model(X_V)
        vloss += loss_fun(pred, Y_V).item()
```

```
        cm += confusion_matrix(position(Y_V), position(pred), labels = [0,1])
        del pred
        del X_V, Y_V
    avg_val_loss.append(vloss / len(validation_dataloader))
    print(vloss/len(validation_dataloader))
    print(cm)
    metr = metrics(cm)
    print("acc " + str(metr.accuracy()))
    accur.append(float(metr.accuracy()))
    print("precision " + str(metr.precision()))
    print("f1 " + str(metr.F1()))
    print("Recall " + str(metr.recall()))
    if vloss / len(validation_dataloader) < best_loss:
        best_loss = vloss / len(validation_dataloader)
        model_path = "../content/drive/MyDrive/Colab Notebooks/" + str(model).
↪split("(")[0] + "_{}_{}".format(best_loss, epoch_number + 1)
        torch.save(model.state_dict(), model_path)
        print("MODEL SAVED")
```

[ ]:
```
## pretrained ResNet-18 with MixUP
```

[ ]:
```
import tensorflow as tf
BATCH_SIZE = 64
train_ds_one = (
    tf.data.Dataset.from_tensor_slices((X, Y))
    .shuffle(BATCH_SIZE * 100)
    .batch(BATCH_SIZE)
)
train_ds_two = (
    tf.data.Dataset.from_tensor_slices((X, Y))
    .shuffle(BATCH_SIZE * 100)
    .batch(BATCH_SIZE)
)
# Because we will be mixing up the images and their corresponding labels, we␣
↪will be
# combining two shuffled datasets from the same training data.
train_ds = tf.data.Dataset.zip((train_ds_one, train_ds_two))
```

[ ]:
```
def sample_beta_distribution(size, concentration_0=2, concentration_1=0.5):␣
↪#beta distribution
    gamma_1_sample = tf.random.gamma(shape=[size], alpha=concentration_1)
    gamma_2_sample = tf.random.gamma(shape=[size], alpha=concentration_0)
    return gamma_1_sample / (gamma_1_sample + gamma_2_sample)


def mix_up(ds_one, ds_two, alpha=0.2): #Mix up to merge two images
    # Unpack two datasets
```

```
    images_one, labels_one = ds_one
    print("a")
    images_two, labels_two = ds_two
    print("b")
    batch_size = tf.shape(images_one)[0]
    print("c")

    # Sample lambda and reshape it to do the mixup
    l = sample_beta_distribution(batch_size, alpha, alpha)
    print("d")
    x_l = tf.reshape(l, (batch_size, 1, 1, 1))
    #print(type(x_l))
    print("e")
    y_l = tf.reshape(l, (batch_size, 1))
    print(images_one)
    x_l = tf.cast(x_l,np.float64)
    # Perform mixup on both images and labels by combining a pair of images/
 ↪labels
    # (one from each dataset) into one image/label
    images = images_one * x_l + images_two * (1 - x_l)
    print("g")
    labels = labels_one * y_l + labels_two * (1 - y_l)
    print("h")
    return (images, labels)
```

[ ]:
```
AUTO = tf.data.AUTOTUNE #Constructing the dataset and ploting some images from␣
 ↪the dataset
train_ds_mu = train_ds.map(
    lambda ds_one, ds_two: mix_up(ds_one, ds_two, alpha=0.2),␣
 ↪num_parallel_calls=AUTO
)
```

[ ]:
```
#Simpler Model
```

[ ]:
```
resize_and_rescale = tf.keras.Sequential([
    layers.experimental.preprocessing.Resizing(Image_Size,Image_Size),
    layers.experimental.preprocessing.Rescaling(1.0/255)
])
data_aug = tf.keras.Sequential([
    layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical"),
    layers.experimental.preprocessing.RandomRotation(0.2),

])
```

[ ]:
```
input_shape = (Batch_Size,Image_Size, Image_Size,Channels)
n_classes = 2
```

```
model = models.Sequential([
    resize_and_rescale,
    data_aug,
    layers.Conv2D(32, (3,3), activation='relu', input_shape = input_shape),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, kernel_size = (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, kernel_size = (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),


    layers.Flatten(),
    layers.Dense(64, activation = 'relu'),
    layers.Dense(n_classes, activation= 'softmax'),
    ])

model.build(input_shape=input_shape)
```

[ ]:
```
model.compile(
    optimizer='adam',
    loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)
```

[ ]:
```
history = model.fit(
    X,Y,
    epochs=50,
    batch_size=Batch_Size,
    verbose=1,
    validation_data=(Xv,Yv),
    shuffle = True
)
```

[ ]:
```
#Approach 2 Imbalanced RFF
```

[ ]:
```
f = lambda x: 0 if x.split("\\")[-1].split("_")[-1] == 'fake' else 1 #Function␣
 ↪to extract classes from the path. fake = 0, real = 1
path = "../content/drive/MyDrive/Colab Notebooks/real_and_fake_face" #The path␣
 ↪of the dataset. It is situated in two subdirectories (Fake and Real)
subdir = glob.glob(path + "/*") #We get a list with the paths of the 2␣
 ↪subdirectories
```

```
images = {} #Initialize an empty dictionary. The images will be passed into it.
real_counter = 0 #The counter/key to pass the images into the images dictionary.
 ↪ Images dictionary will have fake and real images as mentioned above
for i in subdir: #for loop in the path of each subdirectory
    for j in glob.glob(i + "/*.jpg"): #for loop in the path of each image into␣
 ↪a subdirectory
        images[real_counter] = [np.asarray(PIL.Image.open(j).resize((256,256)),␣
 ↪"float32").flatten(), f(i)] #We read the image in a flattern array passing␣
 ↪the label simultaneously
        real_counter += 1 #We add one to the counter
        if real_counter == 1171: # To read less number of fake images
            break
data = pd.DataFrame.from_dict(data = images, orient = 'index',columns =␣
 ↪['image',"label"]).sample(frac=1).reset_index(drop=True) #We create a␣
 ↪dataframe using the dictionary called images
data1 = np.concatenate(data["image"][:]) #We create a flatten array from the␣
 ↪dataframe above
data1 = data1.reshape((1171,256*256*3))#We reshape it
data3 = np.concatenate((data1,np.array(data['label']).reshape((1171,1))),axis =␣
 ↪1) #We concatenate the labels to the array along axis 1 (We add a column)
del data1
train, test1 = train_test_split(data3,test_size = 0.2, stratify = np.
 ↪array(data['label']).reshape((1171,1)))
validation, test = train_test_split(test1, test_size = 0.5, stratify = test1[:
 ↪,-1])
del data
X_t = train[:,:-1]
Y_t = train[:,-1]
X_v = validation[:,:-1]
Y_v = validation[:,-1]
X_te = test[:,:-1]
Y_te = test[:,-1]
```

[ ]:
```
##SMOTE only
```

[ ]:
```
from imblearn.over_sampling import SMOTE
oversample = SMOTE()
X_v = X_v.reshape(X_v.shape[0],256,256,3)
X_te = X_te.reshape(X_te.shape[0],256,256,3)
X, Y = oversample.fit_resample(X_t, Y_t)
X = X.reshape(X.shape[0],256,256,3)

print(len(Y[Y==0]))
print(len(Y[Y==1]))
```

[ ]:
```
from tensorflow.keras.utils import to_categorical
Y = to_categorical(Y)
Y_v = to_categorical(Y_v)
```

[ ]:
```
model.compile(
    optimizer='adam',
    loss = "categorical_crossentropy",
    sample_weight_mode = 0,
    metrics=['accuracy',tf.keras.metrics.Precision(class_id = 0),tf.keras.
 ↪metrics.Recall(class_id = 0)]
)
```

[ ]:
```
history = model.fit(X, Y,epochs=275,
    batch_size=32,
    verbose=1,
    validation_data=(X_v, Y_v))
```

[ ]:
```
def conf_matr(precision, recall):#To get the confusion matrix from Recall and␣
 ↪Precision
    TP = int(round(9*recall))
    FN = 9-TP
    if precision == 0:
        FP = 0
        TN = 0
    else:
        FP = int(round((1-precision)*TP/precision))
        TN = 117-TP-FP-FN
    return(np.array([[TP,FN],[FP,TN]]))
```

[ ]:
```
##SMOTE(0.4) Undersampling(0.5)
```

[ ]:
```
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline
over = SMOTE(sampling_strategy=0.4)
under = RandomUnderSampler(sampling_strategy=0.5)
steps = [('o', over), ('u', under)]
pipeline = Pipeline(steps=steps)
X_v = X_v.reshape(X_v.shape[0],256,256,3)
X, Y = pipeline.fit_resample(X_t, Y_t)
X = X.reshape(X.shape[0],256,256,3)
```

[ ]:
```
import tensorflow_addons as tfa
from tensorflow_addons.losses import SigmoidFocalCrossEntropy
model.compile(
    optimizer='adam',
    loss = tfa.losses.SigmoidFocalCrossEntropy(),
    sample_weight_mode = 0,
```

```
    metrics=['accuracy',tf.keras.metrics.Precision(class_id = 0),tf.keras.
 ↪metrics.Recall(class_id = 0)]
)
```

[ ]:
```
history1 = model.fit(X, Y,epochs=275,
    batch_size=32,
    verbose=1,
    validation_data=(X_v, Y_v))
```

[ ]:
```
## SMOTE(0.1) Undersampling(0.5)
```

[ ]:
```
over = SMOTE(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
steps = [('o', over), ('u', under)]
pipeline = Pipeline(steps=steps)
X_v = X_v.reshape(X_v.shape[0],256,256,3)
X, Y = pipeline.fit_resample(X_t, Y_t)
X = X.reshape(X.shape[0],256,256,3)
```

[ ]:
```
model.compile(
    optimizer='adam',
    loss = tfa.losses.SigmoidFocalCrossEntropy(),
    sample_weight_mode = 0,
    metrics=['accuracy',tf.keras.metrics.Precision(class_id = 0),tf.keras.
 ↪metrics.Recall(class_id = 0)]
)
```

[ ]:
```
from tensorflow.keras.callbacks import EarlyStopping
early_stopping_monitor = EarlyStopping(monitor = "val_recall_6",␣
 ↪restore_best_weights = True, patience = 275)
history2 = model.fit(X, Y,epochs=275,
    batch_size=32,
    verbose=1,
    validation_data=(X_v, Y_v),
    callbacks = [early_stopping_monitor])
```

[ ]:
```
##Aproach 3
```

[ ]:
```
from torch import nn
from torch import Tensor
from typing import List
#ConvNeXT Architecture
class ConvNormAct(nn.Sequential):
    def __init__(self,
                 in_features: int,
                 out_features: int,
                 kernel_size: int,
```

```
                 norm = nn.BatchNorm2d,
                 act = nn.ReLU,
                 **kwargs):
        super().__init__(nn.Conv2d(in_features,
                                   out_features,
                                   kernel_size = kernel_size,
                                   padding = kernel_size // 2,
                                   **kwargs),
                         norm(out_features),
                         act())

class BottleNeckBlock(nn.Module):
    def __init__(self,
                 in_features: int,
                 out_features: int,
                 reduction: int = 4,
                 stride: int = 1):
        super().__init__()
        reduced_features = out_features // reduction
        self.block = nn.Sequential(ConvNormAct(in_features,
                                               reduced_features,
                                               kernel_size = 1,
                                               stride = stride,
                                               bias = False),
                                   ConvNormAct(reduced_features,
                                               reduced_features,
                                               kernel_size = 3,
                                               stride = stride,
                                               bias = False),
                                   ConvNormAct(reduced_features,
                                               out_features,
                                               kernel_size = 1,
                                               stride = stride,
                                               bias = False,
                                               act = nn.Identity))
        self.shortcut = (
            nn.Sequential(
                ConvNormAct(
                    in_features, out_features, kernel_size=1, stride=stride,␣
↪bias=False
                )
            )
            if in_features != out_features
            else nn.Identity()
        )

        self.act = nn.ReLU()
```

```
    def forward(self, x: Tensor) -> Tensor:
        res = x
        x = self.block(x)
        res = self.shortcut(res)
        x += res
        x = self.act(x)
        return x

class ConvNexStage(nn.Sequential):
    def __init__(
        self, in_features: int, out_features: int, depth: int, stride: int = 2,␣
 ↪**kwargs
    ):
        super().__init__(
            # downsample is done here
            BottleNeckBlock(in_features, out_features, stride=stride, **kwargs),
            *[
                BottleNeckBlock(out_features, out_features, **kwargs)
                for _ in range(depth - 1)
            ],
        )

class ConvNextStem(nn.Sequential):
    def __init__(self, in_features: int, out_features: int):
        super().__init__(
            ConvNormAct(
                in_features, out_features, kernel_size=7, stride=2
            ),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
        )

class ConvNextEncoder(nn.Module):
    def __init__(
        self,
        in_channels: int,
        stem_features: int,
        depths: List[int],
        widths: List[int],
    ):
        super().__init__()
        self.stem = ConvNextStem(in_channels, stem_features)

        in_out_widths = list(zip(widths, widths[1:]))

        self.stages = nn.ModuleList(
            [
```

```
                ConvNexStage(stem_features, widths[0], depths[0], stride=1),
                *[
                    ConvNexStage(in_features, out_features, depth)
                    for (in_features, out_features), depth in zip(
                        in_out_widths, depths[1:]
                    )
                ],
            ]
        )

    def forward(self, x):
        x = self.stem(x)
        for stage in self.stages:
            x = stage(x)
        return x

class ConvNextStem(nn.Sequential):
    def __init__(self, in_features: int, out_features: int):
        super().__init__(
            nn.Conv2d(in_features, out_features, kernel_size=4, stride=4),
            nn.BatchNorm2d(out_features)
        )

class BottleNeckBlock(nn.Module):
    def __init__(
        self,
        in_features: int,
        out_features: int,
        expansion: int = 4,
        stride: int = 1,
    ):
        super().__init__()
        expanded_features = out_features * expansion
        self.block = nn.Sequential(
            # narrow -> wide
            ConvNormAct(
                in_features, expanded_features, kernel_size=1, stride=stride,␣
↪bias=False
            ),
            # wide -> wide (with depth-wise)
            ConvNormAct(expanded_features, expanded_features, kernel_size=3,␣
↪bias=False, groups=in_features),
            # wide -> narrow
            ConvNormAct(expanded_features, out_features, kernel_size=1,␣
↪bias=False, act=nn.Identity),
        )
        self.shortcut = (
```

```
            nn.Sequential(
                ConvNormAct(
                    in_features, out_features, kernel_size=1, stride=stride,␣
↪bias=False
                )
            )
            if in_features != out_features
            else nn.Identity()
        )

        self.act = nn.ReLU()

    def forward(self, x: Tensor) -> Tensor:
        res = x
        x = self.block(x)
        res = self.shortcut(res)
        x += res
        x = self.act(x)
        return x

class BottleNeckBlock(nn.Module):
    def __init__(
        self,
        in_features: int,
        out_features: int,
        expansion: int = 4,
        stride: int = 1,
    ):
        super().__init__()
        expanded_features = out_features * expansion
        self.block = nn.Sequential(
            # narrow -> wide (with depth-wise and bigger kernel)
            ConvNormAct(
                in_features, in_features, kernel_size=7, stride=stride,␣
↪bias=False, groups=in_features
            ),
            # wide -> wide
            ConvNormAct(in_features, expanded_features, kernel_size=1),
            # wide -> narrow
            ConvNormAct(expanded_features, out_features, kernel_size=1,␣
↪bias=False, act=nn.Identity),
        )
        self.shortcut = (
            nn.Sequential(
                ConvNormAct(
                    in_features, out_features, kernel_size=1, stride=stride,␣
↪bias=False
```

```
                )
            )
            if in_features != out_features
            else nn.Identity()
        )

        self.act = nn.ReLU()

    def forward(self, x: Tensor) -> Tensor:
        res = x
        x = self.block(x)
        res = self.shortcut(res)
        x += res
        x = self.act(x)
        return x

class BottleNeckBlock(nn.Module):
    def __init__(
        self,
        in_features: int,
        out_features: int,
        expansion: int = 4,
        stride: int = 1,
    ):
        super().__init__()
        expanded_features = out_features * expansion
        self.block = nn.Sequential(
            # narrow -> wide (with depth-wise and bigger kernel)
            nn.Conv2d(
                in_features, in_features, kernel_size=7, stride=stride,␣
↪bias=False, groups=in_features
            ),
            # GroupNorm with num_groups=1 is the same as LayerNorm but works␣
↪for 2D data
            nn.GroupNorm(num_groups=1, num_channels=in_features),
            # wide -> wide
            nn.Conv2d(in_features, expanded_features, kernel_size=1),
            nn.GELU(),
            # wide -> narrow
            nn.Conv2d(expanded_features, out_features, kernel_size=1),
        )
        self.shortcut = (
            nn.Sequential(
                ConvNormAct(
                    in_features, out_features, kernel_size=1, stride=stride,␣
↪bias=False
                )
```

```
            )
            if in_features != out_features
            else nn.Identity()
        )


    def forward(self, x: Tensor) -> Tensor:
        res = x
        x = self.block(x)
        res = self.shortcut(res)
        x += res
        return x

class ConvNexStage(nn.Sequential):
    def __init__(
        self, in_features: int, out_features: int, depth: int, **kwargs
    ):
        super().__init__(
            # add the downsampler
            nn.Sequential(
                nn.GroupNorm(num_groups=1, num_channels=in_features),
                nn.Conv2d(in_features, out_features, kernel_size=2, stride=2)
            ),
            *[
                BottleNeckBlock(out_features, out_features, **kwargs)
                for _ in range(depth)
            ],
        )

class BottleNeckBlock(nn.Module):
    def __init__(
        self,
        in_features: int,
        out_features: int,
        expansion: int = 4,
    ):
        super().__init__()
        expanded_features = out_features * expansion
        self.block = nn.Sequential(
            # narrow -> wide (with depth-wise and bigger kernel)
            nn.Conv2d(
                in_features, in_features, kernel_size=7, padding=3, bias=False,␣
↪groups=in_features
            ),
            # GroupNorm with num_groups=1 is the same as LayerNorm but works␣
↪for 2D data
            nn.GroupNorm(num_groups=1, num_channels=in_features),
```

```
            # wide -> wide
            nn.Conv2d(in_features, expanded_features, kernel_size=1),
            nn.GELU(),
            # wide -> narrow
            nn.Conv2d(expanded_features, out_features, kernel_size=1),
        )

    def forward(self, x: Tensor) -> Tensor:
        res = x
        x = self.block(x)
        x += res
        return x

from torchvision.ops import StochasticDepth

class LayerScaler(nn.Module):
    def __init__(self, init_value: float, dimensions: int):
        super().__init__()
        self.gamma = nn.Parameter(init_value * torch.ones((dimensions)),
                                  requires_grad=True)

    def forward(self, x):
        return self.gamma[None,...,None,None] * x

class BottleNeckBlock(nn.Module):
    def __init__(
        self,
        in_features: int,
        out_features: int,
        expansion: int = 4,
        drop_p: float = .0,
        layer_scaler_init_value: float = 1e-6,
    ):
        super().__init__()
        expanded_features = out_features * expansion
        self.block = nn.Sequential(
            # narrow -> wide (with depth-wise and bigger kernel)
            nn.Conv2d(
                in_features, in_features, kernel_size=7, padding=3, bias=False,␣
↪groups=in_features
            ),
            # GroupNorm with num_groups=1 is the same as LayerNorm but works␣
↪for 2D data
            nn.GroupNorm(num_groups=1, num_channels=in_features),
            # wide -> wide
            nn.Conv2d(in_features, expanded_features, kernel_size=1),
            nn.GELU(),
```

```
            # wide -> narrow
            nn.Conv2d(expanded_features, out_features, kernel_size=1),
        )
        self.layer_scaler = LayerScaler(layer_scaler_init_value, out_features)
        self.drop_path = StochasticDepth(drop_p, mode="batch")


    def forward(self, x: Tensor) -> Tensor:
        res = x
        x = self.block(x)
        x = self.layer_scaler(x)
        x = self.drop_path(x)
        x += res
        return x

class ConvNextEncoder(nn.Module):
    def __init__(
        self,
        in_channels: int,
        stem_features: int,
        depths: List[int],
        widths: List[int],
        drop_p: float = .0,
    ):
        super().__init__()
        self.stem = ConvNextStem(in_channels, stem_features)

        in_out_widths = list(zip(widths, widths[1:]))
        # create drop paths probabilities (one for each stage)
        drop_probs = [x.item() for x in torch.linspace(0, drop_p, sum(depths))]

        self.stages = nn.ModuleList(
            [
                ConvNexStage(stem_features, widths[0], depths[0],␣
↪drop_p=drop_probs[0]),
                *[
                    ConvNexStage(in_features, out_features, depth,␣
↪drop_p=drop_p)
                    for (in_features, out_features), depth, drop_p in zip(
                        in_out_widths, depths[1:], drop_probs[1:]
                    )
                ],
            ]
        )


    def forward(self, x):
```

```
        x = self.stem(x)
        for stage in self.stages:
            x = stage(x)
        return x

class ClassificationHead(nn.Sequential):
    def __init__(self, num_channels: int, num_classes: int = 1000):
        super().__init__(
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(1),
            nn.LayerNorm(num_channels),
            nn.Linear(num_channels, num_classes),
        )


class ConvNextForImageClassification(nn.Sequential):
    def __init__(self,
                 in_channels: int,
                 stem_features: int,
                 depths: List[int],
                 widths: List[int],
                 drop_p: float = .0,
                 num_classes: int = 1000):
        super().__init__()
        self.encoder = ConvNextEncoder(in_channels, stem_features, depths,␣
↪widths, drop_p)
        self.head = ClassificationHead(widths[-1], num_classes)
```

[ ]:
```
def one_epoch(model, training_dataloader, loss_fun, optimizer):
    running_loss = 0
    last_loss = 0
    device = "cuda" if torch.cuda.is_available() else "cpu"
    cm = np.array([[0,0],[0,0]])
    lenl = len(training_dataloader)
    for i, (X,Y) in enumerate(training_dataloader):
        X = X.to(device)
        Y = Y.to(device)
        optimizer.zero_grad()
        #print("a")
        out = model(X)
        #print("b")
        cm += confusion_matrix(position(Y),position(out),labels = [0,1])
        loss = loss_fun(out,Y)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
        if i % lenl == lenl - 1:
```

```
            last_loss = running_loss/(lenl)
            print("batch {} loss: {}".format(i+1, last_loss))
            running_loss = 0
    met = metrics(cm)
    print("Accuracy for the training set: {}".format(met.accuracy()))
    print("Precision for the training set: {}".format(met.precision()))
    print("Recall for the training set: {}".format(met.recall()))
    print("F1_score for the training set: {}".format(met.F1()))
    sns.heatmap(cm ,annot = True, cbar = False, cmap = 'PuBu_r', fmt = '.10g')
    plt.show()
    print(np.sum(cm))
    return last_loss

def from_numpy_2_tensor(batch,Y,size = 4072):
    batch = batch.astype(int)
    batch = batch.reshape(size,125,125,3)
    it = datagen.flow(batch, Y, batch_size = size)
    batch, Y = it.next()
    batch = torch.from_numpy(batch)
    batch = batch.permute(0,3,1,2)
    Y = torch.from_numpy(to_categorical(Y))
    return batch, Y

datagen = ImageDataGenerator(samplewise_center=True,␣
 ↪samplewise_std_normalization=True)
```

[ ]:
```
train = h5py.File("../content/drive/MyDrive/Colab Notebooks/train.h5","r")
validation = h5py.File("../content/drive/MyDrive/Colab Notebooks/validation.
 ↪h5","r")
test = h5py.File("../content/drive/MyDrive/Colab Notebooks/test_val_approach.
 ↪h5","r")
```

[ ]:
```
#SMOTE without undersampling -> ConvNeXt 1
```

[ ]:
```
from keras.preprocessing.image import ImageDataGenerator
from imblearn.over_sampling import SMOTE
keys = list(train.keys())
oversample = SMOTE()
modelara = ConvNextForImageClassification(num_classes =2, in_channels = 3,␣
 ↪stem_features=64, depths=[3,4,6,4], widths=[256, 512, 1024, 2048])
modelara.to(device)
optimizer = torch.optim.SGD(modelara.parameters(), lr=0.05, momentum = 0.9)
loss_fun = torch.nn.CrossEntropyLoss()
#The following code is used more than once and from now on, I will replace the␣
 ↪code with the name TRAINING
best_loss = 1_000_000
EPOCHS = 10
```

```
for epoch_number in range(EPOCHS):
    print("---------------------EPOCH " + str(epoch_number)␣
 ↪+"---------------------")
    avg = 0
    modelara.train(True)
    confusion_mat = 0
    counter = 1
    for i in tqdm(keys):
        K = np.array(train.get(i)[:,:-1]*255)
        Z = np.array(train.get(i))[:,-1].astype(int)
        K, Z = oversample.fit_resample(K, Z)
        X, Y = from_numpy_2_tensor(K, Z)
        del K, Z
        training_data = torch.utils.data.TensorDataset(X, Y)
        del X, Y
        training_dataloader = DataLoader(training_data, batch_size = 32)
        del training_data
        v = one_epoch(modelara, training_dataloader, loss_fun, optimizer)
        avg += v
        counter += 1
    modelara.train(False)
    avg_loss = avg/len(keys)
    print("Train Loss: " + str(avg_loss))
    val_keys = list(validation.keys())
    avg_dl = 0
    cm = 0
    counter = 1
    for ke in tqdm(val_keys):
        K = np.array(validation.get(ke)[:,:-1]*255)
        Z = np.array(validation.get(ke))[:,-1].astype(int)
        X_V, Y_V = from_numpy_2_tensor(K, Z, 256)
        del K, Z
        vali = torch.utils.data.TensorDataset(X_V,Y_V)
        validation_dataloader = DataLoader(vali, batch_size = 32)
        del X_V, Y_V, vali
        vloss = 0
        for z, (X_V, Y_V) in enumerate(validation_dataloader):
            X_V = X_V.to(device)
            Y_V = Y_V.to(device)
            pred = modelara(X_V)
            vloss += loss_fun(pred, Y_V).item()
            cm += confusion_matrix(position(Y_V), position(pred), labels =␣
 ↪[0,1])
            del pred
            del X_V, Y_V
        avg_dl += vloss / len(validation_dataloader)
        counter += 1
```

```
    avg_val_loss = avg_dl / len(val_keys)
    print(avg_val_loss)
    print(cm)
    metr = metrics(cm)
    print("acc" + str(metr.accuracy()))
    print("precision" + str(metr.precision()))
    print("f1" + str(metr.F1()))
    print("Recall" + str(metr.recall()))
    if avg_val_loss < best_loss:
        best_loss = avg_val_loss
        model_path = "../content/*/MyDrive/Colab Notebooks/" + str(modelara).
↪split("(")[0] + "_{}_{}".format(avg_val_loss, epoch_number +␣
↪1)#ConvNextForImageClassification_0.032506764774464 97_4
        torch.save(modelara.state_dict(), model_path)
        print("MODEL SAVED")
```

[ ]:
```
#SMOTE(0.4) Undersampling (0.5) Focal Loss -> ConvNeXt 2
```

[ ]:
```
#SMOTE(0.1) Undersampling(0.5) Focal Loss -> ConvNeXt 3
```

[ ]:
```
#SMOTE(0.1) Undersampling(0.5) Categorical Cross entropy -> ConvNeXt 4
```

[ ]:
```
## Ensemble Model
```

[ ]:
```
model_1 = ConvNextForImageClassification(num_classes =2, in_channels = 3,␣
↪stem_features=64, depths=[3,4,6,4], widths=[256, 512, 1024, 2048])
model_1.load_state_dict(torch.load("../content/drive/MyDrive/Colab Notebooks/
↪ConvNextForImageClassification_0.03250676477446497_4"))
model_1.to(device)
model_2 = ConvNextForImageClassification(num_classes =2, in_channels = 3,␣
↪stem_features=64, depths=[3,4,6,4], widths=[256, 512, 1024, 2048])
model_2.load_state_dict(torch.load("../content/drive/MyDrive/Colab Notebooks/
↪ConvNextForImageClassification_0.005846363182095615_2"))
model_2.to(device)
model_3 = ConvNextForImageClassification(num_classes =2, in_channels = 3,␣
↪stem_features=64, depths=[3,4,6,4], widths=[256, 512, 1024, 2048])
model_3.load_state_dict(torch.load("../content/drive/MyDrive/Colab Notebooks/
↪ConvNextForImageClassification_0.008263818079225872_10"))
model_3.to(device)
model_4 = ConvNextForImageClassification(num_classes =2, in_channels = 3,␣
↪stem_features=64, depths=[3,4,6,4], widths=[256, 512, 1024, 2048])
model_4.load_state_dict(torch.load("../content/drive/MyDrive/Colab Notebooks/
↪ConvNextForImageClassification_0.04255600984700507_7"))
model_4.to(device)
```

[ ]:
```
class Combined_model(nn.Module):
    def __init__(self, modelA, modelB, modelC, modelD):
        super(Combined_model, self).__init__()
        self.modelA = modelA
        self.modelB = modelB
        self.modelC = modelC
        self.modelD = modelD
        self.classifier = nn.Linear(8, 2)

    def forward(self, x):
        x1 = self.modelA(x)
        x2 = self.modelB(x)
        x3 = self.modelC(x)
        x4 = self.modelD(x)
        x5 = torch.cat((x1, x2, x3, x4), dim=1)
        x5 = self.classifier(F.relu(x5))
        return x5
```

[ ]:
```
keys = list(train.keys())
datagen = ImageDataGenerator(samplewise_center=True,␣
 ↪samplewise_std_normalization=True)
over = SMOTE(sampling_strategy=0.1)
under = RandomUnderSampler(sampling_strategy=0.5)
steps = [('o', over), ('u', under)]
pipeline = Pipeline(steps=steps)
ensembler = Combined_model(model_1, model_2, model_3, model_4)
ensembler.to(device)
optimizer = torch.optim.SGD(ensembler.parameters(), lr=0.05, momentum = 0.9)
loss_fun = torch.nn.CrossEntropyLoss()
TRAINING
```

[ ]:
```
##Voting Classifier
```

[ ]:
```
#Validation

def position(ar):
    l = []
    for i in range(len(ar)):
        if ar[i,0] > ar[i,1]:
            l.append(0)
        else:
            l.append(1)
    return(np.array(l))
cm = 0
la = 0
val_keys = list(validation.keys())
for ke in tqdm(val_keys):
```

```
    K = np.array(validation.get(ke)[:,:-1]*255)
    Z = np.array(validation.get(ke))[:,-1].astype(int)
    X_V, Y_V = from_numpy_2_tensor(K, Z, 256)
    del K, Z
    vali = torch.utils.data.TensorDataset(X_V,Y_V)
    validation_dataloader = DataLoader(vali, batch_size = 32)
    del X_V, Y_V, vali
    vloss = 0
    for z, (X_V, Y_V) in enumerate(validation_dataloader):
            X_V = X_V.to(device)
            Y_V = Y_V.to(device)
            pred1 = position(ensembler(X_V))
            pred2 = position(model_2(X_V))
            pred3 = position(model_3(X_V))
            pred4 = position(model_4(X_V))
            pred5 = position(model_1(X_V))
            f_pred = pred3 + pred4 + pred1 + pred2 + pred5
            f_pred[f_pred < 5] = 0
            f_pred[f_pred == 5] = 1
            cm += confusion_matrix(position(Y_V), f_pred, labels = [0,1])
            del pred1, pred2, pred3, pred4, pred5, f_pred
            del X_V, Y_V
print(cm)
metr = metrics(cm)
print("acc" + str(metr.accuracy()))
print("precision" + str(metr.precision()))
print("f1" + str(metr.F1()))
print("Recall" + str(metr.recall()))
```

[ ]:
```
#Test
cm = 0
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(samplewise_center=True,␣
 ↪samplewise_std_normalization=True)
test_keys = list(test.keys())
for ke in tqdm(test_keys):
    K = np.array(test.get(ke)[:,:-1]*255)
    Z = np.array(test.get(ke))[:,-1].astype(int)
    X_V, Y_V = from_numpy_2_tensor(K, Z, 256)
    del K, Z
    testi = torch.utils.data.TensorDataset(X_V,Y_V)
    test_dataloader = DataLoader(testi, batch_size = 32)
    del X_V, Y_V, testi
    for z, (X_V, Y_V) in enumerate(test_dataloader):
            X_V = X_V.to(device)
            Y_V = Y_V.to(device)
            pred1 = position(ensembler(X_V))
```

```
            pred2 = position(model_2(X_V))
            pred3 = position(model_3(X_V))
            pred4 = position(model_4(X_V))
            pred5 = position(model_1(X_V))
            f_pred = pred3 + pred4 + pred2 + pred5 + pred1
            f_pred[f_pred == 0] = 0
            f_pred[f_pred == 1] = 2
            f_pred[f_pred == 2] = 2
            f_pred[f_pred == 3] = 2
            f_pred[f_pred == 5] = 1
            f_pred[f_pred == 4] = 2
            print(f_pred)
            print(position(Y_V))
            cm += confusion_matrix(position(Y_V), f_pred, labels = [0,1,2])
            print(cm)
            del pred2, pred3, pred4, pred5, f_pred, pred1
            del X_V, Y_V
print(cm)
metr = metrics(cm[:2,:2])
print("Test acc" + str(metr.accuracy()))
print("precision" + str(metr.precision()))
print("f1" + str(metr.F1()))
print("Recall" + str(metr.recall()))
```

[ ]:
```
sns.heatmap(cm[:2,:] ,annot = True, cbar = False, cmap = 'PuBu_r', fmt = '.
↪10g', xticklabels=['fake predicted', "real predicted", "human␣
↪intervention"], yticklabels=['fake actual','real actual'])
plt.title('Confusion Matrix for the voting classifier on the test set',␣
↪fontsize = 14)
plt.show()
```