CSEC 520/620 Cyber Analytics and Machine Learning

DEEPFAKE DETECTION

Group 6 - Project 3 - ML and Security
January 15, 2024

Bala Prasanna Gopal Volisetty, Mehul Sen, Vaibhav Savala
Department of Cybersecurity
Golisano College of Computing and Information Sciences
Rochester Institute of Technology
`bv1459@rit.edu, ms1450@rit.edu, vs5938@rit.edu`

1 Abstract

This study tackles the problem of detecting ransomware using machine learning. We tested different machine learning models to see which one can best spot ransomware attacks. Our tests included simpler models like Decision Trees and more complex ones like Random Forest and KNN. We found that Decision Trees were surprisingly good at detecting ransomware, even better than the other models. We used a special set of data for training our models, which helped them learn better. Our findings could help make better tools for fighting against ransomware. Through our efforts, we gained substantial insight into enhancing these instruments and believe deeper understanding will emerge as our work continues.

2 Introduction

Access to complex deep learning models and the ease of training and machine learning offer several benefits. Machine learning has proven to be increasingly helpful for automation, scalability, and pattern recognition. Almost every technology we use today has been or could be enhanced using artificial intelligence. However, AI can be misused for malicious purposes, just like any other tool. Threat actors may exploit AI to propagate misinformation, commit fraud, and falsify evidence. One example of such misuse is deepfakes, where AI-generated video or audio is overlaid onto or fully synthesized to impersonate a victim. This could result in falsified forensic evidence, discrediting or defaming victims.

Deepfakes can cause systemic socio-political risks such as manipulation of civil discourse, interference with elections and national security, and erosion of trust in journalism and public institutions. They can also pose personal risks such as pornography deepfakes, identity theft, and fraud, which can be used to threaten, intimidate, and inflict psychological harm on their victims.

There is a clear need for research into identifying and stopping the spread of deepfakes before irreparable damage is done to individuals. A promising new approach gaining popularity is deepfake detection. This technique leverages AI models trained on authentic and artificially generated audio/video to discern whether media has likely been manipulated. To be effective, this technology must not only accurately identify fabricated content but also operate quickly and efficiently enough to detect deepfakes before they can cause harm.

In this project, we have developed a comprehensive dataset, unprecedented in its scale and diversity, comprising 1000 original video sequences sourced from 977 YouTube videos alongside 1000 manipulated videos generated using a Deepfake Variational AutoEncoder (DF-VAE). Leveraging this dataset, we trained three advanced neural network models: Convolutional Neural Networks (CNN), CNN combined with Recurrent Neural Networks (RNN), and Generative Adversarial Networks (GAN). Our approach aims to benchmark the effectiveness of these models in detecting deepfakes, thereby advancing the capabilities of AI in safeguarding digital authenticity.

In this paper, we detail our methodology in assembling the dataset, describe the unique attributes of each neural network model employed, and present a comparative analysis of their performance in deepfake detection. Through this work, we endeavor to contribute a significant leap forward in the ongoing battle against digital deception. [1].

3 Literature Review

3.1 Concurrent Neural Networks (CNN)

This section covers some of the deepfake detection techniques that employ CNNs to solve this problem.

3.1.1 CNN-based Deep Learning Model for Deepfake Detection

This paper [2] argues that traditional image forensic techniques are often ineffective against deepfakes due to data deterioration from compression and advanced manipulation techniques. To address this issue, the paper proposes a deep learning model that uses Convolutional Neural Networks (CNN) for feature extraction and Long Short-Term Memory (LSTM) for analyzing temporal sequences in videos. The model also employs Recycle-GAN to merge spatial and temporal data, enhancing the detection of deepfakes by considering both frame-level and sequence-level features. The researchers trained their model using the FaceForensics++ dataset and achieved high accuracy rates in detecting deepfakes, outperforming several existing algorithms in terms of accuracy. They found that their model demonstrated significant efficiency in detecting various forms of deepfakes, including face swaps and facial reenactment. The combination of CNN, LSTM, and GAN-based approaches provided a more robust solution compared to traditional image forensics methods.

3.1.2 Deepfake Detection using a frame-based approach involving CNN

This paper [3] discusses the challenges in detecting DeepFakes, which are becoming increasingly difficult to identify due to the high level of realism achieved by modern generative techniques, particularly Generative Adversarial Networks (GANs). To address this issue, the authors propose a CNN model that can distinguish between real and fake faces in videos by analyzing visual artifacts and flaws inherent in DeepFakes. The proposed solution involves breaking down videos into frames, detecting faces within them, and analyzing them using the CNN model. The CNN model focuses on identifying pixel distortion, inconsistencies in facial superimposition, skin color differences, blurring, and other visual artifacts. The authors trained and validated the model using a dataset from the DeepFake Detection Challenge on Kaggle, which included both real videos and DeepFakes. The model achieved a validation accuracy of 85.8% and a validation loss of 0.3403 indicating high level of effecitveness. Thus establishing the effectiveness of the CNN model in detecting DeepFakes by focusing on facial features and pixel distortions, and highlighting the importance of analyzing both spatial (individual frames) and temporal (sequence of frames) data for effective detection.

3.1.3 Deepfake Detection using Multi-path CNN and Convolutional Attention Mechanism

According to this paper[4], current deepfake detection methods have limitations, especially when it comes to analyzing videos that are highly realistic or created using novel methods. The paper proposes a multi-path approach that uses three different CNN architectures: ResNet, DenseNet, and InceptionResNet. This approach is designed to extract and analyze a diverse range of features from the input data. Each path in the multi-path CNN incorporates a Convolutional Block Attention Mechanism (CBAM), which helps the model to focus on the most informative features, enhancing its ability to detect subtle anomalies typical of deepfakes. The researchers trained their model using the Deepfake Detection Challenge (DFDC) dataset and achieved an accuracy of 94.0% alongside an F1-score of 93.9%, outperforming several baseline models. The researchers found that their multi-path CNN approach with CBAM allows for a comprehensive analysis of both spatial and channel-wise features in videos, leading to more effective deepfake detection, thus providing a more nuanced and detailed analysis compared to traditional single-path CNN models.

3.1.4 High Performance DeepFake Video Detection using CNN-Based Models with Attention to Specific Regions and Manual Distillation Extraction

This paper [5] argues that current deepfake detection models are often too computationally heavy and inefficient due to their use of excessively large backbones. To address this issue, the authors propose a CNN-based model that utilizes manual distillation extraction and attention to specific regions in videos. This approach focuses on the most informative features, reducing the learning burden on the network. The model involves several preprocessing steps such as face extraction, augmentation, and patch extraction. Additionally, it uses frame and multi-region ensemble techniques to effectively analyze video frames. The authors were able to achieve an impressive Area Under the Curve (AUC) score of 0.958 and an F1-score of 0.92 on the DeepFake Detection Dataset. They achieved an even higher AUC score of 0.978 and F1 score of 0.962 on the smaller Celeb-DF. Thus showcasing that by concentrating on the most relevant elements and using manual distillation, the model is able to achieve a high accuracy.

3.1.5 A Hybrid CNN-LSTM model for Video Deepfake Detection by Leveraging Optical Flow Features

This paper [6] sheds light on the growing sophistication of deepfake technology. The authors point out that traditional detection methods are unable to effectively identify ultra-realistic fake videos created by deepfake technology. This is because these methods fail to capture the temporal information of individual video frames. To address this issue, the authors propose an optical flow-based feature extraction method that captures the temporal dynamics between video frames. This method detects deepfakes by analyzing the patterns of apparent motion. The authors developed a hybrid model that combines Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN), specifically Long Short-Term Memory (LSTM) networks. This allows the model to analyze both spatial and temporal features of video frames. The model was trained and tested on open-source datasets, achieving an accuracy of 66.26% on DFDC, 91.21% on FF++, and 79.49% on Celeb-DF. These results demonstrate the effectiveness of the proposed method against different types of deepfakes, highlighting the importance of temporal feature analysis in deepfake detection.

3.2 Recurrent Neural Networks(RNN)

In this section, we will discuss some of the deepfake detection techniques in the literature which have leveraged Recurrent neural networks to solve this problem.

3.2.1 Deepfake Video Detection Using Recurrent Neural Networks

This paper [7] uses an end-to-end trainable recurrent deepfake video detection system using a convolutional LSTM structure, designed to process sequences of frames for deepfake detection. The system consists of two main components: a Convolutional Neural Network (CNN) for extracting features from each frame and a Long Short-Term Memory (LSTM) network for analyzing the temporal sequence of these features. The CNN employed is an InceptionV3 model, adapted to output a 2048-dimensional feature vector per frame without fine-tuning. These vectors are then fed into the LSTM, which includes a 2048-wide unit with a dropout chance of 0.5, followed by a 512 fully-connected layer with another dropout chance of 0.5. The LSTM effectively processes the sequential data, and a softmax layer is used at the end to compute the probability of the video being either a deepfake or unaltered. This LSTM module, integral to the pipeline, is trained in an end-to-end manner, negating the need for auxiliary loss functions. This setup aims to robustly identify manipulated video content by analyzing both the individual frame features and their temporal sequence. They use HOHA dataset as the source and collected 300 deepfake videos from multiple video-hosting websites. They use 70/15/15 split for training, validation and testing the model. The performance of the model is based on the number of frames selected in this paper. For 20 frames, the accuracy is 96% on the test dataset, For 40 and 80 frames, it is 97.1% on the test dataset.

3.2.2 DeepfakeVideo Detection by Combining Convolutional Neural Network (CNN) and Recurrent Neural Network (RNN)

The paper[8] designs an architecture to detect manipulated videos by combining two types of neural networks: Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN). Specifically, it uses a model called resnext50 32x4d for the CNN part and a single LSTM (a kind of RNN) layer. The LSTM layer takes in a large input of 2048 data points and has the same number of latent (hidden) features. It also includes a dropout rate of 40% and uses a ReLU (Rectified Linear Unit) Activation function, which helps the model make better predictions. the CNN first processes video frames to identify and extract faces. These faces are then fine-tuned, meaning additional layers are added to the network, and the learning rate is adjusted. This fine-tuning ensures that the model learns effectively. The features identified by the CNN are then passed on to the LSTM for further training. Once trained, the LSTM helps determine the accuracy of the model in identifying whether a video has been manipulated or not. Essentially, this system efficiently learns to spot fake videos by analyzing the faces in each frame and understanding their sequence over time. The dataset used in this paper is a mixed dataset. It includes data from DFDC, Face Forensics and Celeb-DF dataset. On a whole, this dataset contains 50% real and 50% fake videos in the dataset. With 20 epochs and using 100 frames, the accuracy obtained with this dataset was 99.83%

3.2.3 Exposing AI-generated videos with motion magnification

In this paper[9], the model used is a joint CNN-LSTM framework designed for deepfake video detection. It incorporates a modified Inception V3 CNN for feature extraction from video frames and an LSTM network to process the temporal sequence of these features. The LSTM has a 2048-wide unit with a dropout probability of 30%, followed by a 512 fully-connected layer with the same dropout probability, and a sigmoid layer for classification. The dataset used for evaluation includes manipulated videos created using different methods: DeepFakes, Face2Face, NeuralTextures, and FaceSwap. The dataset is divided into training, validation, and testing sets, each containing 720 videos for training and 140 videos for both validation and testing for each manipulation method. The results of the study show promising performance, with an average accuracy of 98.99% across the four sub-datasets. This accuracy is closely related to the visual saliency of the videos; the more distinguishable a video is to the naked eye, the higher the accuracy achieved by the model. Notably, even for videos like NeuralTextures, which are hard for humans to distinguish, the model still achieves an accuracy of 98.30%. The Area Under the Curve (AUC) for all models is above 0.99, indicating high overall performance.

3.2.4 In Ictu Oculi: Exposing AI Generated Fake Face Videos by Detecting Eye Blinking

This paper[10] presents a novel method for detecting deepfake videos by focusing on the physiological signal of eye blinking, which is often neglected in AI-generated fake videos. The model used is a Long-term Recurrent Convolutional Network (LRCN), combining a Convolutional Neural Network (CNN) with a Recursive Neural Network (RNN). This model captures the temporal regularities in eye blinking. The LRCN comprises three parts: feature extraction (using VGG16 framework without fc7 and fc8 layers), sequence learning (implemented with LSTM cells), and state prediction. For training and testing, the model used image datasets of open eye states and evaluated the algorithm on authentic and fake videos generated with the DeepFake algorithm. The datasets included the CEW Dataset for closed eye detection. Data preparation involved face detection, landmark extraction, and alignment, with augmentation applied to increase robustness. The results showed that the LRCN method outperformed other methods like Eye Aspect Ratio (EAR) and CNN in distinguishing different eye states, demonstrating its effectiveness in detecting deepfake videos.

3.2.5 Deepfakes Detection with Automatic Face Weighting

The paper[11] describes a method for detecting deepfakes focusing on extracting visual and temporal features from faces within videos using a combination of a Convolutional Neural Network (CNN) and a Recurrent Neural Network (RNN). Recognizing that visual manipulations mainly occur in face regions and that these usually occupy a small part of the frame, the method specifically targets face areas. It involves three key steps: (1) detecting faces across multiple frames using MTCNN, (2) extracting features from these faces using a pre-trained CNN, and (3) estimating predictions with a layer called Automatic Face Weighting (AFW) in conjunction with a Gated Recurrent Unit (GRU). This model is trained and evaluated with the DFDC dataset. The testing accuracy of this model comes out to be 91%.

3.3 Generative Adversarial Networks (GAN)

3.4 Generative Adversarial Networks

The paper [12] addresses the problem of generative modeling using a novel approach known as Generative Adversarial Networks (GANs). The authors introduce a framework involving two neural networks, a generator and a discriminator, which are trained simultaneously through adversarial processes. The generator aims to produce data resembling the training set, while the discriminator evaluates the authenticity of the generated data. This setup creates a dynamic competition, driving improvements in the quality of generated data. The methodology focuses on learning data distributions without requiring explicit density function estimation. The paper evaluates GANs based on their ability to generate realistic images and their potential in various applications like image generation and domain adaptation. The findings highlight GANs' effectiveness in producing high-quality, realistic images and their potential for broader applications in machine learning.

3.4.1 Ten Years of Generative Adversarial Nets (GANs): A survey of the state-of-the-art

The paper [13] provides a comprehensive overview of the developments in GAN technology over a decade. It discusses various GAN architectures, their applications across multiple domains, and recent theoretical advancements. The paper evaluates different GAN models based on their performance in generating realistic and diverse data. Additionally, it explores the integration of GANs with newer deep learning frameworks like Transformers and Large Language Models. The survey also identifies current limitations of GANs and suggests future research directions, highlighting both the achievements and challenges in the field of GANs.

3.4.2 A review of Generative Adversarial Networks (GANs) and its applications in a wide variety of disciplines - From Medical to Remote Sensing

This paper [14] provides a detailed examination of Generative Adversarial Networks (GANs), their variants, and a broad range of applications across different fields. The authors discuss various GAN models, including cGAN, WGAN, and DCGAN, highlighting their unique features and improvements over the original GAN concept. The paper also presents a comprehensive overview of GAN applications in diverse areas such as medical and healthcare, astronomy, remote sensing, and finance, showcasing the versatility and extensive impact of GANs. Additionally, it addresses the challenges and future directions in GAN research, underscoring ongoing developments and potential areas for further exploration.

3.4.3 Deepfake Detection using GAN Discriminators

The paper [15] focuses on the challenge of detecting deepfake videos. The authors explore the use of Generative Adversarial Network (GAN) discriminators for this purpose. They utilize a methodology that involves training GAN discriminators with different datasets, including MesoNet as a baseline, and evaluate their effectiveness in deepfake detection. The paper tests several discriminator architectures and proposes an ensemble method to enhance performance. However, the findings indicate that GAN discriminators, even augmented by ensemble methods, do not perform well on videos from unknown sources, underscoring the difficulty of generalizing deepfake detection across diverse datasets.

3.4.4 Deepfake Forensics, an AI-synthesized Detection with Deep Convolutional Generative Adversarial Networks

The paper [16] delves into the detection and analysis of deepfakes using AI and GANs. It evaluates over a hundred published papers, exploring the application of GANs in creating digital multimedia data and the techniques for identifying deepfakes. The paper discusses the benefits and threats of deepfake technology and proposes methods to mitigate the production of unethical and illegal deepfakes. It concludes by highlighting limitations and suggesting future research directions.

3.5 Our Project

All our models build upon [3] by using a dataset that breaks down videos into frames to analyze their visual artifacts. We then use the three models: CNN, CNN+RNN, GAN on our comprehensive dataset to compare their performance, and come up with a straightforward approach to make deepfake detection more accessible and easier to train and deploy.

4 Project Implementation

As part of our efforts to detect deepfake videos, we developed three different models and tested them using data from two distinct datasets. In this section, we provide details about the datasets, preprocessing, models, and their implementations.

4.1 Datasets

We used data from two datasets for our project: FaceForensics++ [17] and DeeperForensics-1.0 [1].

1. **FaceForensics++:** This dataset contains 1000 original video sequences from 977 YouTube videos. All videos feature a trackable front face without occlusions. We used these original videos as the source for our training and testing data. The videos were downloaded from the FaceForensics++ Github with a C23 compression rate. (https://github.com/ondyari/FaceForensics/tree/master)
2. **DeeperForensics-1.0:** This dataset is comprised of 11,000 manipulated videos created using various end-to-end face swapping methods and Deepfake Variational AutoEncoder (DF-VAE) methods. The dataset includes 7 types of real-world perturbations at 5 different intensity levels. For our project, we utilized the 1000 raw manipulated videos generated by DF-VAE in an end-to-end manner. These videos were downloaded from the DeeperForensics-1.0 Github repository and can be found in the manipulated_videos_part_00.zip file. (https://github.com/EndlessSora/DeeperForensics-1.0/tree/master)

4.2 Preprocessing

The data preprocessing for the project involved a combination of 1000 original and 1000 manipulated videos. The *project_3_preprocessing.ipynb* file was used to define several functions to modify the videos. Additionally, the *ffmpeg* executable was used to change the encoding of the videos. The preprocessing was carried out in the following steps:

1. **Consistent Encoding:** The original encoding of the videos was in C23, which was re-encoded to H.264 for easier handling and manipulation. This was done using the **ffmpeg** tool with the following command when the ffmpeg executable was in the dataset folders.

```
for %i in (*.mp4) do (
    ffmpeg -i "%i" -c:v libx264 -preset medium -crf 23 "temp_%i"
    move /Y "temp_%i" "%i"
)
```

2. **Renaming Videos:** The 'rename_videos' function was used to rename the video files in a given directory. It extracted the first three characters of each filename and renamed the file accordingly. This was done to ensure that the naming for original videos was consistent with the naming for manipulated videos.

3. **Trimming Videos:** The 'trim_videos' function was used to trim each of the input videos to a maximum of 10 seconds long. This was done to ensure faster training and testing times and to make each video consistent with the others.

4. **Extracting Frames:** From the 2000 videos, each 10 seconds long, the 'extract_frames' function was used to extract nine frames, each evenly spaced 1 second apart. The frames were then stored as JPEG images in another folder.

5. **Face Detection and Extraction:** The 'detect_faces' function was designed to detect and extract faces from the extracted frames. It involved reading each frame, using the frontal face detector algorithm by the 'dlib' library to find faces, and saving them as separate images.

6. **Loading Faces into Arrays:** Finally, the images were loaded into a numpy array X using the 'load_faces' function, resizing them to (224,224). A corresponding label array y was also generated containing the corresponding labels. These were then split and used for training and testing the models.

Figure 1 showcases an example of the preprocessing done on a randomly selected video in the dataset for both the original and the manipulated video. Note: The entire code for

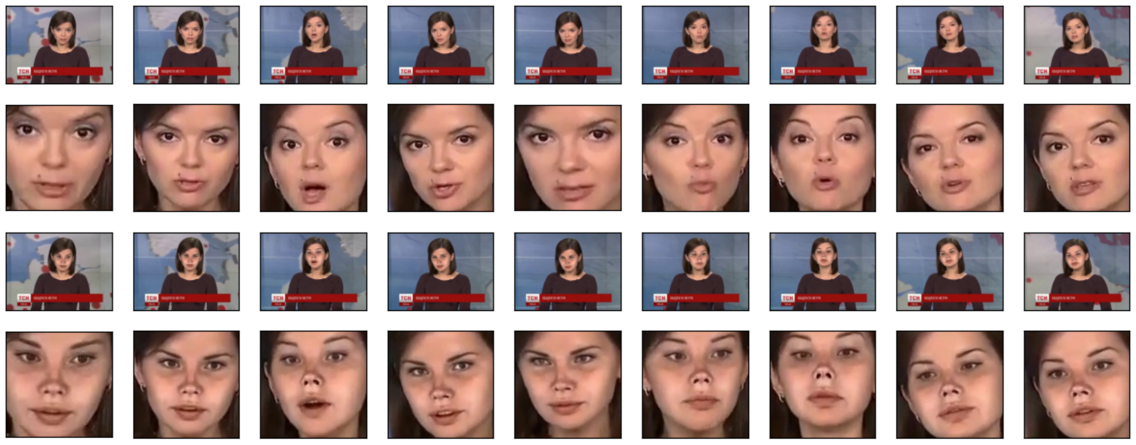

Figure 1: Preprocessing of Videos (From top to bottom: Frames of the original video, Faces extracted from the original frames, Frames of the manipulated video, Faces extracted from the manipulated frames)

preprocessing can be found in the Appendix section 7.1

4.3 Model 1: Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNN) are highly effective in image recognition tasks, making them suitable for detecting deepfakes. For our project, we used a sequential CNN specifically designed for this purpose.

To train the model, we randomly selected 80% of the dataset for training and used the remaining 20% for testing. Additionally, 10% of the training data was used for validation during the training process. The model was trained using a batch size of 32 over 50 epochs, but we implemented the early stopping callback to avoid redundant epochs.

The model itself consisted of Convolutional 2D layers, Batch Normalization layers, and Dense layers. It also used ReLU activation, Max Pooling, Dropout, and Flatten to detect deepfakes.

Here's a breakdown of all the layers and hyperparameters used:

1. **Conv2D Layers:** These are the convolutional layers that are crucial for feature extraction in images. Each layer uses a set of filters (32, 64, and 128 in each successive layer) to scan the input images and learn various features. The filter size (3x3 in this case) is standard for capturing local features like edges and textures. 'Padding=same' ensures the output size is the same as the input size, preserving border information.

2. **ReLU Activation:** The Rectified Linear Unit (ReLU) activation function introduces non-linearity, allowing the model to learn complex patterns. It's computationally efficient and helps avoid the vanishing gradient problem.

3. **Batch Normalization:** This normalizes the output of the previous layer, reducing internal covariate shift, which helps in faster and more stable training.

4. **Max Pooling:** These layers reduce the spatial dimensions (height and width) of the input volume for the next convolutional layer. It helps in reducing the computational load, memory usage, and the number of parameters.

5. **Dropout:** This is a regularization technique to prevent overfitting. By randomly setting a fraction of input units (0.25 and 0.5) to 0 at each update during training, it helps make the model more robust.

6. **Flatten:** This layer flattens the 2D arrays from the pooled feature maps into a single long vector. It's needed before using fully connected layers.

7. **Dense Layers:** These are fully connected layers. A dense layer with 128 neurons is used for further learning from the features extracted and flattened. The final dense layer with a single neuron and sigmoid activation functions as the output layer, providing the probability of the input being a deepfake.

8. **Adam Optimizer:** Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iteratively. The learning rate is set to 0.0001, which determines how big a step is taken during optimization.

9. **Callbacks:** This model uses two callbacks: **ReduceLROnPlateau**, which reduces the learning rate when val_loss has stopped improving, helping fine-tune the model and avoid overfitting, and **EarlyStopping**, which stops training when val_loss has stopped improving, preventing overfitting and saving computational resources.

Combining these layers and hyperparameters in a CNN made it highly effective for Deepfake detection. The model learns hierarchical feature representations from the input images, which are critical in distinguishing between genuine and manipulated images. Using dropout and regularization techniques like Batch Normalization helps build a model that generalizes well on unseen data.

The code used for the CNN model can be found in the Appendix 7.2.

4.4 Model 2: CNN+Recurrent Neural Network (RNN)

A combination of CNN and Recurrent Neural Networks (RNN) has been highly effective in detecting deepfakes. With the help of CNN, we can extract the image features which then can be used to detect inconsistencies by the RNN model. In this way, we will be using a combination of CNN and RNN in our network. We use VGG16 CNN network for our image extraction technique and a RNN model which contains 2 layers of Long Short Term Memory(LSTM) layers and 1 dense layer in our approach. The specific details are as follows:

VGG16 This is a convolutional neural network consisting of 16 layers. Out of which, 13 are convolutional layers and 3 are fully connected layers. the convolutional layers use 3x3 filters with a stride of 1 and same padding, and max pooling is performed over 2x2 pixel window with stride 2.

The network model used is a sequential neural network model with TensorFlow's Keras API. Let's break down each component of out model:

Sequential Model: This network is a linear stack of layers. In a sequential model, each layer has exactly one input tensor and one output tensor.

Input Layer: specifies the shape of the input data. In this case, the model expects input data to have the shape of (49, 512). This could represent a sequence of 49 timesteps, each with 512 features. The input layer doesn't perform any computation; it's just a starting point for the network.

First LSTM Layer: This is a Long Short-Term Memory (LSTM) layer with 64 units. LSTM layers are a type of recurrent neural network (RNN) layer, ideal for processing sequences and time-series data. The return_sequences=True parameter means that this layer will return the full sequence of outputs for each timestep, which is necessary when stacking LSTM layers. Each of the 64 units is a cell in the LSTM layer, where each cell processes information of the input sequence, maintains its cell state, and passes information along to the next cell.

Second LSTM Layer: This is another LSTM layer, but with 32 units. This layer doesn't have return_sequences=True, so it only returns the output of the last timestep. This is typical when we are preparing the sequence data to be flattened and fed into a dense layer, especially in sequence classification tasks.

Dense Layer: This is a fully connected (dense) layer with a single unit. The sigmoid activation function is used to output value between 0 and 1, typically used for our binary classification tasks ie real or fake.

4.5 Model 3: Generative Adversarial Networks (GAN)

1. **Data Preparation:** We used the train_test_split to divide our dataset into training and testing sets. This is standard practice for machine learning tasks to evaluate the model on unseen data.

2. **Discriminator Model:** We then built a CNN-based discriminator model using TensorFlow/Keras Sequential API. The model starts with a Conv2D layer, suitable for processing image data (assuming np-data-X contains image data).The Flatten layer transforms the 2D output of the convolutional layer into a 1D vector. The final Dense layer with sigmoid activation functions as a binary classifier, distinguishing between real and fake/generated images.

3. **Model Compilation and Training:** The discriminator is compiled with the Adam optimizer and binary cross-entropy loss, which is typical for binary classification. The model is trained using the entire dataset. In a typical GAN setup, this step would involve alternating training between the generator and discriminator, but our code focuses solely on the discriminator.

4. **GAN Context:** In the broader context of GANs, the discriminator is one half of the architecture. The other half, the generator, was not the focus of the provided code but is equally important. The generator creates images from random noise, aiming to produce images realistic enough to 'fool' the discriminator.

5. **Training Dynamics:** In GAN training, the generator and discriminator are trained simultaneously in a competitive setup where the generator tries to produce increasingly realistic images, and the discriminator improves at distinguishing real from fake.

5 Performance Discussion

5.1 CNN Performance

The model took 18 epochs to train with Earlystopping callback. The training saw validation loss increase and decrease rapidly. The validation accuracy consistently increased, decreased between epochs 8 and 9, and then increased. The training loss consistently decreased, while the training accuracy consistently increased. Figure 6 shows the model's training and validation loss and accuracy.

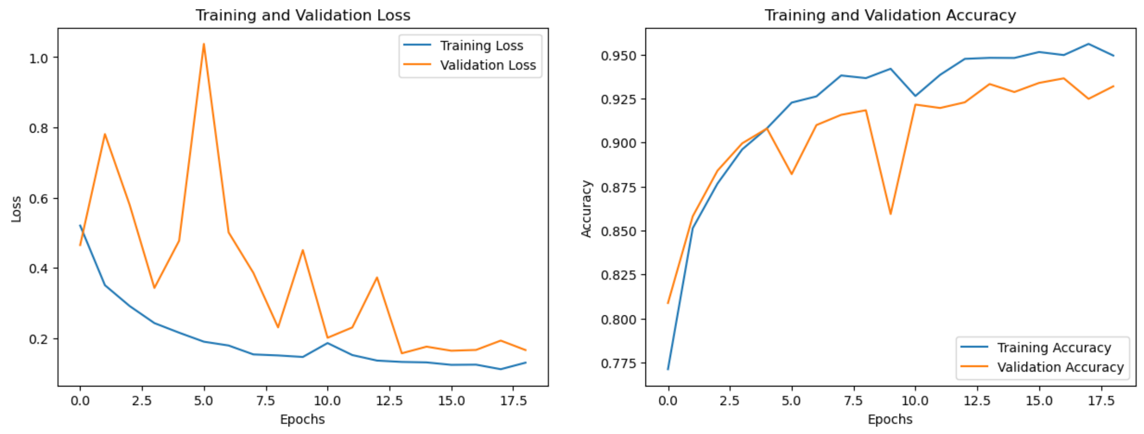

Figure 2: Training and Validation Loss and Accuracy of the CNN Model

On the 20% test dataset, the performance of the CNN model was as follows:

1. **Accuracy: (94.37%)** This high accuracy score indicates that the model correctly classified approximately 94.37% of the test data. It strongly indicates the model's effectiveness, especially in a binary classification task.

2. **Loss: (0.1346%)** This is the model's average Loss over the test data. A loss of 0.1346 is relatively low, suggesting that the model's predictions are, on average, quite close to the actual values.

3. **F1 Score: (0.95 for Original and 0.94 for Manipulated)** The F1-score is a weighted average of precision and recall. Scores near 0.95 and 0.94 suggest an outstanding balance between precision and recall, which is often difficult to achieve, especially in imbalanced datasets.

4. **ROCAUC: (0.9891)** The Receiver Operating Characteristic (ROC) curve is a graphical representation of a classifier's performance, and the Area Under the Curve (AUC) provides an aggregate measure of performance across all possible classification

thresholds. An AUC score of approximately 0.989 is excellent, indicating a very high level of separability between the classes. This means the model is very good at distinguishing between the two classes.

The confusion matrix for CNN classifications is shown in Figure 3

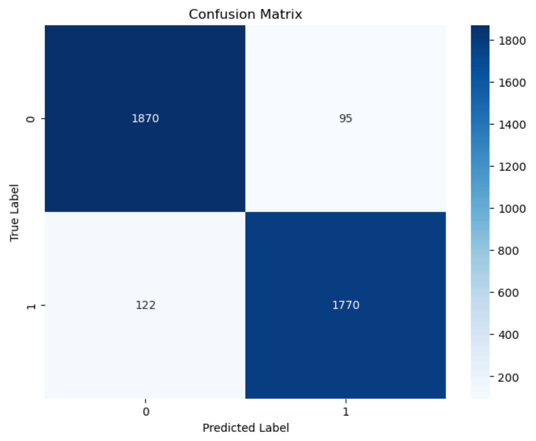

Figure 3: Confusion Matrix of the CNN Model

5.2 CNN+RNN Performance

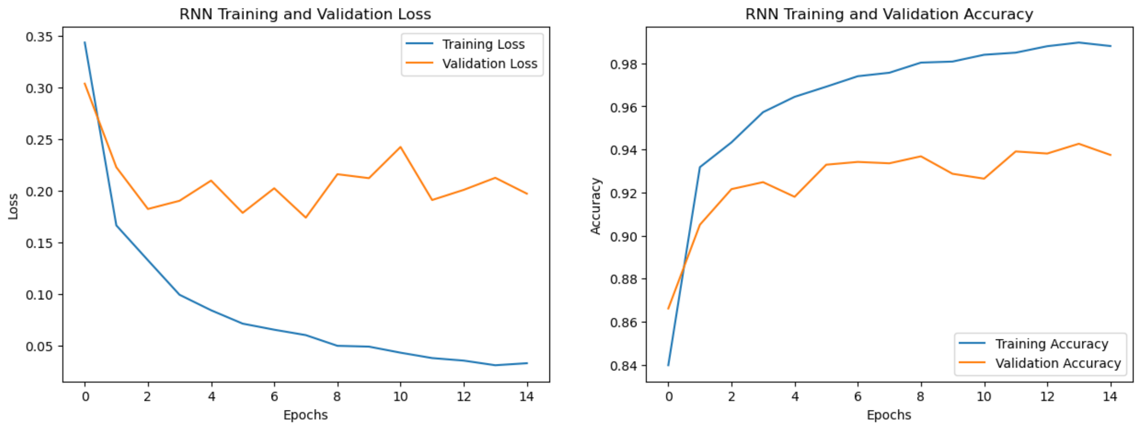

Figure 4: Training and Validation Loss and Accuracy of the CNN+RNN Model

1. **Accuracy: (93.23%)** This is a high accuracy score. This implies that high percentage of images which have been labelled correctly.

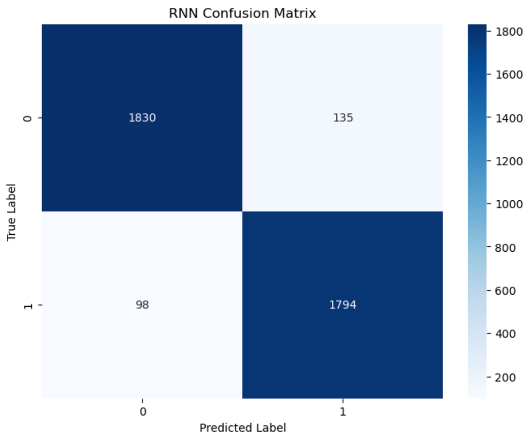

Figure 5: Confusion Matrix of the CNN+RNN Model

2. **Loss: (0.1903%)** This is a low loss value indicating that the predictions are quite close to the actual values.

3. **F1 Score: (0.935)** - F1-score is a measure that combines precision and recall into a single metric, providing a balanced view of a model's performance. 0.93 suggests a good balance between these two metrics. It means the model is not heavily biased towards either precision or recall.

4. **ROC-AUC Score: (0.9867)** An AUC score of approximately 0.9867 is a good score, indicating a very high level of separability between the classes. This means the model is very good at distinguishing between the two classes.

5.3 GAN Performance

Measuring the performance of deep fake detection systems, especially those based on Generative Adversarial Networks (GANs), involves several key metrics and considerations. These performance measurements are crucial to ensure the reliability and effectiveness of these systems in differentiating real from synthetic media.

1. **Accuracy: (88.87%)** This is a high accuracy score. This implies that high percentage of images which have been labelled correctly.

2. **Precision: (0.947%)** This is a low loss value indicating that the predictions are quite close to the actual values.

3. **F1 Score: (0.878)** - F1-score is a measure that combines precision and recall into a single metric, providing a balanced view of a model's performance. 0.93 suggests a

good balance between these two metrics. It means the model is not heavily biased towards either precision or recall.

4. **ROC-AUC Score: (0.953)** An AUC score of approximately 0.953 is a good score, indicating a very high level of separability between the classes. This means the model is very good at distinguishing between the two classes.

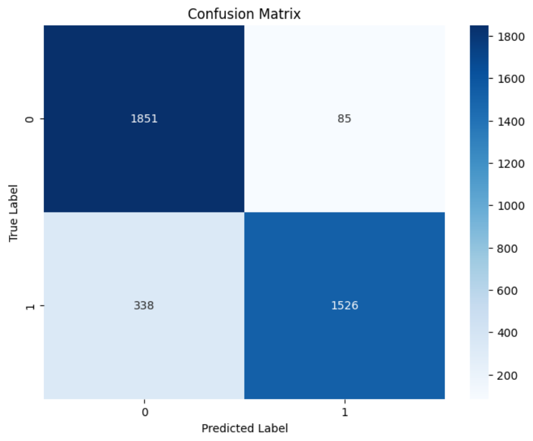

Figure 6: Training and Validation Loss and Accuracy of the GAN Model

6 Conclusions

Our project marks a significant stride in the realm of deepfake detection, achieving an impressive accuracy rate of over 90%. Our experiments underscored the superior efficacy of Convolutional Neural Networks (CNNs) in this domain. The synergistic combination of CNNs with Recurrent Neural Networks (RNNs) also yielded promising results, particularly when harnessing image extraction techniques to highlight visual inconsistencies in deepfakes.

The practical implications of these findings are far-reaching, offering robust tools for combating digital misinformation and enhancing content authenticity. However, we recognize the limitations of our current dataset and model configurations. Future explorations could incorporate a more diverse range of deepfake types and datasets, such as the DFDC dataset, to bolster the robustness of our models. Encoder-decoder techniques, which have shown potential in deepfake detection, merit further investigation.

We are acutely aware of the threat of concept drift, where newer, more sophisticated deepfakes could outpace current detection capabilities. Therefore, experimenting with advanced image extraction techniques, like InceptionV3 and ResNet, could be pivotal in enhancing accuracy. Additionally, expanding our analysis beyond facial features to include other body parts may offer a more holistic approach to detecting deepfakes.

In sum, our work contributes a significant piece to the puzzle of digital content verification. As deepfakes continue to evolve, so must our methods of detection. It is our hope that this research not only advances the field technically but also serves as a catalyst for broader discussions on ethical AI usage and the protection of digital integrity in an increasingly virtual world.

References

[1] L. Jiang, R. Li, W. Wu, C. Qian, and C. C. Loy, "DeeperForensics-1.0: A large-scale dataset for real-world face forgery detection," in *CVPR*, 2020.

[2] V. Jolly, M. Telrandhe, A. Kasat, A. Shitole, and K. Gawande, "Cnn based deep learning model for deepfake detection," *2022 2nd Asian Conference on Innovation in Technology (ASIANCON)*, pp. 1–5, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:252849418

[3] A. Ajoy, C. U. Mahindrakar, D. Gowrish, and V. A, "Deepfake detection using a frame based approach involving cnn," *2021 Third International Conference on Inventive Research in Computing Applications (ICIRCA)*, pp. 1329–1333, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:238242025

[4] R. B. P. and M. S. Nair, "Deepfake detection using multi-path cnn and convolutional attention mechanism," *2022 IEEE 2nd Mysore Sub Section International Conference (MysuruCon)*, pp. 1–6, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:254640427

[5] V.-N. Tran, S.-H. Lee, H.-S. Le, and K.-R. Kwon, "High performance deepfake video detection on cnn-based with attention target-specific regions and manual distillation extraction," *Applied Sciences*, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:238726736

[6] P. Saikia, D. Dholaria, P. Yadav, V. M. Patel, and M. Roy, "A hybrid cnn-lstm model for video deepfake detection by leveraging optical flow features," *2022 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–7, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:251224159

[7] D. GÃ¼era and E. J. Delp, "Deepfake video detection using recurrent neural networks," in *2018 15th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, 2018, pp. 1–6.

[8] Y. Al-Dhabi and S. Zhang, "Deepfake video detection by combining convolutional neural network (cnn) and recurrent neural network (rnn)," in *2021 IEEE International Conference on Computer Science, Artificial Intelligence and Electronic Engineering (CSAIEE)*, 2021, pp. 236–241.

[9] J. Fei, Z. Xia, P. Yu, and F. Xiao, "Exposing ai-generated videos with motion magnification," *Multimedia Tools and Applications*, vol. 80, pp. 30 789–30 802, 2021.

[10] Y. Li, M.-C. Chang, and S. Lyu, "In ictu oculi: Exposing ai generated fake face videos by detecting eye blinking," *arXiv preprint arXiv:1806.02877*, 2018.

[11] D. M. Montserrat, H. Hao, S. K. Yarlagadda, S. Baireddy, R. Shao, J. Horvath, E. Bartusiak, J. Yang, D. Guera, F. Zhu, and E. J. Delp, "Deepfakes detection with automatic face weighting," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2020.

[12] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, pp. 139 – 144, 2014. [Online]. Available: https://api.semanticscholar.org/CorpusID:1033682

[13] T. Chakraborty, U. Reddy, S. M. Naik, M. Panja, and B. Manvitha, "Ten years of generative adversarial nets (gans): A survey of the state-of-the-art," *ArXiv*, vol. abs/2308.16316, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:261396141

[14] A. Dash, J. Ye, and G. Wang, "A review of generative adversarial networks (gans) and its applications in a wide variety of disciplines - from medical to remote sensing," *ArXiv*, vol. abs/2110.01442, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:238259175

[15] S. A. Aduwala, M. Arigala, S. Y. Desai, H. J. Quan, and M. Eirinaki, "Deepfake detection using gan discriminators," *2021 IEEE Seventh International Conference on Big Data Computing Service and Applications (BigDataService)*, pp. 69–77, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:239039052

[16] D. Gong, "Deepfake forensics, an ai-synthesized detection with deep convolutional generative adversarial networks," *International Journal of Advanced Trends in Computer Science and Engineering*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:225790480

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

7 Appendices

includes all source code, scripts, and any material that helps other people to replicate the results.

7.1 Preprocessing Code

The code used for preprocessing the dataset is as follows:

```
# Importing required libraries
import os
from moviepy.editor import VideoFileClip
import cv2
import dlib
import os
import matplotlib.pyplot as plt
import random

# Helper Functions
# Rename the videos to the format 001.mp4, 002.mp4, etc.
def rename_videos(directory):
    # List all files in the directory
    files = os.listdir(directory)
    print("Renaming files in ", directory)
    for file in files:
        # Check if the file is an mp4 file
        if file.endswith('.mp4'):
            # Get the first three characters of the filename
            new_name = file[:3] + '.mp4'
            # Renaming the file
            os.rename(os.path.join(directory, file), os.path.join(directory, new_name))

# Trim the videos to 10 seconds
def trim_videos(folder_path):
    output_folder = 'Dataset/Manipulated/Trimmed/'  # Specify the output folder

    for filename in os.listdir(folder_path):
        print("Trimming", filename, "in", folder_path)
        if filename.endswith('.mp4'):
            input_filepath = os.path.join(folder_path, filename)
            output_filepath = os.path.join(output_folder, filename)  # Save as a new file
            try:
                with VideoFileClip(input_filepath) as video:
                    trimmed_clip = video.subclip(0, 10)  # Trim the first 10 seconds
                    trimmed_clip.write_videofile(output_filepath, codec='libx264')
                    trimmed_clip.close()  # Explicitly close the clip
            except Exception as e:
                print(f"Error processing {filename}: {e}")

# Extract evenly spaced frames from videos
def extract_frames(video_path, output_folder, num_frames=10):
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    # Extract the base filename without extension
    base_filename = os.path.splitext(os.path.basename(video_path))[0]

    vidcap = cv2.VideoCapture(video_path)
    total_frames = int(vidcap.get(cv2.CAP_PROP_FRAME_COUNT))
    interval = total_frames / (num_frames - 1)  # Adjust the interval calculation
    frames_to_capture = [int(interval * i) for i in range(num_frames)]

    count = 0
    frame_number = 1  # Start numbering frames from 1
    success, image = vidcap.read()

    while success:
        if count in frames_to_capture:
            # Use the base filename in the saved file name
            cv2.imwrite(os.path.join(output_folder, f"{base_filename}_{frame_number}.jpg"),
            frame_number += 1  # Increment the frame number
            if len(frames_to_capture) == 0:  # Break once all required frames are captured
                break
            frames_to_capture.remove(count)
        success, image = vidcap.read()
        count += 1

# Extract frames from all videos in a folder
def extract_frames_from_videos(original_videos_path, output_folder):
    for video_file in os.listdir(original_videos_path):
        print("Extracting frames from", video_file)
        video_path = os.path.join(original_videos_path, video_file)
        extract_frames(video_path, output_folder)

# Detect faces in images
def detect_faces(image):
    detector = dlib.get_frontal_face_detector()
    faces = detector(image, 1)
    face_images = []
    for face in faces:
        x, y, w, h = face.left(), face.top(), face.width(), face.height()
        face_img = image[y:y+h, x:x+w]
        face_images.append(face_img)
    return face_images

# Extract faces from all frames in a folder
def extract_faces_from_frames(frames_folder, output_folder):
    if not os.path.exists(output_folder):
        os.makedirs(output_folder)

    for frame_file in os.listdir(frames_folder):
        # Extract the base filename without extension
        base_frame_name = os.path.splitext(os.path.basename(frame_file))[0]

        print("Extracting faces from", frame_file)
        frame_path = os.path.join(frames_folder, frame_file)
        frame = cv2.imread(frame_path)
        faces = detect_faces(frame)
        for i, face in enumerate(faces):
            if face is not None and not face.size == 0:
                cv2.imwrite(os.path.join(output_folder, f"{base_frame_name}_{i}.jpg"), face)
            else:
                print(f"No face detected in {frame_file} or face is empty.")

# Load images from a folder
def load_faces(folder_path, X, y, label, target_size=(224, 224)):
    for filename in os.listdir(folder_path):
        print("Loading", filename)
        if filename.endswith('.jpg'):
            img_path = os.path.join(folder_path, filename)
            img = cv2.imread(img_path)
            if img is not None:
                img_resized = cv2.resize(img, target_size)
                X.append(img_resized)
                y.append(label)
            else:
                print(f"Failed to load image: {img_path}")
    return X, y

# Delete the temporary folders
def cleanup():
    print("Cleaning up temporary folders")
    os.system("rm -rf Dataset/Frames/Original/*")
    os.system("rm -rf Dataset/Frames/Manipulated/*")
    os.system("rm -rf Dataset/Faces/Original/*")
    os.system("rm -rf Dataset/Faces/Manipulated/*")

# Preprocessing Main Code
original_videos_path = 'Dataset/Original/'
deepfake_videos_path = 'Dataset/Manipulated/'
X_file_path = 'Dataset/X_data.npy'
Y_file_path = 'Dataset/Y_data.npy'
X = []
y = []

# Rename the videos
rename_videos('Dataset/Original/')
rename_videos('Dataset/Manipulated/')

# Trim the videos
trim_videos('Dataset/Original/')
trim_videos('Dataset/Manipulated/')

# Extract frames from the videos
extract_frames_from_videos(original_videos_path, 'Dataset/Frames/Original')
extract_frames_from_videos(deepfake_videos_path, 'Dataset/Frames/Manipulated')

# Extract faces from the frames
extract_faces_from_frames('Dataset/Frames/Original', 'Dataset/Faces/Original')
extract_faces_from_frames('Dataset/Frames/Manipulated', 'Dataset/Faces/Manipulated')

# Load the images
X, y = load_faces('Dataset/Faces/Original', X, y, 0)
X, y = load_faces('Dataset/Faces/Manipulated', X, y, 1)
```

```
# Convert lists to numpy arrays
X = np.array(X)
y = np.array(y)

# Save the dataset
np.save(X_file_path, X)
np.save(Y_file_path, y)

# Cleanup
cleanup()

# Preprocessing Example Code
def show_images(images, titles, cols=5):
    rows = (len(images) + cols - 1) // cols
    plt.figure(figsize=(15, rows * 3))

    for i, image in enumerate(images):
        plt.subplot(rows, cols, i + 1)
        plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
        plt.title(titles[i] if i < len(titles) else '')
        plt.xticks([]), plt.yticks([])

    plt.show()

def random_video_display(original_videos_path, deepfake_videos_path):
    rename_videos(original_videos_path)
    rename_videos(deepfake_videos_path)
    random_video = random.choice(os.listdir(original_videos_path))
    print("Randomly selected video: ", random_video)
    original_video_path = os.path.join(original_videos_path, random_video)
    deepfake_video_path = os.path.join(deepfake_videos_path, random_video)

    # Display original frames
    extract_frames(original_video_path, 'Dataset/Temporary/Original')
    original_frames = [cv2.imread(os.path.join('Dataset/Temporary/Original', f)) for f in os
    show_images(original_frames, ['Original Frame'] * len(original_frames), 5)

    # Display faces from original frames
    original_faces = []
    for frame in original_frames:
        original_faces.extend(detect_faces(frame))
    show_images(original_faces, ['Original Face'] * len(original_faces), 5)

    # Display deepfake frames
    extract_frames(deepfake_video_path, 'Dataset/Temporary/Manipulated')
    deepfake_frames = [cv2.imread(os.path.join('Dataset/Temporary/Manipulated', f)) for f in
    show_images(deepfake_frames, ['Deepfake Frame'] * len(deepfake_frames), 5)

    # Display faces from deepfake frames
    deepfake_faces = []
    for frame in deepfake_frames:
        deepfake_faces.extend(detect_faces(frame))
    show_images(deepfake_faces, ['Deepfake Face'] * len(deepfake_faces), 5)

    # Cleanup
    os.system("rm -rf Dataset/Temporary/*")

original_videos_path = 'Dataset/Original/'
deepfake_videos_path = 'Dataset/Manipulated/'
random_video_display(original_videos_path, deepfake_videos_path)
```

7.2 CNN Code

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(224, 224, 3), padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Conv2D(128, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    Flatten(),
    Dense(128, activation='relu'),
    BatchNormalization(),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])

# Optimizer with adjusted learning rate
optimizer = Adam(learning_rate=0.0001)

# Callbacks
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.2, patience=5, min_lr=0.00001)
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
```

7.3 CNN+RNN Code

```
# Load pre-trained VGG16 model without the classification layer
VGG_model = VGG16(weights='imagenet', include_top=False)

# Build the model
model = Sequential([
    keras.layers.InputLayer(input_shape=(49, 512)),
    LSTM(64, return_sequences=True),
    LSTM(32),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Function to extract features from images
def predict_batch(batch):
    return VGG_model.predict(batch)

print("Extracting features...")
# Splitting data into batches
batch_size = 1000
batches = [X[i:i + batch_size] for i in range(0, len(X), batch_size)]

# Using ThreadPoolExecutor for parallel execution
with ThreadPoolExecutor(max_workers=5) as executor:
    # Extracting features from images
    results = list(executor.map(predict_batch, batches))
```

```
# Combining the results into a single feature array
features = np.concatenate(results)
features_reshaped = features.reshape(features.shape[0], 49, 512)
```

7.4 GAN Code

```
def make_generator_model():
    model = Sequential()

    # Start with a dense layer that takes a random noise vector as input
    model.add(Dense(7*7*256, use_bias=False, input_shape=(100,)))  # 100 is the dimensionali
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    model.add(Reshape((7, 7, 256)))

    # Upsample to 14x14
    model.add(Conv2DTranspose(128, kernel_size=5, strides=2, padding='same', use_bias=False)
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    # Upsample to 28x28
    model.add(Conv2DTranspose(64, kernel_size=5, strides=2, padding='same', use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    # Upsample to 56x56
    model.add(Conv2DTranspose(64, kernel_size=5, strides=2, padding='same', use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    # Upsample to 112x112
    model.add(Conv2DTranspose(32, kernel_size=5, strides=2, padding='same', use_bias=False))
    model.add(BatchNormalization())
    model.add(LeakyReLU())

    # Final Upsample to 224x224, with 3 channels for the RGB image
    model.add(Conv2DTranspose(3, kernel_size=5, strides=2, padding='same', use_bias=False, a

    return model
```

```
# Create the generator model
generator = make_generator_model()

def make_discriminator_model():
    model = Sequential()

    # First convolutional layer
    model.add(Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[224, 224, 3]))
    model.add(LeakyReLU())
    model.add(Dropout(0.3))

    # Second convolutional layer
    model.add(Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(LeakyReLU())
    model.add(Dropout(0.3))

    # Third convolutional layer
    model.add(Conv2D(256, (5, 5), strides=(2, 2), padding='same'))
    model.add(LeakyReLU())
    model.add(Dropout(0.3))

    # Flatten the output
    model.add(Flatten())

    # Dense output layer with a single neuron for binary classification
    model.add(Dense(1))

    return model

# Create the discriminator model
discriminator = make_discriminator_model()

# This method returns a helper function to compute cross entropy loss
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)

def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss
```

```
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)

generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)

def train_step(images, labels):
    noise_dim = 100
    batch_size = images.shape[0]

    # Generating noise from a normal distribution
    noise = tf.random.normal([batch_size, noise_dim])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)

        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output, labels)
        #print("Discriminator Loss:"+ str(disc_loss))

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_varia

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_vari
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.tr

def discriminator_loss(real_output, fake_output, labels):
    real_loss = cross_entropy(labels, real_output)  # Using labels here
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def train(train_dataset, epochs):
    for epoch in range(epochs):
        print("Epoch "+str(epoch))
        for images, labels in train_dataset.take(10000):
            train_step(images, labels)
```

```
# Train the model
epochs = 10
train(train_dataset, epochs)

def discriminator_evaluate(discriminator, images, labels):
    # Real images
    predictions = discriminator(images, training=False)
    roc_curve_val = roc_auc_score(labels, predictions)
    #print(print('ROC AUC score:', roc_auc_score(labels, predictions)))
    #real_accuracy = np.mean(predictions >= 0.5)  # Assuming 0.5 as the threshold for classi
    binary_labels = (predictions >= 0.5)
    # # Fake images
    # fake_predictions = discriminator(fake_images, training=False)
    # fake_accuracy = np.mean(fake_predictions < 0.5)
    accuracy = np.mean(binary_labels == labels)
    return (accuracy,roc_curve_val,binary_labels)

real_accuracy_list = []
sum = 0
roc_auc_sum =0
binary_label_total =[]
for i in range(19):
  acc,roc_val,binary_labels = discriminator_evaluate(discriminator, features_test[200*i:200*
  sum+=acc
  roc_auc_sum+=roc_val
  binary_label_total.extend(np.squeeze(np.array(binary_labels)))
binary_label_total = [int(value) for value in binary_label_total]
```