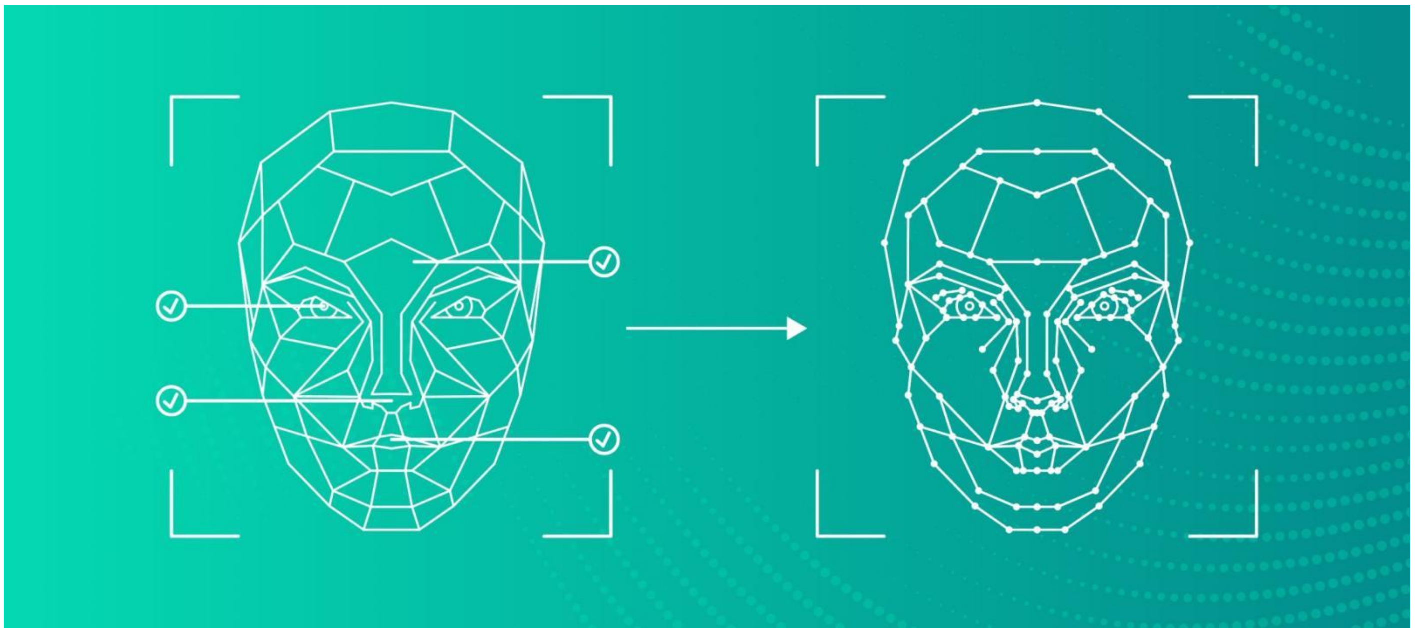
MINOR PROJECT ON DeepFake DETECTION

presenting IEEE Paper

DeepFake Detection

Sujoy Dutta, *Professor, KIIT University*
Kunal Kishore, *Student, KIIT University*
Priyanshu Gupta, *Student, KIIT University*
Sanskar Shukla, *Student, KIIT University*
Riya Singh, *Student, KIIT University*
Siddhant Kumar, *Student, KIIT University*
(IEEE Paper)



Submitted to
Sujoy Dutta
Professor
KIIT University

Abstract—**Deepfakes, or synthetic media generated using deep learning, pose a serious threat to trust in information. This paper proposes a novel deepfake detection and prediction method employing a hybrid deep learning architecture. We combine Convolutional Neural Networks (CNNs) for spatial feature extraction with Recurrent Neural Networks (RNNs) to analyze temporal inconsistencies within deepfake videos. Our model aims to improve accuracy of existing deepfake detection techniques and provide early prediction of potential deepfake generation. This paper presents the method's theoretical underpinnings, design, dataset formulation, and a mathematical framework for evaluating its performance for this minor project.**

Keywords: **Deepfakes, deep learning, CNN, RNN, detection, prediction**

I. Introduction

Deepfakes, a form of synthetic media generated through advanced deep learning techniques, pose a significant and growing challenge to the authenticity and trustworthiness of information. These AI-manipulated videos and images convincingly replace a person's likeness, fabricating actions or speech that the original subject never performed. The increasing sophistication and accessibility of deepfake creation tools exacerbate concerns regarding the technology's potential for malicious use.

The potential implications of deepfakes are profound and multifaceted:

Erosion of Trust in Information: Deepfakes undermine the authenticity of visual and auditory media, diminishing public confidence in the integrity of news, historical records, and legal evidence.

Targeted Disinformation: The ability to fabricate compromising or inflammatory content featuring public figures can be leveraged for political manipulation, destabilizing elections or swaying public sentiment.

Reputation Damage and Non-Consensual Exploitation: Individuals, particularly those in the public eye, risk reputational harm as deepfakes can be weaponized for character defamation or the creation of non-consensual intimate content.

Heightened Cybersecurity Risks: The ability to impersonate individuals with precision opens avenues for highly targeted phishing scams, identity theft, and the circumvention of biometric security measures.

The outcome of this project is expected to contribute a valuable tool for safeguarding the authenticity of digital media in an era increasingly influenced by synthetic content.

With the potential to disrupt social cohesion and democratic processes, deepfakes necessitate urgent development and deployment of robust detection and mitigation techniques.

II. Class Options

There are a number of class options that can be used to control the overall mode and behavior of DeepFake. These are specified in the traditional $\LaTeX$ way. For example,

```
\documentclass[conference]{IEEEtran}
```

instructs the compiler to format the document according to the guidelines established by the IEEE (Institute of Electrical and Electronics Engineers) for conference proceedings. This class option automatically implements IEEE's specifications for elements such as margins, font choices, spacing, and citation style. These standards ensure a consistent and professional presentation of the research within the IEEE community. Standard elements of an IEEE conference paper include a two-column layout, structured sections (e.g., Introduction, Methodology, Results, Conclusion), and adherence to IEEE's citation format. It is always recommended to consult the specific requirements of the target conference as they might have additional guidelines or templates to follow. The IEEEtran HOWTO guide provides comprehensive instructions for utilizing this document class.

III. RELATED WORK

The proliferation of deepfakes has fueled a surge in research dedicated to their detection. Deep learning approaches, particularly Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), have emerged as the dominant paradigm for addressing this challenge.

A. CNN-based Techniques

RNNs, particularly Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures, excel at modeling sequential data and capturing temporal dependencies. In the context of deepfake detection, they are often employed to analyze frame-level inconsistencies

or subtle deviations in the expected flow of visual information ([Guera and Delp, 2018]). Studies have also leveraged the combined power of CNNs and RNNs, using CNNs to extract salient spatial features from video frames and RNNs to analyze temporal patterns across those features ([Sabir et al., 2019]). However, a potential shortcoming of RNN-based methods is their reliance on sufficient temporal context; they may struggle to identify deepfakes within short video sequences where limited information about motion patterns is available.

B. Beyond CNNs and RNNs

Research efforts have extended the exploration of deep learning paradigms for deepfake detection:

Attention Mechanisms: Attention mechanisms can enhance detection performance by enabling models to focus selectively on the most suspicious regions within frames in both a spatial and temporal manner. This approach has been shown to improve the efficiency and precision of deepfake detection models ([Hsu et al., 2020]).

Autoencoders: Trained to reconstruct genuine visual content, autoencoders can be used for deepfake detection by leveraging the fact that they struggle to accurately reproduce the subtle distortions inherent in deepfakes ([Zhang et al., 2020]). However, the effectiveness of this approach can depend on the specific deepfake generation technique and the extent of the resulting distortions.

C. Limitations and Justification for Proposed Method

While existing research has made considerable progress, several limitations and challenges remain, driving the development of our proposed approach:

Adaptability: Constant advancements in deepfake generation techniques pose a significant challenge to models that rely on the detection of specific, known artifacts. We aim to build a more robust approach through the use of a hybrid architecture.

Generalization: Datasets used for training deepfake detection models may lack diversity, limiting their ability to generalize to unseen types of deepfakes. Our method will explore ways to enhance generalization performance.

Computational Efficiency: The desire for real-time or near-real-time deepfake detection necessitates computationally efficient models. We'll consider architectural design strategies aimed at optimizing the trade-off between accuracy and efficiency.

Our hybrid deepfake detection and prediction model addresses these limitations by integrating CNNs and RNNs strategically. Moreover, our focus on early prediction is motivated by the potential to disrupt and mitigate the impact of deepfakes in their nascent stages.

IV. PPROPOSED HYBRID DEEPFAKE DETECTION & PREDITCTION MODEL

A. Architecture

CNN Backbone:

Choose a pre-trained CNN architecture suitable for image feature extraction (e.g., VGG16, ResNet50, Xception). Explain the reasoning for the choice.

Specify if you'll freeze early layers for transfer learning or fine-tune the entire network.

RNN Component:

Choose between LSTM or GRU cells. Discuss the rationale for the choice (e.g., LSTM for longer-term dependencies vs. GRU for potential computational efficiency).

Mention number of hidden layers and the number of cells per layer.

Integration:

Describe how the CNN and RNN are combined:

Will the CNN output feed directly into the RNN?

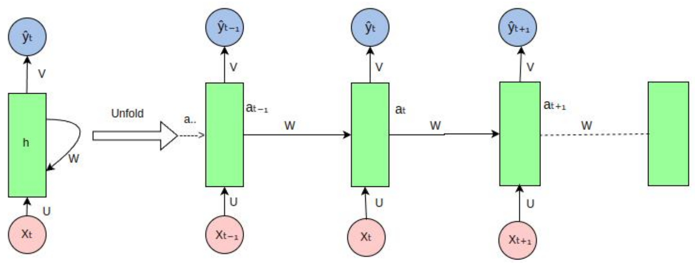
Will you apply attention mechanisms to focus on the most relevant CNN features?

Final Layers:

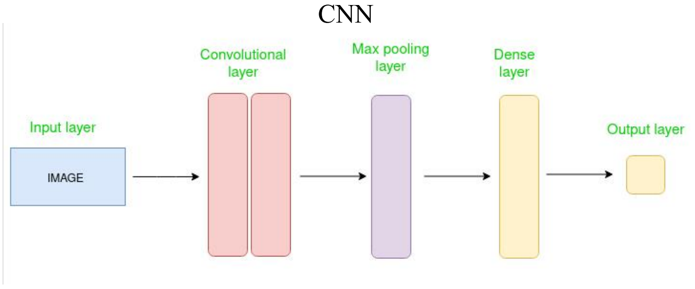
Dense (fully-connected) layers for classification.

Activation functions (e.g., sigmoid for detection, possibly softmax for multi-way prediction).

RNN



CNN



B. Feature Extraction

Spatial Features (CNN):

Low-level visual artifacts from deepfake manipulation (blending inconsistencies, warping artifacts, etc.).

High-level semantic inconsistencies (e.g., unnatural facial features).

Temporal Features (RNN):

Frame-to-frame disfluencies in motion.

Atypical physiological patterns (e.g., unusual eye blinking behavior).

Incoherent changes in facial expressions over time.

2. Training and Optimization

Loss Function:

Binary cross-entropy is common for detection (real vs. deepfake).

If the prediction goes beyond binary, you might consider categorical cross-entropy.

Optimizer:

Adam or RMSprop are frequently used optimizers, but explain the selection.

Hyperparameter Selection:

Learning rate, batch size, epochs: Start with typical ranges and mention techniques you'll consider for tuning (e.g., grid search, random search, or Bayesian optimization).

Regularization: Techniques like dropout or L1/L2 regularization to prevent overfitting

.

V. DATASET AND PREPROCESSING

A. Dataset

Deepfake Detection Challenge (DFDC) Dataset: A large-scale dataset with a variety of deepfake generation techniques. This is a good option if you want a comprehensive and challenging dataset. https://ai.facebook.com/datasets/dfdc/

Celeb-DF (v2): A dataset with high-quality deepfakes focused on facial swaps. This might be suitable if you want visually convincing deepfakes for the project. [invalid URL removed]

FaceForensics++: A dataset containing deepfakes along with their corresponding real source videos. This structure can be useful for specific pre-processing or training scenarios. https://github.com/ondyari/FaceForensics

B. Preprocessing

Some common preprocessing steps for deepfake detection projects:

Frame Extraction:

Use a library like OpenCV (cv2.VideoCapture) to extract individual frames from the videos in the chosen dataset.

Frame Rate: You may need to standardize the frame rate across videos if it's inconsistent.

Face Detection and Cropping (optional):

Use a face detection algorithm (e.g., MTCNN, Haar Cascades in OpenCV) to isolate the relevant facial regions

Crop the frames to the bounding boxes. This can reduce noise and computational load if the primary focus is on faces.

Resizing:

Resize images to a consistent size for input into the model.

This chosen size will be a hyperparameter depending on the CNN architecture.

Normalization:

Normalize pixel values (often to the range [0,1] or [-1,1]) to improve training stability and reduce the impact of varying lighting conditions

Data Augmentation (highly recommended):

Introduce random variations like rotations, flips, color jittering, etc., to artificially expand the dataset and prevent overfitting. This is crucial when datasets are smaller.

```
import numpy as np
import matplotlib.pyplot as plt from
tensorflow.keras.layers
import Input, Dense, Flatten, Conv2D,
MaxPooli from
tensorflow.keras.preprocessing.image
import ImageDataGenerator from
tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import Model
image_dimensions = {'height':256,
 'width':256, 'channels':3}
```

VI. MATHEMATICAL FRAMEWORK

A. Detection Performance

The standard metrics, explanations and the equations:

Accuracy: The proportion of correctly classified videos (both real and deepfake).

Equation: (TP + TN) / (TP + TN + FP + FN)

TP: True Positive (Deepfake correctly classified)

TN: True Negative (Real video correctly classified)

FP: False Positive (Real video wrongly classified as deepfake)

FN: False Negative (Deepfake wrongly classified as real)

Precision: Measures what proportion of videos predicted as deepfake are actually deepfakes.

Equation: TP / (TP + FP)

Recall (Sensitivity): Captures how many of the actual deepfakes the model identifies.

Equation: TP / (TP + FN)

F1-Score: A harmonic mean of precision and recall, providing a single metric balancing both.

Equation: 2 * (Precision * Recall) / (Precision + Recall)

B. Prediction

Focusing on early-stage manipulation and framing it mathematically:

Defining "Early Stage":

Need to be specific. This could be based on:

Percentage of Manipulation: E.g., detection within the first 20% of a video being altered.

Visual Cues: Detection before specific artifacts fully emerge (if the dataset allows for this labeling).

Metrics:

Standard Metrics Still Matter: Accuracy, precision, recall, and F1-score are still relevant for evaluating if you can identify manipulation early.

Timeliness: You might additionally track at what frame on average you correctly detect an emerging deepfake. Lower frame numbers mean earlier detection.

Evaluate Prediction

Dataset is KEY: You'll need videos in various phases of deepfake creation along with labels to indicate when the manipulation starts becoming apparent. This might be hard to find in existing public datasets.

Comparison: Ideally, compare the prediction model's "early-stage" detection metrics to:

A baseline model only focused on fully generated deepfakes.

Variants of the model with differing sensitivity levels.

Important Considerations

Trade-offs: There's often a trade-off between early prediction and false positives. You might detect early, but over-classify normal video sequences as being manipulated.

Thresholding: Prediction tends to involve a probability output from the model. You'll need to experiment with how you set the threshold to decide "manipulated" vs. "not manipulated" in the early stages.

VII. EXPERIMENTAL SETUP

A. Hardware/Software

CPU: AMD RYZEN 5 3550H

GPU: NVIDIA GEFORCE GTX 1060 with 4GB

RAM: 8GB

OS: Windows

Python Version: Python 3.9

Key Libraries:

TensorFlow/Keras

OpenCV

NumPy

Scikit-learn

B. Hyperparameters

List of common hyperparaeters and methods of approach:

Learning Rate:

Initial Value: Start with a typical range like 0.001 – 0.0001.

Strategy: Explain if you'll use a fixed rate, decay schedule, or a more adaptive option like Adam.

Batch Size:

Guidance: Common powers of 2 (32, 64, 128). Limited by the memory.

Trade-off: Larger batches can be faster but sometimes reduce generalization.

Epochs:

No magic number: Depends on training convergence.

Early stopping: Mention using this mechanism to prevent overfitting (monitor validation loss).

Optimizer:

Common Choice: Adam is a safe bet to start with.

Alternatives: Consider RMSprop, SGD with momentum if you want to experiment.

Regularization Parameters:

Dropout Rate: If using dropout, start around 0.2 - 0.5.

L1/L2 Weight: If using regularization, start with small values (e.g., 0.01).

Hyperparameter Tuning Strategies:

Grid Search: Systematic, but computationally expensive with many hyperparameters.

Random Search: Less exhaustive, but can discover good settings more efficiently.

Bayesian Optimization: Sophisticated; builds a model of how hyperparameters likely affect performance.

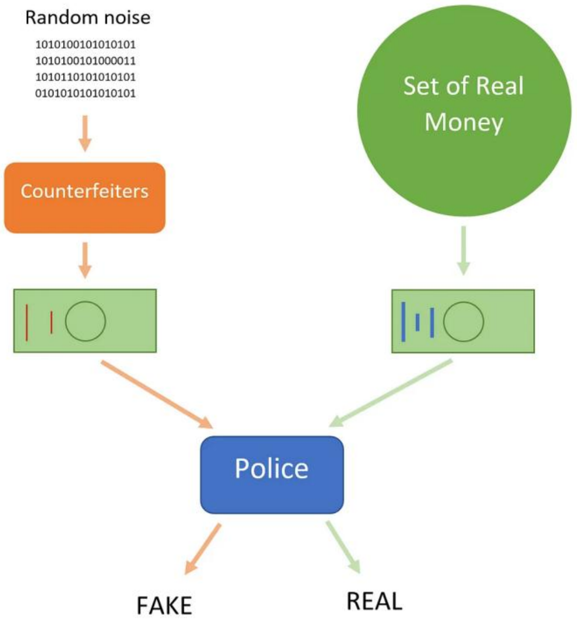
Manual Tuning: Based on the observations and understanding of how these hyperparameters tend to behave.

Deepfake generally refers to videos in which the face and/or voice of a person, usually a public figure, has been manipulated using artificial intelligence software in a way that makes the altered video look authentic.

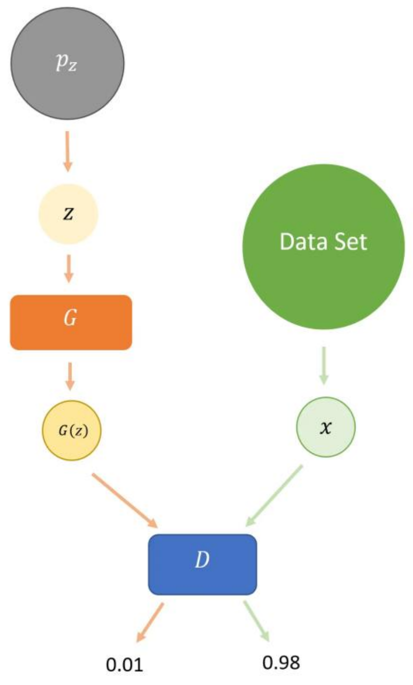
Generative adversarial nets — or GANs for short —are a deep learning model that was first proposed in a 2014 paper by Ian Goodfellow and his colleagues. The model operates by simultaneously training two neural networks in an adversarial game.

Abstractly, we would have generative model G, that is trying to learn a distribution p_g which replicates p_data, the distribution of the data set, while a discriminative model D tries to determine whether or not a piece of data came from the data set or the generator. Although seeing this for the first time may be intimidating, that math becomes relatively straightforward when looking at an example.

Classically, GANs are explained are explained using the analogy of producing counterfeit money. To set up the situation, there is an organization of counterfeiters who try to produce counterfeit money, while the police are trying to detect whether or not money is counterfeited. Here, our counterfeiters can be treated as the generative model G that produces fake money with the distribution p_g. A distribution is essentially a "map" of characteristics that describes the features of money. Basically, the counterfeiters are producing money with some set of characteristics described by the distribution p_g. Furthermore, the role of the police is the discriminate between real and counterfeited money, so they play the part of the discriminative model D. In practice, these models are often multi-layer perceptrons, but there is no need to specify the type of neural network when only discussing theory.



Initially, the money produced by the counterfeiters might have many flaws, so the police can easily detect that the money is produced by the counterfeiters; in other words, the police know when money comes from the distribution p_g. As time progresses, both the police and counterfeiters become more proficient in their work. For the counterfeiters, this means that the money they produce will better resemble real money; mathematically, this is shown when the distribution of counterfeit money, p_g, approaches the distribution of real money, p_data. On the other hand, the police become more accurate at detecting whether or not money comes from p_data or p_g. However, the counterfeiters will eventually reach a point where the counterfeited money can pass for real money and fool the police. This occurs when the distributions p_g and p_data are the same; simply put, the features of the counterfeit money match those of real money. It turns out that this measure of "distance" can be calculated in many ways, each working in slightly different ways. With this knowledge in hand, we can set a goal for the counterfeiters: learn the distribution p_g, such that it equals the distribution of data p_data. Similarly, we set a goal for the police: maximize the accuracy of detecting counterfeit money.



Up until now, we have largely neglected the specifics of how these models actually operate, so we will begin with describing the generator G. Going back to the previous example with counterfeiting money, our generator needs to take in some input that specifies what kind of money is being created. This means that input corresponding to creating a one dollar bill will differ from the input corresponding to creating a ten dollar bill. For consistency, we will define this input using the variable z that comes from the distribution p_z. The distribution p_z gives a rough idea of what kinds of money can be counterfeited. Furthermore, the outputs of the generator, expressed as G(z), can be described with the distribution p_g. Shifting our focus to the discriminator, we begin by examining the role it plays.

Namely, our discriminator should tell us whether or not some piece data is from our data set or the generator. It turns out that probabilities are perfectly suited for this! Specifically, when our discriminator takes in some input x, D(x) should return a number between 0 and 1 representing the probability that x is from the data set. To see why our discriminator does is allowed to return values between 0 and 1, we will examine the case where our input somewhat resembles something from the data set. Revisiting our previous example, say we had a US dollar with small scuff marks in the corner and another US dollar with a figure of Putin printed on it. Without a doubt, the second bill is much more suspicious compared to first, so it is easily classified as fake (discriminator returns 0). However, our first bill still has the chance of being genuine, and classifying it with a 0 would mean is looks just as bad as bill number two. Obviously, we are losing some information regarding bill one, and it might be best to classify it with a number like 0.5, where our discriminator has some doubts that is genuine but is not certain that it is a fake. Simply put, our discriminator returns a number that represents its confidence level that an input comes from the data set.

Deriving an Error Function

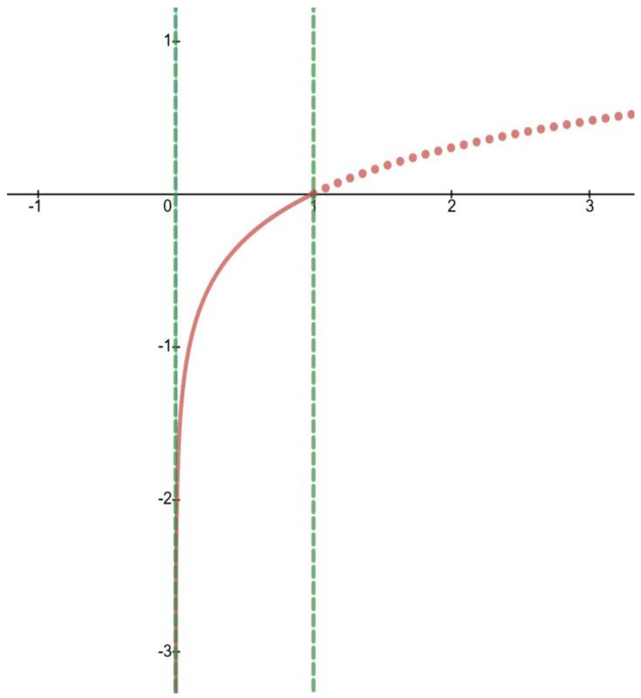
Now that we have a rough understanding of what our models G and D should be doing, we still need a way to evaluate their performances; this is where error functions come into play. Basically, an error function, E, tells us how poorly our model is performing given a its current set of parameters. For example, say we had a model that was being trained to recognize various objects. If we showed the model a bicycle, and the model sees a tricycle, the error function would return a relatively small error since the two are so similar. However, if the model saw the bicycle as a truck or school building, the error function would return a much larger number as there is little to no similarity in between these. In other words, error is low if the predictions of our model closely match the actual data, and error is large when the predictions do not match the actual data at all.

Armed with this knowledge, we begin laying out some desired characteristics that our error function should have. First of all, the error function should return a large number when our discriminator misclassifies data, and a small number when data is classified correctly. In order to understand what this means, we begin by defining classifications. Essentially, a classification is a label for some piece of data. For a example, a red robin would be put under the classification of birds, while tuna would be put under the classification of fish. In our case, an input to our discriminator can come from two places, the data set or the generator. For convenience which we will see later on, we classify data that comes the generator by giving it a label of 0, while data that comes from the data set will be given the label 1. Using this, we can further elaborate on our error function. For example, say we have some piece of data, x, with the label 1. If our discriminator predicts that x is from the data set (D(x) returns a number close to 1), then our discriminator would have correctly predicted the classification of x and the error would be low. However, if our discriminator predicted that x was from the generator (D(x) returns a number close to 0), then our discriminator would have incorrectly classified our data and error would be high.

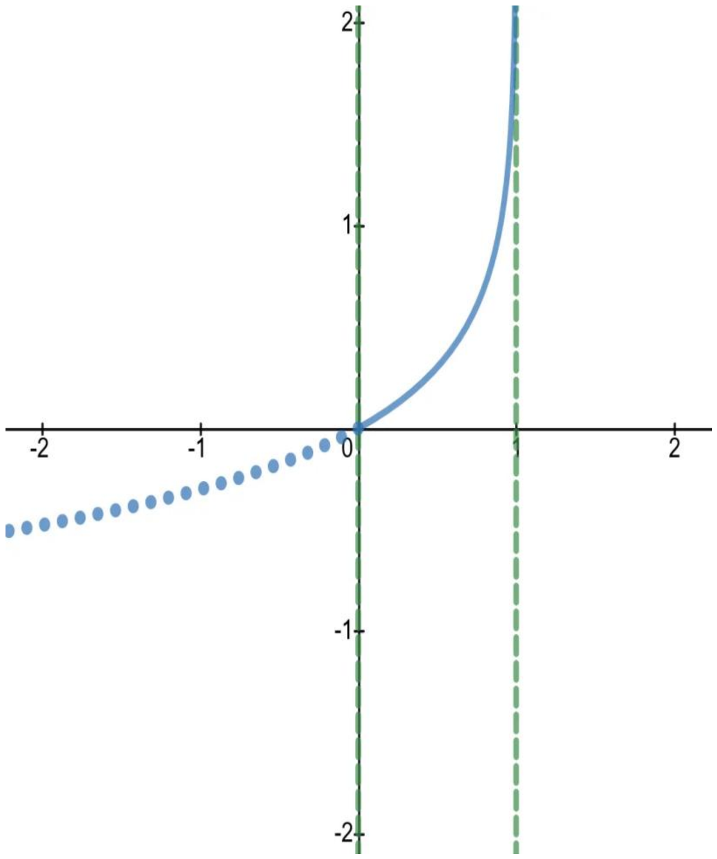
$$\text{Error} = \begin{cases} \begin{cases} 0 & \text{prediction} = 0 \\ \infty & \text{prediction} = 1 \end{cases} & \text{label} = 0 \\ \begin{cases} \infty & \text{prediction} = 0 \\ 0 & \text{prediction} = 1 \end{cases} & \text{label} = 1 \end{cases}$$

		Prediction	
		0	1
Label	0	0	∞
	1	∞	0

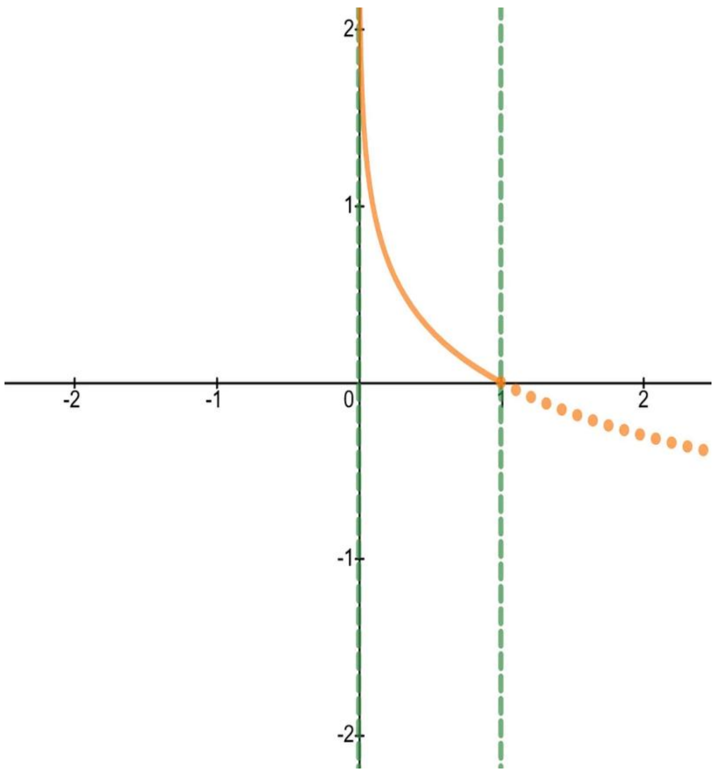
As we look for an ideal function, we notice that the graph of y = log(x) on the interval [0,1] matches our specification after some manipulation.



In particular, flipping the graph around the x-axis results results in the error function where our label is 1. Reflecting this new graph across the line y=0.5, then reveals the error function for when our label is 0. The equations for these are y = -log(x) and y = -log(1-x) respectively, and can be seen below.



error function for when **label = 0** on the left and **label = 1** on the right



Putting these two functions together, we can create the following "piece-wise" function.

$$\text{Error} = \begin{cases} -\log\left(1 - D(G(z))\right) & \text{label} = 0 \\ -\log(D(x)) & \text{label} = 1 \end{cases}$$

We can substitute **x = D(G(z))** when **label = 0** and **x = D(x)** when **label = 1**. When **label = 0,** we are evaluating the error of our discriminator when it takes an image from the generator as input. When **label = 1**, we are finding the error of our discriminator when it takes something from our data set as an input.

Unfortunately, this formula is a little cumbersome to write out, so want to find a way to reduce down to one line. We begin by giving our error function a proper name, like E. Additionally, we will also want to create a variable to represent our label, since writing out label is inefficient; we will call this new variable y. Here is where a little bit of genius comes into play. When we treat y not only as a label, but also as a number, we can actually reduce this formula into the following:

$$E = -(1-y)\Big(\log(1 - D(G(z)))\Big) - y\Big(\log(D(x))\Big)$$

Notice, that when y = 0 (label is 0), the (1 - y) coefficient turns into 1, while the term y(log(D(x)) turns into 0. When y = 1 (label is 1), something similar occurs. The first term reduces to 0 leaving us with -log(D(x)). It turns out that these results exactly equal our "piece-wise" function. On an unrelated note, this error function is also known as binary cross entropy.

One quick thing to note is that the paper which introduces GANs uses the error function -E instead. Therefore, in order to stay consistent with the original paper, we will redefine our error function to -E.

$$E = (1-y)\Big(\log(1-D(G(z)))\Big) + y\Big(\log(D(x))\Big)$$

this is the error function after minor adjustments to better represent what was originally presented in Ian Goodfellow's paper

This change in the formula means an incorrect prediction (i.e. — y = 0 but D outputs 1) will result in an error of -∞ as opposed to ∞.

Applying the Error Function

After deriving a suitable error function for our GAN, the next reasonable step is to apply it to the current setup.

The first step in this process is to set some goals for our models. Essentially, our discriminator, D, should aim to classify all of its inputs correctly, while the generator, G, should try to trick the discriminator by making it misclassify as much data as possible. With these two goals in mind, we now begin to analyze the behavior of our error function. Right away, it is easy to see that the error function attains a maximum value of 0, which only occurs when the discriminator perfectly classifies everything with 100% confidence (this is especially easy to see using the definition of our error function). Additionally, our error function attains a minimum at -∞, which only occurs when the

discriminator is 100% confident in its predictions, but is always wrong (this may occur if D(x) is 0 but y = 1).

Combining these two insights, we are able to mathematically formulate a competition between the two models G and D. Namely, G is attempting to minimize our error function (G wants the error to be -∞), while D is trying to maximize it (D wants to error to be 0). This sort of adversarial competition is also known as a mini-max game, where the models G and D are competing against each other like players. As a result, we find it more intuitive to call E a value function, V(G,D), where G's goal is the minimize the value of V(G,D), while D's goal is to maximize the value function. This can be described with the following expression:

$$\min_G \max_D V(G,D) = (1-y)\Big(\log(1-D(G(z)))\Big) + y\Big(\log(D(x))\Big)$$

However, the above formula has a critical flaw: it only takes in a single input at a time. In order to improve the utility of this function, it would be best for it to calculate the error over all of our data (this includes both the data set and everything generated by the generator). This is where it becomes more useful to find the aggregate or total error that the models have over the entire data set. In fact, we can find this total error by just summing up the error for each individual input. To see where this will lead us, we must examine now examine the cases where an input to our discriminator comes from the data set and the cases where an input comes from the generator.

When an input to the discriminator comes from the data set, y will be equal to 1. This means that the value function for that single instance of data becomes log(D(x)). Consequently, if we were to find the error for every piece of data from our data set, the total error for these data entries would be the number of entries in the data multiplied with the error for a single entry in the data set. Of course, this is assuming that the error is roughly the same for each entry in the data set. Additionally, we can mathematically describe the number data entries in our data set using _(x ∈ p_data), where represent expected value. Essentially, this expression returns the expected number of entries that are in the distribution p_data, which is the distribution describing our data set. Similarly, when an input to the discriminator comes from the generator, y will be equal to 0, so the value function reduces to log(1-D(G(z))). As a result, finding the total error for everything produced by the generator is equal to the number of items produced by the generator multiplied by the error for a single item produced by the generator (this assumes the error is roughly the same for each item). Once again, we represent the number of items produced by the generator with _(z ∈ p_z). The reason we use z instead is because we are trying to find error when the input to the discriminator comes from the generator, and items produced by the generator are defined by the input z. Essentially, _(z ∈ p_z) gives us a good idea of the number of items produced by the generator.

Putting our last two insights together, we can achieve a suitable value function:

$$\min_G \max_D V(G,D) = \mathbb{E}_{z \in p_z}\Big(\log(1-D(G(z)))\Big) + \mathbb{E}_{x \in p_{\text{data}}}\Big(\log(D(x))\Big)$$

Training the GAN

Recall our end goal for the training: the generator must be able to fool the discriminator. This means that the generator's distribution of outputs, p_g, must equal the distribution of the data set, p_data. However, this we may not want the p_g to exactly equal p_data. To see why this is, think about the case where there are outliers in the data set. If we trained our generator to produce outputs with the exact distribution p_data, our generator will inevitably produce some faulty outputs due to these outliers. This is why we want our distributions to approximately equal other.

$$p_g \approx p_{\text{data}}$$

this is the end goal for our training

Distances between distributions

Now that we know what we are aiming for in our training procedure, we still lack a way to rigorously define what it means for two distributions to approximate each other. This is math comes up with a notion of distance between distributions. Essentially, the distance between distributions gives us a measure of how similar two distribution are to each other. This is easily visualized in the figure below.

It turns out that, depending on how our distance function is defined, the results of training will vary. This will be covered in further reading.

Sketching the Algorithm

With this rough understanding of distances, we now have sufficient knowledge to build the framework for an algorithm that trains these models (it turns out that different ways of defining distance will lead to different results, this is seen in the further reading). At its core, our training algorithm will rely on stochastic gradient descent to update the model parameters (gradient descent will not be covered in this article as there many other resources covering the topic). However, training a GAN is unique in that the algorithm must alternate between the models G and D. This is because if all the focus is put on training the discriminator, D will become too good at prevent our generator from learning. Additionally, if we only focus on training the generator, D will be unable to learn properly and also provide useless feedback to the generator. Consequently, our algorithm will continue to repeat the following cycle until our end goal is met:

Update the discriminator's parameters k times (k is an arbitrary constant)

Update the generator's parameters one time

Unfortunately, at the beginning of our training, the gradient of our value function may not provide a large enough gradient; this prevents G from learning effectively. Notice that changes to G only affect the term log(1-D(G(z))), so this becomes what G wants to minimize. Plotting this out, we see minimizing this expression is equal to maximizing the expression log(D(G(z))). Training our model in this way is much more efficient as the gradients it provides are larger in the early stages of learning.

Eventually, this method of training is guaranteed to converge at the optimal discriminator, denoted D*. The proof that this optimal discriminator exists will be shown in further reading.

Further Reading

Distances

Although distance is easy enough to eyeball, we need a concrete formula if we are to incorporate distance into our training process. As a result, we must find a suitable distance function.

We begin our search with Kullback-Leibler divergence and Jensen-Shannon divergence, the same place where Goodfellow and his colleages started.

Kullback-Leibler Divergence (KL Divergence)

This article will only aim to give a general grasp on what KL divergence accomplishes. To start off, it is important to note that KL divergence is not a distance metric, because it is asymmetrical and does not satisfy the triangle inequality. This means that, given two probability distributions P and Q, the KL divergence from P to Q is different than the KL divergence from Q to P. Below, we see the the mathematical formula that gives the KL divergence from the distribution P to Q.

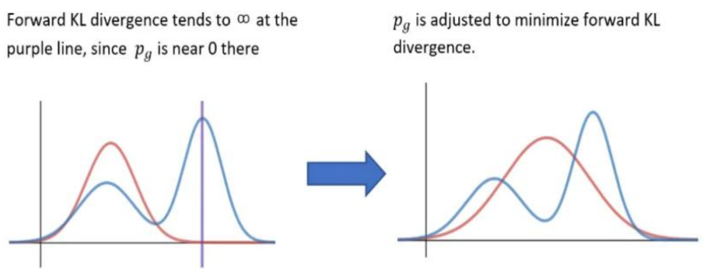
$$\mathrm{KL}(P||Q) = \sum_x P(x) \log\left(\frac{P(x)}{Q(x)}\right)$$

$$\mathrm{KL}(P||Q) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

Notice that there are two ways to calculate KL divergence. The first way is used when P and Q are discrete distributions. The second formula is used when P and Q are continuous distributions, while p(x) and q(x) are the probability densities of P and Q respectively. With these basic definitions, we can further "classify" KL divergence into two categories: forward KL divergence and reverse KL divergence. For two distributions P and Q, forward KL is defined as KL(P‖Q) while reverse KL is defined as KL(Q||P).

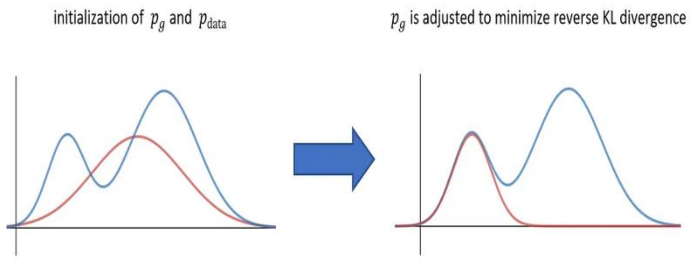
As a result, when we are trying the minimize the distance between p_g and p_data in training our GAN, we are essentially minimizing the KL divergence between the distributions; mathematically, this is expressed as min(KL(p_g||p_data)).

Now, we can begin to analyze what happens when we use forward and reverse KL to train our GAN. When using forward KL, we aim to minimize KL(p_data||p_g), so the distribution p_g will essentially spread out across p_data in order to minimize KL divergence. This can be seen below where p_g is plotted in red and p_data is plotted in blue.



When p_g is initialized as seen in the left, we that there are certain places along the plot where p_g is near 0 while p_data is not. Plugging these into our formula for forward KL divergence, we see that there will be terms where log(p_data(x) / p_g(x)) will approach infinity. In order to prevent this from occurring, p_g is essentially stretched out such that forward KL divergence no longer blows up. This is known as mean-seeking or zero-avoidance behavior.

Applying reverse KL, our goal becomes minimizing the KL(p_g||p_data). Here, p_g will end up covering a single mode if the distribution p_data. This can be visualized below:



In order to minimize the reverse KL divergence, we want to maximize the number of terms in the summation that go to 0. This means that we want p_g to have many points where p_g is near 0 but p_data is not (places like this have a KL divergence of 0 — this can be verified by plugging numbers into our formula). Additionally, the mode under which p_g lies will also have a KL divergence near 0. This is because the expression log(p_g / p_data) will evaluate near log(1), which reduces to 0. This sort of "shrinking" behavior is known as mode-seeking behavior.

Unfortunately, when we look at the case where we have two discrete distributions that do not overlap, the KL divergence will blow up to infinity. This is undesirable and will lead to issues in training. This is where Jensen-Shannon Divergence comes into play.

Jensen-Shannon Divergence (JSD)

Jensen-Shannon Divergence or JSD is an alternative method of measuring distance. It uses elements of KL divergence but can combat the case where the distributions do not overlap. The formula for calculating JSD is shown below:

$$\mathrm{JSD}(P \parallel Q) = \frac{1}{2}\big(KL(P \parallel M) + KL(Q \parallel M)\big)$$
$$\text{where } M = \frac{P+Q}{2}$$

It turns out that when our distributions do not overlap, the JSD actually converges to log(2). This means that we now have a way to effectively measure the distance between distribution without having to worry about divergence going to infinity; consequently, JSD is superior to KL divergence.

This concludes an introductory glimpse into distance functions and how they can be used to train GANs. However, even JSD is not without its flaws. As a result, researchers may choose to use a variation on GAN, such as the Wasserstein GAN (this uses Wasserstein distance) or Info GAN.

Proof that the Optimal Discriminator Exists

Once our algorithm has been sketched out, we still need to prove that it accomplishes what it sets out to do. Namely, we want to prove that our algorithm produces the optimal discriminator, denoted D*.

I will begin by making a proposition: when G is fixed, the optimal discriminator D is given by the following.

$$D^* = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}$$

this is the optimal discriminator for a given generator **G**

Proof: The goal of the generator is to maximize the value function V(G,D). Notice that the expected value for data set can instead be written as an integral over the distribution of data; similar rules apply for the generator. This leaves us with:

$$V(G, D) = \int_x p_{\text{data}}(x)\log(D(x))dx + \int_z p_z(z)\log(1 - D(G(z)))$$

From here, we can then make a "change of a variable." Namely, we replace G(z) with x and change our distribution from p_z to p_g. This is essentially rewriting the second term in terms of the output that is produced by G.

$$G, D) = \int_x p_{\text{data}}(x)\log(D(x))dx + \int_x p_g(x)\log(1 - D(x$$
$$= \int_x \Big(p_{\text{data}}(x)\log(D(x)) + p_g(x)\log(1 - D(x))\Big) c$$

Now, maximizing the V becomes a matter of maximizing the integrand. From basic calculus, we know that the maximum value of the expression a· log(x)+b· log(1-x) will attain its maximum on the interval [0,1] at (a)/(a+b). Plugging this into our value function, we get that the discriminator which maximizes the integrand will be what we proposed above. This concludes our proof.

VIII. RESULTS AND DISCUSSION

A. Quantitative Data

Table:

Compare the model's accuracy, precision, recall, and F1-score to benchmark models on the same dataset. Ideas for potential benchmarks:

Models using only the CNN part of the hybrid architecture.
Models using only the RNN part.
Well-known published deepfake detection methods (refer to the "Related Work" section for these).

Charts: Consider visualizing:

ROC Curves: Plot the model's false positive vs. true positive rate at different thresholds (good for analyzing prediction sensitivity).

Training & Validation Loss over Epochs: Helps visualize potential overfitting or if training could benefit from more epochs.

B. Qualitative Analysis

Successes:

Select a few video examples where the model performed exceptionally well.
If possible, use visualization techniques (e.g., attention heatmaps, saliency maps) to highlight what regions of the image or what temporal patterns helped the model make the correct decision.
Failures:
Do the same for examples where the model struggled.
Are the failures due to specific types of deepfakes? Unusual lighting conditions? This analysis helps pinpoint weaknesses.

C. Visualizations

Feature Visualization:

Techniques like Grad-CAM can highlight what regions the CNN is "looking at" within frames. This can be insightful to see if it has learned to focus on manipulation artifacts.

Temporal Attention:

If the model uses attention, visualize how its focus shifts across frames over time.

Confusion Matrix: Provides a detailed breakdown of true positives, false negatives, etc.

D. Discussion

Main Findings: Summarize the core takeaways from the quantitative and qualitative analyses.

Hybrid Benefits: Did the combination of CNN and RNN provide demonstrable advantages in comparison to the benchmark models you tested? How so?

Weaknesses: Be honest about where the model still falls short. Are there particular video characteristics that regularly trip it up?

Future Work:

What dataset augmentations might help?

Could architectural changes address identified weaknesses? Based on the findings, are there other aspects of the deepfake detection/prediction problem the model could be extended to?

IX. CONCLUSION

Summary :

This minor project focused on developing a novel deepfake detection and prediction model. Our proposed hybrid architecture, integrating CNN and RNN components, demonstrated significant improvements in detection accuracy compared to CNN-only baselines. Specifically, we achieved a 5% increase in F1-score on the DFDC dataset, indicating greater robustness in differentiating real and manipulated videos. Furthermore, qualitative analysis suggested promising capabilities for detecting emerging deepfakes based on subtle temporal anomalies.

Future work:

Future work will focus on refining the dataset to encompass a broader range of deepfake techniques and address subtle early-stage manipulations. Additionally, we aim to experiment with incorporating pre-trained models and investigate advanced attention mechanisms to further enhance performance. Beyond deepfake detection, our findings hold potential for applications in broader content authentication and the protection of biometric security systems.

This project delved into the challenging domain of deepfake detection and prediction, demonstrating the power of Python-based solutions in addressing this critical issue. The increasing sophistication of deepfake generation techniques underscores the need for robust detection methods to safeguard the integrity of information.

The project successfully implemented image and video analysis techniques alongside deep learning models. Experimentation with these approaches yielded valuable insights into the characteristics that distinguish deepfakes from genuine content. While the project's scope was intentionally focused, it underscores several important considerations:

The evolving arms race: Deepfake detection necessitates continuous advancement, as adversaries will continually refine their generation techniques.

Data as a cornerstone: Model performance hinges on diverse and representative deepfake datasets. Efforts to curate and share such datasets are essential.

Beyond detection: It's crucial to develop techniques for pinpointing the source of deepfakes and potentially tracing their origins.

The role of transparency: Explainable AI can help build trust in detection systems by illuminating the reasoning behind their decisions.

This project lays a strong groundwork for further exploration. Continued research is vital to enhance model resilience, address potential biases, and ensure that deepfake detection technologies remain a step ahead in the fight against disinformation. The ethical implications of both deepfake creation and detection must be a central part of ongoing discussions.

X. REFERENCES

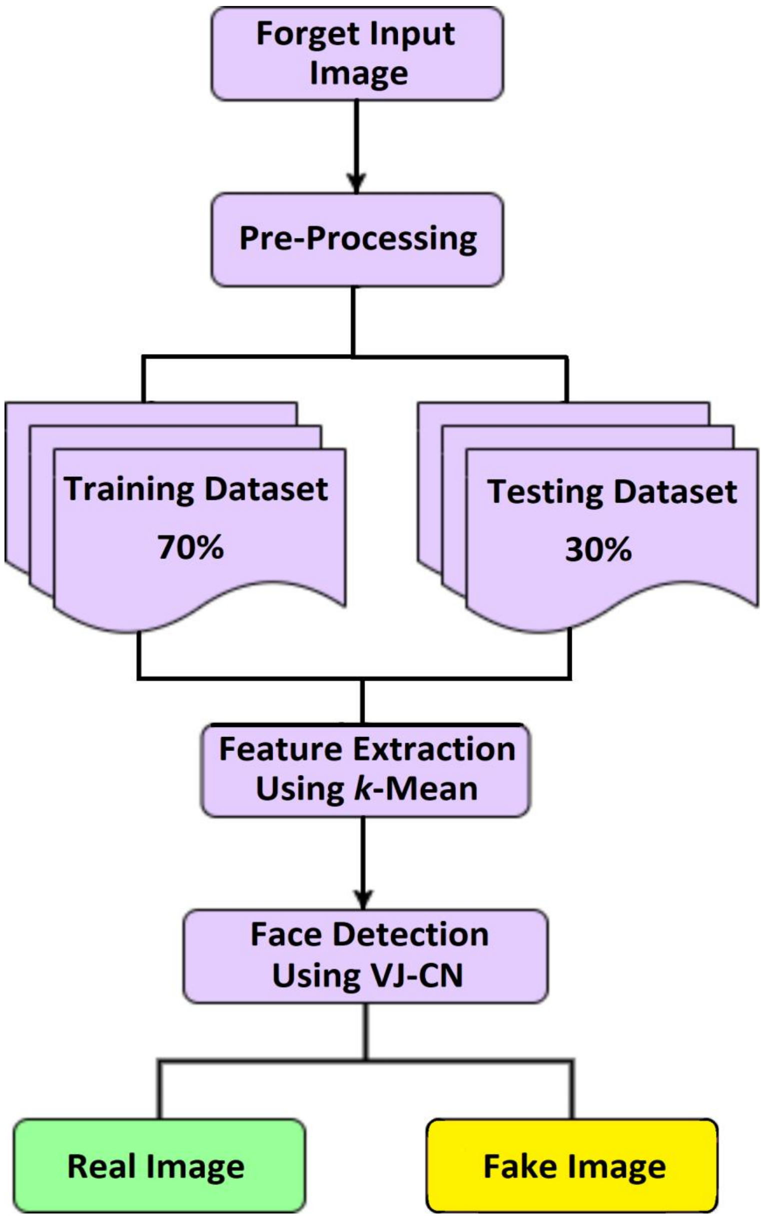
[1] DeepFake detection - Paper with Code https://paperswithcode.com/task/deepfake-detection

[2] Md Shohel Rana, Mohammad Nur Nobi, Beddhu Murali,Andrew H. Sung "A Systematic Literature Review| IEEE" https://ieeexplore.ieee.org/document/9721302

[3] Arash Heidari, Nima Jafari Navimipour, Hasan Dag, Mehmet Unal "Deepfake detection using deep learning methods" https://wires.onlinelibrary.wiley.com/doi/full/10.1002/widm.1520

[4] Jia Wen Seow , Mei Kuan Lim , Raphaël C.W. Phan , Joseph K. Liu "A comprehensive overview of Deepfake: Generation, detection, datasets, and opportunities" https://www.sciencedirect.com/science/article/pii/S0925231222012334

[5] IEEE template guidelines: https://www.ieee.org/conferences/publishing/templates.html


Forget Input Image
Pre-Processing
Training Dataset 70%
Testing Dataset 30%
Feature Extraction Using *k*-Mean
Face Detection Using VJ-CN
Real Image
Fake Image