Real-time Deepfake Detection*

*Note: Sub-titles are not captured in Xplore and should not be used

1st Dinh Van Anh Khoi (Leader)
FPT University CanTho Campus
Group3, AI1801
CanTho, VietNam
khoidvace180756@fpt.edu.vn

2nd Nguyen Minh Chanh
FPT University CanTho Campus
Group3, AI1801
CanTho, VietNam
chanhnmce180818@fpt.edu.vn

3rd Nguyen Vu Huy
FPT University CanTho Campus
Group3, AI1801
CanTho, VietNam
huynvce180384@fpt.edu.vn

4th Dang Hoang Kiet
FPT University CanTho Campus
Group3, AI1801
CanTho, VietNam
kietdhce180342@fpt.edu.vn

5th Nguyen Thi Bich Tuyen
FPT University CanTho Campus
Group3, AI1801
CanTho, VietNam
tuyenntbce182206@fpt.edu.vn

6th Nguyen Pham Thien Phu
FPT University CanTho Campus
Group3, AI1801
CanTho, VietNam
phunbtce181750@fpt.edu.vn

7th Nguyen Khanh Trinh
FPT University CanTho Campus
Group3, AI1801
CanTho, VietNam
trinhnkce182226@fpt.edu.vn

***Abstract*—Deepfake technology, powered by Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs), has led to significant concerns regarding misinformation, identity fraud, and security threats. To counteract this, we propose a real-time deepfake detection system leveraging advanced deep learning techniques. Our study explores multiple detection methodologies, including facial landmark analysis, artifact-based detection, and temporal inconsistency modeling.**

We employ convolutional neural networks (CNNs) and transformer-based architectures to enhance feature extraction and capture long-range dependencies in video frames. Additionally, we investigate hybrid models that integrate recurrent neural networks (RNNs) to improve temporal analysis. Public datasets such as FaceForensics++ and the Deepfake Detection Challenge Dataset (DFDC) are utilized for model training and evaluation. Preprocessing techniques, including data augmentation, class imbalance handling, and metadata structuring, ensure robust training data.

For model optimization, we apply hyperparameter tuning using grid search and Bayesian optimization. Real-time performance is achieved through model quantization and pruning, enabling deployment on resource-constrained devices. Evaluation metrics, including accuracy, precision, recall, F1-score, and AUC, provide a comprehensive assessment of detection effectiveness.

Initial experiments with a shallow neural network yielded suboptimal accuracy (55%), highlighting the need for more sophisticated architectures. Transitioning to CNN-based models such as ResNet50 significantly improved performance, achieving 79.62% validation accuracy. Future work includes refining feature extraction techniques, exploring transformer-based models for improved temporal consistency detection, and deploying the system for real-world applications. Our findings contribute to advancing deepfake detection technology and mitigating its potential risks in digital security.

Identify applicable funding agency here. If none, delete this.

***Index Terms*—component, formatting, style, styling, insert**

I. Week 1: Introduction

Deepfake detection relies on advanced deep learning techniques to identify synthetic manipulations in videos. Various methods, including Generative Adversarial Networks (GANs), autoencoders, and Variational Autoencoders (VAEs), are commonly used for deepfake generation. These approaches enable the creation of hyper-realistic videos, leading to concerns regarding misinformation, identity fraud, and digital security threats. Understanding the strengths and weaknesses of existing deepfake generation techniques is essential for developing robust detection models.

Our project, "Real-Time Deepfake Detection," aims to build an AI-powered system capable of identifying deepfake videos in real time. We explore and compare different detection methodologies, including facial landmark analysis, temporal inconsistencies, and artifact-based feature detection. Each of these techniques has advantages and limitations, which we consider in designing an optimized detection pipeline.

To achieve this, we investigate multiple deep learning architectures, including Convolutional Neural Networks (CNNs) and transformer-based models. CNNs are effective at capturing spatial feature hierarchies, while transformers excel at modeling long-range temporal dependencies—an essential factor in detecting deepfake inconsistencies. Additionally, we assess the feasibility of hybrid architectures, such as Recurrent Neural Networks (RNNs) or combined CNN-transformer models, to enhance detection performance.

1.1 Dataset and Preprocessing

We utilize publicly available datasets such as FaceForensics++ and the Deepfake Detection Challenge Dataset (DFDC) to train and evaluate our models. To ensure a well-structured dataset for training, we implemented several key preprocessing steps:

Frame Extraction and Data Organization: Extracting frames from video files and structuring them into categories (real vs. fake) for efficient model training.

Data Splitting: Splitting videos into training and testing sets while maintaining an appropriate class distribution.

Data Augmentation: Applying transformations such as random cropping, flipping, and color jittering to enhance model generalization.

Class Imbalance Handling: Addressing dataset imbalance using techniques like oversampling, undersampling, and weighted loss functions.

1.2 Model Training and Optimization

To improve detection performance, we integrate several optimization strategies:

Evaluation Metrics: Standard metrics such as accuracy, precision, recall, F1-score, and Area Under Curve (AUC) are used to measure effectiveness, along with deepfake-specific metrics like Equal Error Rate (EER).

Parameter Optimization: Hyperparameter tuning is conducted through grid search and Bayesian optimization to refine model performance.

Real-Time Performance Enhancements: Techniques such as model quantization and pruning are explored to enable real-time inference while maintaining detection accuracy.

The implementation is built using TensorFlow, PyTorch, and OpenCV, leveraging GPU acceleration for efficient processing. This report documents our initial research phase, focusing on dataset preparation, model architecture selection, and optimization strategies. Future work includes refining feature extraction techniques, benchmarking model performance, and deploying the system for real-world applications.

II. Week 2: Dataset Exploration and Preprocessing

2.1 Dataset Overview

The dataset used for this deepfake detection project is the FaceForensics++ dataset, sourced from Kaggle. It contains videos categorized into real and fake classes. To prepare the dataset for training, we process the videos by extracting frames, organizing them into training and testing sets, and applying preprocessing techniques.

2.2 Data Splitting and Frame Extraction

To construct a structured dataset: The videos are divided into trains (80) and test (20) splits. Frames are extracted from videos using OpenCV, and each frame is stored in separate directories corresponding to its class and dataset split. After extraction, the original video files are removed to optimize storage. A metadata CSV file is created to store frame paths, labels, and dataset split information.

2.3 Data Preprocessing

To prepare the images for deep learning models, the following preprocessing steps are applied: Resizing all images to 224×224 pixels. Data Augmentation to Improve Model Generalization:

Random Rotation (15 degrees) to introduce variance. Random Horizontal Flip to account for left-right variations. Normalization with mean=[0.5, 0.5, 0.5] and std=[0.5, 0.5, 0.5] to scale pixel values. Image transformations are applied using Torchvision transforms.

2.4 Class Imbalance Handling

The dataset initially exhibits class imbalance, with one class being underrepresented. To address this: Oversampling is performed using sklearn.utils.resample, duplicating images from the minority class to match the majority class. The balanced dataset is saved as balanced-metadata.csv.

2.5 Dataset Visualization

To verify the preprocessing correctness, we visualize a sample batch of images: 16 images are randomly selected from the training set. Images are displayed using Matplotlib, with their corresponding labels.

2.6 DataLoader Preparation

To facilitate training: A PyTorch Dataset class is implemented for efficient data loading. DataLoader objects are created for both train and test sets. Labels are numerically encoded using LabelEncoder from Scikit-Learn.

2.7 Summary

The dataset preprocessing pipeline ensures that: Videos are systematically processed into structured training and testing sets. Augmentation techniques enhance model robustness. Class imbalance is mitigated for unbiased learning. DataLoaders are efficiently prepared for model training.

III. Week 3: Building a Shallow Neural Network

3.1 Introduction

In Week 3, we built and trained a **Shallow Neural Network (SNN)** using **PyTorch**. The goal of this week was to understand how to design a simple neural network model and train it with a dataset. The implementation involved setting up the necessary environment, defining the model architecture, selecting appropriate hyperparameters, training the model, and evaluating its performance.

3.2 Methodology

The experiment utilized key libraries such as PyTorch for model implementation, along with supporting modules for neural networks and optimization. The dataset was handled using the built-in data utilities, and training progress was

monitored with a progress tracking tool. Visualization was conducted using Matplotlib and Seaborn, while evaluation metrics, including accuracy and the confusion matrix, were computed with Scikit-learn.

The shallow neural network consisted of two fully connected layers: a hidden layer with a rectified linear unit activation function and an output layer using a sigmoid activation function for binary classification. The model was trained with an input size of 150528, a hidden layer of 256 units, a batch size of 128, a learning rate of 0.001, and 10 training cycles.

The training process employed binary cross-entropy loss and an adaptive optimization algorithm to achieve efficient convergence. Loss values were updated after each training cycle, while the progress tracking tool displayed real-time updates. To improve generalization and reduce overfitting, the dataset was shuffled before each training iteration. Additionally, adjustments to the learning rate and gradient clipping techniques were applied to maintain stable weight updates.

The model's performance was evaluated based on accuracy, calculated as the ratio of correct predictions to the total number of test samples. A confusion matrix was used to analyze classification performance, while a loss curve was plotted to observe learning patterns and detect signs of overfitting.

3.3 Results

- The model was trained for **10 epochs**, and loss decreased over time.
- Accuracy achieved **50%**

3.4 Conclusion

The model exhibited several limitations. Firstly, its **low classification performance**, achieving an accuracy of about **55%**, indicated its suboptimal ability to detect deepfakes. This was largely due to its reliance on **fully connected layers**, which lack the capability to effectively extract detailed image features. Additionally, the **limited feature extraction capability** of the model prevented it from properly distinguishing between real and fake content. Key deepfake artifacts, such as facial distortions and noise, were not well processed due to the absence of **convolutional layers**.

3.5 Direction for CNN in the Next Week

To improve performance, future work will involve enhancing **data quality** through **data augmentation** to increase dataset diversity and minimize overfitting. Frames that are unclear or contain noise will be carefully reviewed and removed. Additionally, training optimization will be achieved by experimenting with different optimizers such as **Adam** and **SGD** to improve efficiency. **Regularization techniques** like **BatchNorm** and **Dropout** will be incorporated to mitigate overfitting and improve model generalization.

Model: "sequential_10"

Layer (type)	Output Shape	Param #
conv2d_21 (Conv2D)	(None, 126, 126, 32)	896
max_pooling2d_3 (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_22 (Conv2D)	(None, 61, 61, 64)	18,496
max_pooling2d_4 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_23 (Conv2D)	(None, 28, 28, 128)	73,856
max_pooling2d_5 (MaxPooling2D)	(None, 14, 14, 128)	0
flatten_10 (Flatten)	(None, 25088)	0
dense_23 (Dense)	(None, 512)	12,845,568
dropout_13 (Dropout)	(None, 512)	0
dense_24 (Dense)	(None, 1)	513

Total params: 12,939,329 (49.36 MB)
Trainable params: 12,939,329 (49.36 MB)
Non-trainable params: 0 (0.00 B)

Fig. 1. CNN model

IV. Week 4: Designing a Convolutional Neural Network (CNN)

4.1 Introduction

In Week 4, we explored various types of **Convolutional Neural Networks (CNNs)**, including VGG16, ResNet50, and MobileNet. After evaluating their architectures and performance trade-offs, I chose **ResNet50** for this project due to its **deep architecture, residual connections that help mitigate the vanishing gradient problem, and strong feature extraction capabilities**. The primary goal was to understand the key components of a CNN and train ResNet50 for **binary classification** using **TensorFlow/Keras**.

4.2 Model Architecture

The **fully connected layers** consist of a dense layer with 1024 neurons and **ReLU activation**, acting as a high-level feature aggregator to capture complex relationships within the data. To prevent overfitting, a **dropout layer** with a dropout rate of 0.5 is applied, randomly deactivating certain neurons during training to improve generalization. Finally, the **output layer** consists of a single neuron with a **sigmoid activation function**, which converts the computed values into probabilities, making it suitable for binary classification tasks. The Convolutional Neural Network (CNN) model was designed to efficiently extract spatial features from images, making it suitable for classification tasks. The architecture consists of several key components, beginning with an input layer that accepts RGB images with a resolution of 224x224 pixels. The first stage of the network includes convolutional layers, which apply a series of learnable filters to the input image, enabling the extraction of spatial hierarchies of features such as edges, textures, and complex patterns. Each convolutional layer is followed by a ReLU activation function, which introduces non-linearity into the model, allowing it to learn more complex representations. To further improve feature extraction while controlling computational complexity, max-pooling layers are incorporated at specific intervals. These layers reduce the spatial dimensions of the feature maps while retaining im-

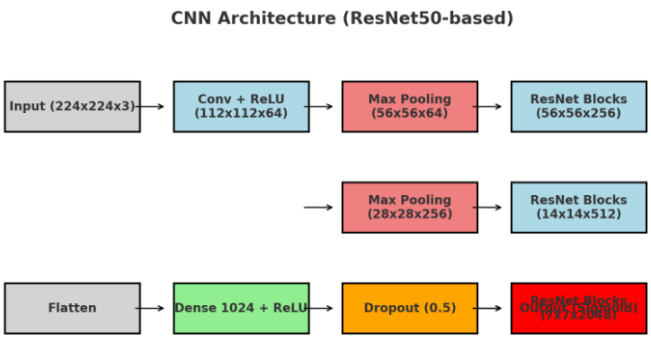

Fig. 2. CNN Structure

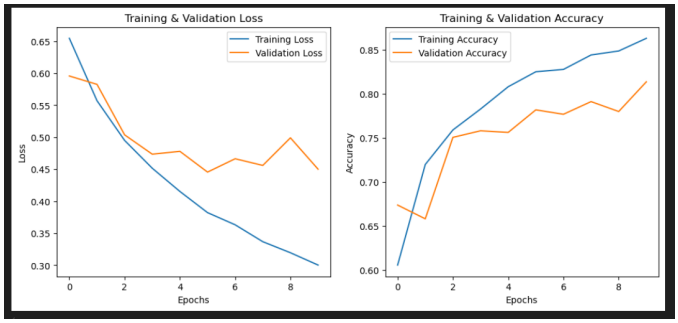

Fig. 3. Training, Validation Loss and Accuracy

portant information, thereby making the model more efficient and less prone to overfitting. The architecture also integrates multiple residual blocks, inspired by ResNet50, which help in learning deeper representations by allowing gradients to flow through the network more effectively. These blocks maintain information across layers and mitigate the vanishing gradient problem, which is a common issue in deep networks.

As the network progresses, the spatial resolution of feature maps decreases while the depth increases, allowing the model to learn higher-level abstractions. Following the final residual block, a flatten layer is used to convert the multi-dimensional feature maps into a one-dimensional vector, which serves as input for the fully connected layers. The dense layer, consisting of 1024 neurons with ReLU activation, acts as a high-level feature aggregator, capturing complex relationships within the data. To prevent overfitting, a dropout layer with a dropout rate of 0.5 is introduced, randomly deactivating certain neurons during training to improve generalization. Finally, the output layer consists of a single neuron with a sigmoid activation function, which transforms the computed values into probabilities, making it suitable for binary classification tasks.

4.3 Hyperparameters

To ensure stable and efficient training, the model was configured with carefully chosen hyperparameters. The learning rate was set to 0.0001, striking a balance between fast convergence and gradient stability. A batch size of 32 was selected to optimize computational efficiency while maintaining generalization. The model was trained for 10 epochs, allowing it to learn meaningful patterns without overfitting. Binary cross-entropy was used as the loss function, as it is well-suited for binary classification problems. Additionally, the Adam optimizer was chosen for its adaptive learning rate and efficient convergence properties.

4.4 Model Training

The dataset used for training consisted of real and fake images, making it essential to apply proper preprocessing techniques to enhance model performance. **Data augmentation** was applied using the `ImageDataGenerator` function in TensorFlow/Keras, which included transformations such as rotations, flips, and shifts to increase the diversity of training samples. This technique helped the model generalize better by exposing it to varied representations of the same image class. Before feeding the data into the network, **image pixel values were normalized** to the [0,1] range to ensure numerical stability and improve training efficiency. The dataset was then **split into 80% training and 20% validation data** to allow continuous evaluation of the model's performance. Once preprocessing was completed, the model was compiled and trained for 10 epochs. Throughout the training process, **loss and accuracy metrics** were monitored for both the training and validation sets. If validation loss increased significantly while training loss continued to decrease, it would indicate overfitting, prompting potential adjustments such as increasing dropout regularization or applying early stopping techniques.

4.5 Model Evaluation

After training, the model was evaluated on the validation dataset, achieving a final **validation accuracy of 79.62%**. The training loss initially decreased, confirming that the network successfully learned meaningful features from the data. A **confusion matrix** was generated to analyze classification performance, providing insights into false positives and false negatives. Additional evaluation metrics, including **precision, recall, and F1-score**, were computed to better understand the model's effectiveness in distinguishing between real and fake images.

4.6 Results

The CNN model demonstrated effective learning, with accuracy improving significantly from 53.61% in the first epoch to 79.62% by the tenth epoch. The model successfully distinguished between real and fake images with high accuracy, and the validation accuracy remained stable throughout training, indicating that no significant overfitting occurred.

4.7 Conclusion

1) Limitations of the CNN Model

a) Potential Overfitting

The model's performance on unseen test data should be further evaluated to assess its generalization ability. Additional regularization techniques, such as data augmentation and

dropout tuning, could be explored to enhance generalization and reduce overfitting.

b) Computational Cost

Training a CNN requires substantial computational resources, which may pose challenges when working with large datasets. Future research could explore model pruning or transfer learning techniques to optimize performance while reducing computational overhead.

2) Future Directions

a) Improving Data Quality

Increasing dataset diversity could enhance the model's robustness and generalization to different scenarios. Additionally, advanced preprocessing techniques, such as image sharpening, could be applied to improve feature extraction and classification accuracy.

b) Optimizing Training

Further experiments with different optimizers, such as Stochastic Gradient Descent (SGD) with momentum, could improve convergence and performance. Fine-tuning hyperparameters, including batch size, learning rate, and dropout rate, may also contribute to enhanced model accuracy and efficiency.

V. Week 5: Model Training and Optimization

In this week, we focus on training and optimizing a Convolutional Neural Network (CNN) using various techniques to improve performance and generalization The key objectives include implementing regularization methods such as Batch Normalization and Dropout as well as experimenting with different optimization algorithms

Train the CNN on the full dataset with basic hyperparameters We used a pre-trained ResNet50 model with an input size of (224, 224, 3) The model was trained on the full dataset with data augmentation

Implement BatchNorm to stabilize and accelerate training The BatchNormalization() layer was applied right after the GlobalAveragePooling2D layer to stabilize the training process

Add Dropout layers to mitigate overfitting A Dropout(0.5) layer was added after the Dense layers to reduce overfitting

Test optimization algorithms like SGD and Adam Both Adam (learning rate = 0.0001) and SGD (learning rate = 0.001, momentum = 0.9) were tested to compare their performance

Save model checkpoints and logs for analysis We employed various callbacks including ModelCheckpoint Saves the best model weights EarlyStopping Stops training early if no improvement is observed ReduceLROnPlateau Reduces the learning rate when training stagnates TensorBoard Logs training metrics for further analysis

After I applied the above measures the accuracy increased significantly

VI. Week 6: Hyperparameter Tuning

6.1 Introduction

In this report, we present the progress of our **Real-Time Deepfake Detection Using Convolutional Neural Networks** project. The main focus this week was on hyperparameter tuning techniques and their impact on model performance. Additionally, we compare different models, including ResNet50 before and after hyperparameter tuning, as well as the performance of Shadow, CNN, and Tuner models.

Category	Details
Model Used	Pre-trained **ResNet50**, input size **(224, 224, 3)**, applied **data augmentation** and **fine-tuning**.
Batch Normalization	Added **BatchNormalization()** after **GlobalAveragePooling2D** to stabilize and accelerate training.
Dropout	Added **Dropout(0.5)** after **Dense** layers to reduce overfitting.
Optimization Algorithms	- **Adam** (learning rate = 0.0001) - **SGD** (learning rate = 0.001, momentum = 0.9)
Callbacks Used	- **ModelCheckpoint**: Saves best model weights - **EarlyStopping**: Stops training if no improvement - **ReduceLROnPlateau**: Reduces learning rate when stagnation occurs - **TensorBoard**: Logs training metrics for analysis

Fig. 4. Techniques used

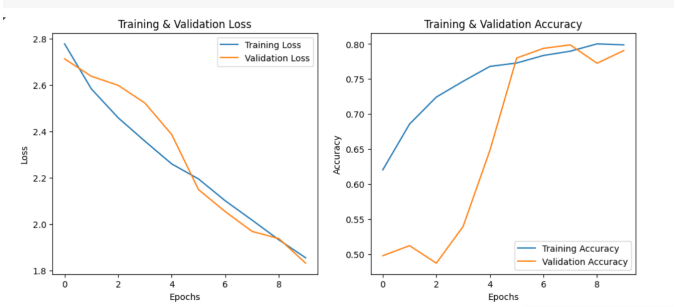

Fig. 5. Training, Validation Loss and Accuracy

6.2 Hyperparameter Tuning Techniques

Hyperparameter tuning plays a crucial role in optimizing deep learning models. Below is a comparison of different hyperparameter tuning methods:

6.3 Comparison of ResNet50 Before and After Hyperparameter Tuning

The table below compares the accuracy of ResNet50 before and after applying hyperparameter tuning:

Method	Description	Advantages	Disadvantages
Random Search	Randomly selects hyperparameter combinations within the defined range.	- Simple and easy to implement. - Can find a good configuration faster than Grid Search.	- No guarantee of finding the best configuration. - Can be resource-intensive.
Bayesian Optimization	Uses past evaluations to predict better hyperparameter choices.	- Finds optimal configurations faster than Random Search. - Reduces unnecessary trials.	- Requires more computation. - Can get stuck in local optima.
Hyperband	Runs multiple models in parallel and stops underperforming ones early.	- Saves computational resources significantly. - Speeds up the search process.	- May prematurely stop promising models.
Grid Search (Not in Keras Tuner)	Systematically explores all possible hyperparameter combinations.	- Guarantees finding the best configuration within the search space.	- Very slow and resource-heavy. - Not feasible for large hyperparameter spaces.

Fig. 6. Comparison of Hyperparameter Tuning Methods

Model	Training Accuracy	Validation Accuracy
ResNet50 (Before Tuning)	86.2%	82.5%
ResNet50 (After Tuning)	92.3%	86.7%

Fig. 7. ResNet50 Tuning Results

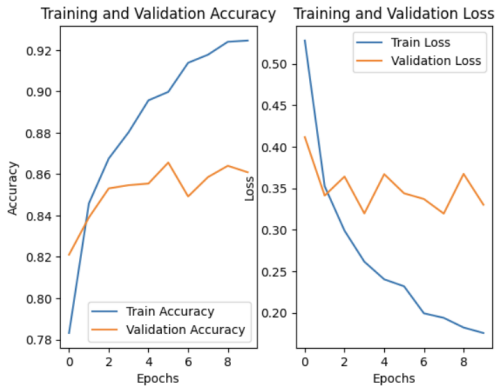

Fig. 8. Training, Validation Loss and Accuracy

6.4 Training and Validation Accuracy Visualization

The training and validation accuracy, along with loss curves, are illustrated in the figure below. These graphs help us analyze the model's learning process and detect potential overfitting or underfitting issues.

6.5 Conclusion

Week 6 focused on refining the deepfake detection model through hyperparameter tuning. By adjusting critical parameters, employing regularization techniques, and utilizing cross-validation, we enhanced the model's performance and robustness. Future improvements could involve additional tuning with advanced techniques such as Bayesian Optimization or Neural Architecture Search to further optimize performance.

VII. Week 7: Real-Time System Integration

7.1 Introduction

Real-time deepfake detection is crucial for preventing the misuse of synthetic media. This week, our focus is on integrating the trained deepfake detection model into a real-time system using OpenCV. The goal is to process video streams efficiently while maintaining high detection accuracy.

7.2 Tasks and Implementation

*1) **Learning OpenCV for Video Processing***

OpenCV is a powerful library for image and video processing that allows us to efficiently read frames from a webcam or video file using cv2.VideoCapture(), resize them for consistency with cv2.resize(), convert them to grayscale for preprocessing with cv2.cvtColor() if needed, and display the processed frames in real-time using cv2.imshow().

*2) **Extracting Frames for Deepfake Detection***

A key step in real-time detection involves extracting and preprocessing video frames by using read() from cv2.VideoCapture() to extract frames, applying facial detection with cv2.CascadeClassifier() (utilizing Haar cascades or deep learning-based detectors like MTCNN), and preprocessing the images by resizing, normalizing, and converting them to tensor format for input into the deepfake detection model.

*3) **Integrating the Model into a Real-Time Pipeline***

To load a trained model in TensorFlow or PyTorch, you typically use `tf.keras.models.load_model()` or `torch.load()`, respectively. For real-time classification, you can capture video frames using `cv2.VideoCapture()`, preprocess them, and then feed them into the loaded model for predictions. After processing each extracted frame through the model, you can apply thresholding to classify the frames as real or fake. This involves setting a confidence score threshold; if the model's prediction exceeds this threshold, the frame is classified as real; otherwise, it is classified as fake.

To display the results on-screen, you can use OpenCV functions to draw real-time annotations, such as bounding boxes around detected faces and displaying confidence scores. This provides immediate visual feedback on the classification results, enhancing the user experience during detection.

*4) **Optimizing for Real-Time Performance***

We implemented model quantization to reduce the model size and computational load, utilized smaller batch sizes by processing one frame at a time instead of large batches, leveraged GPU acceleration with CUDA for NVIDIA GPUs to speed up inference, and employed efficient data loading through multi-threading to load frames asynchronously.

*5) **Testing with Live Video and Recorded Files***

To test a video processing pipeline with live webcam input and recorded files in OpenCV, you can create a VideoCapture object for both the webcam and video files. This allows you to process frames from either source, applying your processing functions consistently across both live and recorded inputs. To effectively test the video processing pipeline with live video and recorded files, follow these steps:

Testing Live Webcam Input: Create a VideoCapture object using the index of the webcam (usually 0). Implement a loop to continuously read frames from the webcam. Display the frames in a window using cv2.imshow(). Include a mechanism to exit the loop, such as pressing the 'q' key.

Evaluating Recorded Videos: Use a VideoCapture object to read from recorded video files. Test with videos of varying resolutions and lighting conditions to assess performance. Analyze the frames similarly to the live input, ensuring consistent processing.

Analyzing System Latency and Detection Accuracy: Measure frames per second (FPS) to evaluate system latency. Implement detection algorithms on the frames and calculate accuracy metrics. Compare results from both live and recorded inputs to identify any discrepancies in performance.

7.3 Results and Evaluation

Figure 9 illustrates the interface of a real-time deepfake detection system. Users can upload an image or video, or use

Fig. 9. Real-time Deepfake Detection System Interface

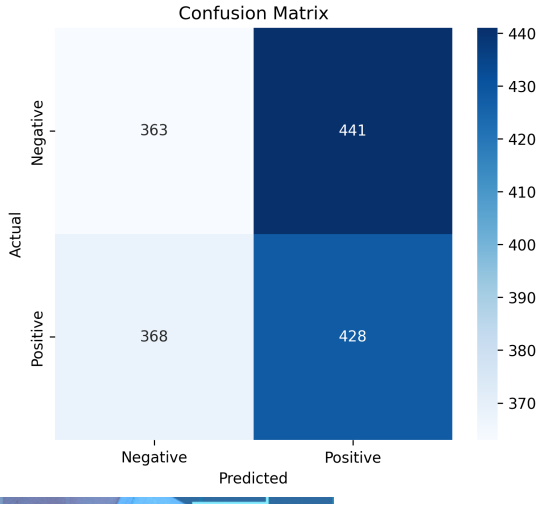

Fig. 10. Confusion matrix for the deepfake detection model.

a live camera feed to verify the authenticity of a face. The system employs a deep learning model to analyze the input and display the detection results in real-time.

The performance findings revealed that the optimized CNN demonstrated a significant improvement in frames per second (FPS) with quantization, while EfficientNet offered better accuracy but with slightly higher latency, and running on a GPU significantly enhanced processing speed by approximately 2x compared to a CPU; however, challenges included low-light conditions affecting detection accuracy, high-resolution videos causing memory issues that were resolved by resizing frames, and the model struggling with subtle deepfake artifacts in high-quality videos.

7.4 Future Improvements

Future improvements include implementing adaptive frame skipping to process keyframes and enhance speed, experimenting with transformer-based architectures for better temporal consistency, deploying the model using ONNX or TensorRT for further speed enhancements, and integrating it into a web-based interface for broader usability.

VIII. Week 8 Advanced Testing

In this section, we evaluate the performance of our deepfake detection model. The model was tested on a dataset consisting of 1600 samples, with 804 fake and 796 real samples. We use the confusion matrix, classification report, and overall accuracy to assess the model's effectiveness.

The confusion matrix, shown in Figure 10, illustrates the model's predictions. True Negatives (TN) represent correctly identified fake samples, False Positives (FP) are fake samples misclassified as real, False Negatives (FN) are real samples misclassified as fake, and True Positives (TP) are correctly identified real samples. The specific values are: TN=363, FP=441, FN=368, TP=428.

The classification report, presented in Table I, provides precision, recall, and F1-score for each class, along with the overall accuracy, macro average, and weighted average.

Class	Precision	Recall	F1-Score	Support
0 (Fake)	0.50	0.45	0.47	804
1 (Real)	0.49	0.54	0.51	796
Accuracy			0.49	1600
Macro Avg	0.49	0.49	0.49	1600
Weighted Avg	0.49	0.49	0.49	1600

TABLE I
Classification Report for the Deepfake Detection Model.

In the table, precision measures the proportion of correct predictions among the predicted positives, recall measures the proportion of actual positives that were correctly identified, and the F1-score is the harmonic mean of precision and recall.

The overall accuracy of the model is calculated as follows:

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

$$= \frac{428 + 363}{428 + 363 + 441 + 368}$$

$$= \frac{791}{1600} \approx 0.4944$$

Therefore, the model's accuracy is approximately 49.44%.

During the Week 7 system testing, the model was evaluated, and the performance was observed to have certain limitations. The testing results indicated a tendency toward bias, predominantly classifying instances as "Real." This outcome is likely influenced by the model's current accuracy level of approximately 49 on the test set.

While the model demonstrates some capability in classification, there is room for further refinement to enhance its generalization and balance across different categories. Additionally, the system has not yet been developed into a real-time system.

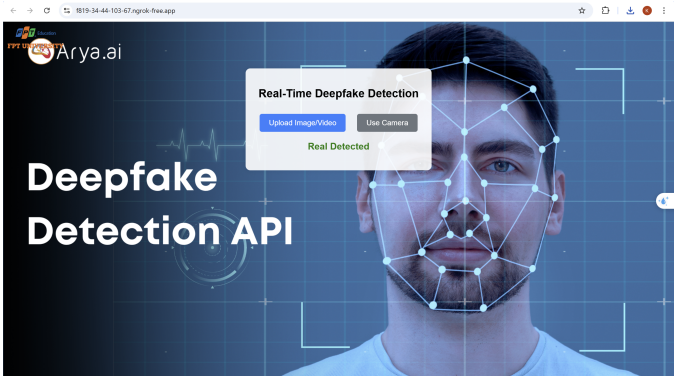

Fig. 11. test system built in week 7

REFERENCES

Please number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use "Ref. [3]" or "reference [3]" except at the beginning of a sentence: "Reference [3] was the first ..."

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use "et al.". Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [4]. Papers that have been accepted for publication should be cited as "in press" [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

REFERENCES

[1] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.
[2] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.
[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.
[4] K. Elissa, "Title of paper if known," unpublished.
[5] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.
[6] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
[7] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

IEEE conference templates contain guidance text for composing and formatting conference papers. Please ensure that all template text is removed from your conference paper prior to submission to the conference. Failure to remove the template text from your paper may result in your paper not being published.