BSc project

Deepfake Detection Using AI

Delivery date: 2. June 2025

Danmarks Tekniske Universitet

Antonijs Bolsakovs
Student Number: s225124
Supervisor: Gaurav Choudhary

Table of content

Abstract

This bachelor project 'Deepfake Detection Using Ai' is dedicated to the detection of deepfake images using artificial intelligence (AI). The use of this technique helps classify static images of persons as real or fake. At the moment, it is increasingly difficult to distinguish real images from fake, because deepfake technology develop with a huge speed that leads to consequences, such as disinformation and identification fraud. That is why this project is aimed at combating fake images and making a contribution to reliable detection methods that will help to solve the problems in society.

To work on this project, was created a balanced dataset from the popular Facefornsics++ dataset. The dataset was distributed on Train and Test, which contains 8000 real and 8000 fake images for training the model and 2000 real and 2000 fake images for testing the best saved model. After creating the necessary dataset, began the stage of development and training the Custom CNN as a baseline model, which achieved a validation accuracy of 81.19%. In addition to the custom CNN, two pretrained state-of-the-art architectures, xceptionNet and EfficientNet-B0, were fine-tuned and evaluated on the same dataset to compare their performance. Both models outperformed the custom cnn, with XceptionNet achieving the highest validation accuracy of 98.06%, while Efficientnet-B0 also performed strongly with a validation accuracy of 96.62%.

The project included detailed evaluation using metrics such as accuracy, precision, recall, and F1-score, as well as interpretability analysis through Grad CAM visualizations to better understand model behavior. As well as, the limitations which project faced - such as hardware constraints and limited training epochs, which were acknowledged, especially due to dependence on Google Colab for training.

The custom CNN achieved a strong result, reaching a test accuracy of 79%, proving the feasibility of detecting deepfakes even with a lightweight architecture. Comparing its performance to state of the art pretrained models such as xceptionNet and efficientNet-B0 provided valuable insights into areas for improvement. This comparison showed the benefits of deeper architectures and pretrained features, offering guidance for improving the custom model's design in future iterations. These results confirm that transfer learning is an effective strategy for improving detection performance. Future work could focus on deeper custom architectures, longer training (adding more epochs), and extending the detection approach to video-based deepfakes for even broader applicability.

1 Introduction

1.1 Background and Context

With the rapid development of Artificial Intelligence and Deep Learning, a problem has emerged in society, namely, we are faced with synthetically generated images (deepfake technology). Deepfake images and videos are related to manipulated digital content, especially those in which a person's identity and facial expressions, and speech are artificially changed or completely generated. These manipulations are often achieved using generative models such as generative adversarial networks (GANs) [1], variational autoencoders (VAEs), or transformer based architectures. We can see that the visual quality of deepfakes is rapidly increasing, making these images more difficult to detect by both human eyes and algorithms. Initially viewed as a new demonstration of machine learning capabilities, deepfakes have transitioned into a tool with serious societal implications very fast. But despite all the negative aspects, there exist a lot of positive ones, for example, deepfake is used in cinema as a digital face replacement, which greatly reduces the work of actors, it is also used in education, for example, language dubbing, and increasingly gaining popularity - personalized avatars for ordinary people and people with disabilities. However, the malicious uses of deepfakes have become a huge problem and have attracted a lot of attention. For example, they began to be used to spread false political narratives, as well as to create nonconsensual explicit materials involving celebrities or other individuals, and with the use of deepfake technologies there are a large number of cases when scammers pretend to be public figures or someone from a family for selfish purposes, and even to try to hack security systems with facial recognition. For example, in recent years, deepfake photos and videos have been distributed with various public figures who made controversial statements, which ultimately caused misinformation in the public and undermined trust in the content on the Internet.

There is a key reason for the urgency of research in this area, which is the increasing accessibility of deepfake generation tools. Open source frameworks and consumer-friendly applications have made it possible for almost anyone to create convincing fake media with minimal technical expertise. Deepfakes are no longer a niche research concern, they are a growing cybersecurity threat with real world consequences in legal systems, journalism, online banking, and digital identity verification.

Currently most of the academic research on deepfake detection focuses on video analysis. Typically, these approaches use temporal inconsistencies, motion artifacts, or audio visual inconsistencies between video frames. While these methods are good at detecting deepfakes inside the videos, they are not effective when only the single image is available. That's why, detecting deepfakes in images becomes critical in scenarios where individual frames are extracted from videos or when users upload images to social media or identity verification portals. This eliminates the benefit of temporal analysis as in videos, and requires that models be able to infer subtle visual artifacts only from static images.

However, we face various technical challenges when detecting deepfakes in images. For example, artifacts introduced during manipulation, such as unnatural blending, inconsistent lighting, or slight distortions, may be minimal and heavily masked by post-processing or compression. In addition, the detection performance itself may be degraded if deepfakes are generated by other unseen generation techniques. These challenges point to the need for robust deepfake detection models that can effectively function in a variety of real-world conditions.

This project aims to address this challenge by developing and evaluating deep learning models for accurate deepfake image classification. Focusing on single-frame detection contributes to a domain of high practical relevance, where image-only analysis can serve as a frontline defense mechanism in combating the spread of synthetic media.

1.2 Problem Statement

Society faces a big problem, as the growing realism of images and videos, together with the availability of deepfake technologies, creates a problem for digital media. Artificially generated images of faces created using neural networks can convincingly imitate real people. This makes traditional detection methods increasingly ineffective. This situation makes serious threats to the authenticity of information, digital

identity verification and, of course, public trust in visual content.
As mentioned, detecting deepfakes in images presents unique technical challenges. Unlike video based detection, where motion inconsistencies and temporal patterns may reveal manipulations, single frame detection relies solely on static visual features. Modern deepfakes often leave behind only subtle artifacts - such as minor inconsistencies in lighting, texture, blending, or facial geometry, that may be further obscured by image compression or postprocessing. These artifacts are frequently imperceptible and require specialized models capable of conducting detailed analysis.
Furthermore, datasets used for deepfake detection often exhibit bias toward specific manipulation techniques or lack diversity in terms of facial characteristics, lighting conditions, or compression levels. These limitations hinder the generalization ability of trained models and reduce their effectiveness in real-world applications.

In this project, the core research problem is defined as follows: *Can a deep learning model be trained to reliably classify static facial images as real or fake, using features learned only from image-level data, and how does its performance compare with state-of-the-art pretrained models?*
To address this problem, several steps were undertaken:

- **Dataset construction:** A custom dataset was created by extracting and labeling individual frames from the FaceForensics++ video dataset. As the result we got a balanced collection of 8,000 real and 8,000 fake images for training, and 2,000 real and 2,000 fake images for testing.
- **Model development:** A custom convolutional neural network CNN was designed and trained from scratch using PyTorch. However, its initial performance was limited, achieving approximately 50% accuracy, which motivated further experimentation.
- **Use of pretrained architectures:** To improve detection accuracy, two established deep learning models - XceptionNet and EfficientNet-B0, were fine-tuned and evaluated on the same dataset.
- **Evaluation and analysis:** Model performance was assessed using metrics such as accuracy, precision, recall, and F1-score. The results were visualized and compared to identify strengths and limitations of each approach.

By focusing on deepfake detection exclusively on static images and comparing both custom and existing pretrained models, this project aims to develop robust and accurate deepfake detection methods applicable in real-world.

1.3 Project Goals and Objectives

The main objective of this project is to develop, implement, and evaluate deep learning models that can accurately detect fake facial images. Considering all the factors such as the increasing difficulty of identifying fake images, high resource capacity and the growing need for reliable deepfake detection systems, this project aims to explore both custom-built and existing pre-trained models.
To achieve this goal, the following specific objectives were defined and achieved throughout the project:

1. **Construct a balanced dataset of real and fake images.** As discussed earlier, since the FaceForensics++ dataset is primarily video-based [2], individual frames were extracted and labeled to create a large-scale dataset tailored for image level classification. This dataset consisted of 8,000 real and 8,000 fake images for training, and 2000 real and 2,000 fake images for testing (see Section 3.1).
2. **Develop a custom CNN model for image based deepfake detection.** A lightweight convolutional neural network was designed and trained from scratch using the PyTorch framework. Model was intended to detect subtle visual artifacts in facial images and serve as a baseline for comparison with more complex architectures (see Section 3.3).

3. **Fine-tune and evaluate state-of-the-art pretrained models.** In order to benchmark the performance of the custom CNN, two well known architectures, such as XceptionNet and EfficientNet-B0, were fine-tuned on the same dataset. These models were selected for their proven effectiveness in image classification and previous success in deepfake detection tasks (see Section 3.4).

4. **Assess model performance using quantitative and visual metrics.** All models were evaluated using standard classification metrics, including accuracy, precision, recall, and F1-score. In addition, training and testing dynamics were visualized through plots of loss and accuracy curves, and key statistics were stored in structured formats such as CSV for future analysis (see Sections 4.1–4.3).

5. **Analyze experimental results to identify challenges and areas for improvement.** The project includes a comparative analysis of all models, highlighting the strengths and weaknesses of each approach (see Sections 4.4–6). The findings aim to guide future research and development of more generalizable and interpretable deepfake detection models.

Through these objectives, the project contributes to the ongoing efforts in developing robust and scalable solutions for preventing deepfake content, especially in contexts where only image-level evidence is available.

1.4 Report Structure

This report is organized into seven core sections and multiple subsections, each focusing on a critical stage of the project. Below is an overview of each section:

- **Section 1: Introduction** – Introduces the topic of deepfake detection, outlines the background and context, formulates the problem statement, states the project goals, and provides a structural roadmap for the report.

- **Section 2: Literature Review** – Discusses key concepts in deepfake generation and detection, surveys existing approaches, examines public datasets, and identifies research gaps that the project aims to address.

- **Section 3: Methodology and Results** – Describes the preparation of the dataset, implementation environment, the development of the custom CNN, and the fine-tuning of pretrained models (XceptionNet and EfficientNet-B0). It also describes how model outputs such as weights, evaluation metrics, and visualizations were saved and organized to ensure reproducibility and transparency.

- **Section 4: Comparative Analysis of Results** - Presents a comprehensive comparison of all models in terms of performance metrics, confusion matrices, training dynamics, and Grad-CAM interpretability visualizations. A discussion summarizes insights from these comparisons.

- **Section 5: Challenges and Limitations** – Describes hardware constraints, migration to Google Colab, model checkpointing practices, and limitations in training scope or dataset diversity.

- **Section 6: Conclusion and Future Work** – Summarizes the main outcomes and model performances, and proposes directions for future improvements such as deeper architectures, longer training, and video-based detection.

- **Section 7: Appendix** – Contains full Python implementations of training and evaluation scripts for the custom CNN, XceptionNet, and EfficientNet-B0, as well as saved best model checkpoints for reproducibility.

2 Literature Review

2.1 Introduction to deepfakes

Lets first understand what the term deepfake mean. The term *deepfake* is a combination of several words "deep learning" and "fake," and refers to synthetic media content that has been manipulated or entirely generated using artificial intelligence techniques, especially deep neural networks. The concept gained a lot of attention in 2017 on the popular platform Reddit, where a user named "deepfakes" began sharing face swapping videos generated using autoencoders. Since then, deepfake technology has evolved very fast, driven by advances in machine learning, as well as the availability of high quality training data, and increased computing power.

Deepfakes most often involve various facial manipulations, such as replacing one person face with another person in images or videos. However, over time, the range of possibilities has expanded to include voice replacement and cloning, lip syncing to a person's speech, facial expression changing, and even full body movement. The great success of deepfakes is their ability to reproduce high-dimensional distributions of facial features, thereby reproducing fine details such as skin texture, lighting, expressions, and also micromovements, making them increasingly difficult to distinguish from the real content.

In the context of this project, the focus lies specifically on static facial images, rather than video-based detection. This image-level scenario presents a distinct technical challenge: without access to temporal cues such as blinking or motion inconsistencies, the detection model must rely solely on spatial artifacts present in individual frames.

Several techniques are commonly used in the generation of deepfakes:

- **Autoencoders:** Among the earliest tools for face swapping, autoencoders consist of an encoder that compresses image into a latent representation and a decoder that reconstructs the original image. By training a shared encoder and two separate decoders (one per identity), early deepfakes could map faces between two individuals
- **Generative adversarial networks (GANs):** Introduced by Goodfellow et al. in 2014 [1], GANs consist of a generator that produces fake images and a discriminator that attempts to detect them. As these models compete during training, they improve iteratively. GAN-based models such as StyleGAN and StyleGAN2 have achieved impressive results in face synthesis, generating high-resolution images that are nearly indistinguishable from real faces.
- **Facial reenactment:** Methods like Face2Face and NeuralTextures allow manipulation of facial expressions by transferring movements from a source actor to a target identity [3]. These techniques operate in real time and preserve the original identity while modifying expressions.
- **Lip-syncing and audio-driven synthesis:** Tools such as Wav2Lip and Synthesia align facial movements with spoken audio [4]. These systems use both visual and auditory signals to generate synchronized mouth movements, creating the illusion that a subject said something they never actually did.
- **Transformer-based and diffusion models:** Although still emerging in the context of deepfake generation, transformer architectures and diffusion-based methods offer promising avenues for generating more controlled, photorealistic synthetic content. Their potential has yet to be fully realized in publicly available tools.

When creating deepfakes, the motive for their creation can vary greatly. For example, let's look at the positive use of deepfakes, such as use in the entertainment industry for such purposes as rejuvenating actors, dubbing films for other languages and for people with hearing impairments, as well as bringing historical figures to life in documentaries. Also for positive purposes deepfake technology is used in various applications that include creating your AI avatar for simplified communication with people and also for people with disabilities this is a great way to remove an inferiority complex.

On the other hand, the misuse of deepfake technology has raised significant ethical and legal concerns. Deepfakes have been used to spread political misinformation, create non consensual explicit content,

impersonate individuals in fraud schemes, and undermine trust in journalism and digital communications. The growing availability of open-source frameworks, for example, DeepFaceLab, Faceswap and mobile applications as Reface has significantly lowered the barrier to entry, allowing almost anyone to produce convincing manipulated media.
Several high-profile cases have underscored the seriousness of the threat. Fabricated videos of world leaders making controversial statements have gone viral, creating confusion and undermining trust in political discourse. Recently happend one very notable incident where cybercriminals used voicecloning technology to impersonate a company executive and authorize a fraudulent financial transaction. Such examples illustrate the need for reliable detection methods and proactive policy development to combat misuse.
In summary the deepfakes represent both an impressive technological achievement and a growing threat to information authenticity. Understanding how deepfakes are generated is a crucial step in developing reliable detection systems. This foundation directly motivates the present project, which focuses on building and evaluating AI-based models for detecting deepfakes in static facial images.

2.2 Deepfake generation techniques

The generation of deepfakes is based on a few important things - combination of advanced generative models and high-quality facial datasets. While the visual outcome of deepfakes may appear seamless to the human eye, they are the result of complex processes involving neural networks trained to model and manipulate facial data distributions. This section explores the most common deepfake generation techniques, with a focus on those represented in publicly available datasets such as FaceForensics++, which was used in this project.

Autoencoder-based methods

Autoencoders were among the first architectures used for face-swapping applications. These models learns to encode an input face into a compressed latent space and decode it back into an image. In deepfake generation, a shared encoder typically trained alongside two separate decoders, one for the source face and another one for the target. During inference, the encoder captures the facial features of the source and the decoder reconstructs them in the style of the target identity.
This approach is relatively simple and computationally lightweight, which in turn often results in visible artifacts around facial boundaries and has significant difficulties with complex expressions or lighting conditions. Nevertheless, autoencoder-based methods formed the basis for many early deepfake examples and are still represented in datasets like FaceForensics++ under the “DeepFakes” category [2].

Generative adversarial networks (GANs)

The most dominant method for creating photorealistic synthetic faces has become a Generative Adversarial Networks method. A GAN consists of two components: a generator that produces fake images and a discriminator that evaluates their authenticity. Through adversarial training, the generator gradually learns to produce more realistic outputs.
Several GAN based models are widely used in facial synthesis tasks:

- **FaceSwap:** A model that replaces facial regions based on detected landmarks and identity mapping. It uses GAN refinement to reduce boundary artifacts and improve realism.
- **NeuralTextures:** A method that learns a neural texture representation of the face and synthesizes new views using 3D geometry and texture projection.
- **StyleGAN/StyleGAN2:** They do not traditionally used for face swapping. These architectures can generate highly realistic synthetic faces from latent vectors, and are often used to create fully fabricated identities.

In the FaceForensics++ dataset, several video manipulation methods are based on GANs or similar rendering techniques. The presence of multiple manipulation pipelines, for example - FaceSwap, Face2Face,

NeuralTextures, creates a rich but heterogeneous distribution of visual features, which makes training detection models more challenging.

Facial reenactment and Expression transfer

There is a class that includes generation methods that change the facial expression or movement of a subject while preserving its identity. These techniques are often referred to as facial reenactment or expression transfer.
For example, Face2Face uses a realtime expression mapping system that captures the facial movements of a source actor and transfers them to a target face in a video. Unlike full face swapping, this method manipulates only dynamic features like mouth movements and eye expressions. These subtle modifications are particularly difficult to detect in still images, as they leave minimal spatial artifacts.
In deepfake detection, models must be sensitive to these low-level inconsistencies, especially when the manipulated content comes from reenactment methods that save most of the original facial structure.

Audio-driven manipulation techniques

While this structure is not the direct focus of this project, it is still worth mentioning that many modern deepfakes are generated using audiovisual pipelines. Techniques such as Wav2Lip or ATVGnet use an input audio signal to drive the motion of a target face, creating synchronized lip movements that match spoken content.
Even though the output appears consistent in video format, single-frame extractions from such manipulations can still contain trace inconsistencies around the mouth region or unnatural alignment, which can be detectable with the right model architecture.

Implications for Detection models

The variety of generation techniques has a direct impact on the complexity of deepfake detection. Each manipulation method introduces different types of artifacts, for example some affect skin texture, others impact boundary consistency or illumination cues. From a machine learning perspective, this heterogeneity increases the risk of overfiting to specific manipulation patterns if the model is not properly regularised or exposed to diverse examples.

In this project, all detection models were trained and tested on images extracted from the FaceForensics++ database, which contains samples from multiple generation techniques. Therefore, understanding how these deepfakes are created is not only important for context, but essential for designing detection models that can generalize across different types of manipulations.

2.3 Deepfake detection approaches

As deepfake generation techniques have advanced in realism and complexity, research on automated detection methods has become increasingly important. Detection approaches can be broadly categorized into traditional handcrafted methods, machine learning pipelines using feature engineering, and modern deep learning based solutions. This section outlines evolution of detection techniques, with a particular emphasis on convolutional neural networks (CNNs) and image-based analysis, which are central to this project.

Traditional Feature-based approaches

Early methods for detecting manipulated visual media relied on handcrafted features designed to capture inconsistencies in visual artifacts. These techniques often involved statistical analysis of pixel distributions, color histograms, edge mismatches, or lighting inconsistencies. Some examples include:

- Analysis of head pose and eye-blinking patterns.
- Detection of abnormal shadows or inconsistent illumination.

- Examination of JPEG compression artifacts or double quantization effects.

While such methods were computationally efficient and interpretable, they are lacked robustness. Even minor postprocessing could remove or obscure the artifacts they relied on. Moreover, handcrafted features often failed to generalize across manipulation techniques or datasets.

Shallow machine learning methods

A later evolution involved extracting relevant features manually, for example Local Binary Patterns, frequency domain coefficients, or histogram of oriented gradients. And feeding them into classifiers such as Support Vector Machines (SVMs), decision trees, or k-nearest neighbors. These approaches provided more control over feature selection but still required expert knowledge and often underperformed on complex or high-resolution manipulations.

Deep Learning-based approaches

The introduction of deep convolutional neural networks revolutionized the field of deepfake detection. Unlike traditional methods, CNNs can automatically learn hierarchical and spatial features from raw image data. This allows them to detect subtle manipulation traces that are difficult to handcraft or model explicitly.
Many architectures have been explored in the context of deepfake detection:

- **XceptionNet:** Originally proposed for image classification [5], this architecture employs depth-wise separable convolutions, allowing for efficient feature extraction. It was adapted for deepfake detection in the FaceForensics++ benchmark and remains one of the most widely used baselines.
- **MesoNet:** A compact CNN tailored for forgery detection [6], emphasizing low-level texture inconsistencies while maintaining low computational cost. It is suitable for scenarios with limited resources.
- **EfficientNet:** A scalable architecture [7] that balances depth, width, and resolution to optimize performance and efficiency. The B0 version offers a good compromise between accuracy and speed, making it ideal for large scale image analyzis.
- **Capsule Networks and Attention mechanisms:** Some researchers have explored even more complex architectures that focus on the spatial relationships between features, such as capsule networks or attention-based cnns. While promising, this models often require more training data and computational resources.

Video-based vs. Image-based detection

The main difference in this area is between video-level and image-level detection. Video-based methods often use time features, such as frame inconsistencies, unnatural blinking, head movements or audio-visual mismatches. Recurrent neural networks (RNNs), Long Short-Term Memory (LSTM) units, and 3D CNNs - are commonly used in such setups.
However, in many real-world scenarios, only a single frame or static image is available for analysis. For instance, in social media content moderation or ID verification systems. In such cases, image-level detection becomes very critical. These models must infer manipulation only from spatial patterns such as unnatural skin texture, or blending artifacts, or lighting inconsistencies, without relying on temporal clues.

Challenges in Deepfake detection

Despite the success of CNN-based approaches, several challenges remain:

- **Generalization:** Models often perform well on the dataset they were trained on. but they fail to detect manipulations created by unseen techniques or different rendering pipelines.

- **Compression sensitivity:** Real-world images are often compressed or resized during transmission, which can remove or distort subtle artifacts necessary for detection.
- **Bias and Overfitting:** Deep learning models may learn dataset specific patterns rather than universal manipulation features, which leading to overfitting.
- **Explainability:** CNNs are often seen as black boxes, which means it is hard to understand how they make decisions or check if they can resist manipulations like adversarial attacks.

Relevance to the present project

This project focuses exclusively on image-based detection where only static frames available. Because of the limitations of handcrafted methods and shallow models, a deep learning approach was adopted. The performance of a custom designed CNN was compared against two sota (state of the art) architectures, such as XceptionNet and efficientNet-b0, on images extracted from the FaceForensics++ dataset. This comparison aims to evaluate whether a lightweight, custom model can match or outperform complex pretrained networks in single-frame detection scenarios.

2.4 Public datasets for Deepfake detection

The development and evaluation of deepfake detection models require access to large, diverse, and well-anotated datasets containing both real and manipulated media. Over the past few years, several public datasets have been introduced to support research in this area. These datasets vary in terms of scale, manipulation techniques, actor diversity, and media formats, as (images vs. videos), all of which influence the efficiency and generalisability of detection models.

FaceForensics++

FaceForensics++ is one of the most widely used benchmark datasets [2] in deepfake research and was selected as the primary data source for this project. It consists of over 1,000 original videos which were collected from the internet and manipulated using four different face modification techniques: DeepFakes (autoencoder-based face swaps), Faceswap (landmark-based swapping), Face2Face (facial reenactment),and NeuralTextures (texture sinthesis using 3D geometry).
Each manipulated video is accompanied by its corresponding real version, and the dataset includes various compression levels to simulate real world media conditions (raw, light, strong). Although the dataset was originally designed for video analysis, but it is possible to extract individual frames to create an image-based classification dataset, which was done in this project.
The diversity of manipulation techniques in Faceforensics++ makes it a strong benchmark for testing the generalization of detection models. However its main limitation is that all videos are based on controlled conditions with frontal faces, and relatively clean backgrounds and also limited environmental variation.

DeepFake detection challenge (DFDC)

Organized by Facebook in collaboration with academic institutions, the DFDC dataset is one of the largest deepfake datasets publicly available. It contains over 100,000 video clips of actors speaking in various environments with the both real and fake content generated using multiple undisclosed manipulation techniques.
The DFDC dataset was specifically designed to promote research in robust detection under unconstrained conditions [8]. It includes a wide range of lighting scenarios, backgrounds, angles, and compresion artifacts. However lack of transparency about the exact manipulation methods used makes it more suitable for model validation than fine-grained technical analysis.
But while the DFDC dataset is powerful for video-based detection tasks, it large size and format complexity make it less practical for image studies without significant preprocessing.

Celeb-DF

Celeb-DF is a deepfake video dataset [9] that addresses the visual quality limitations present in earlier datasets. It includes high resolution videos of a celebrity interviews and corresponding manipulated versions generated using an improved face-swapping models.
One of the key strengths of Celeb-DF is the photorealistic quality of the manipulations, which closely resemble those found in online disinformation. The dataset has been used widely to benchmark cross dataset generalization and test model robustness on high-fidelity fakes.
However, very similar to DFDC, the Celeb-DF is distributed primarily in video format, and extracting good labeled frames for image-based classification tasks requires additional processing.

Other datasets

Other publicly available datasets include:

- **DeeperForensics-1.0:** Developed by ByteDance and it includes manipulated videos under various realworld conditions, such as different weather, motion blur and occlusion.
- **WildDeepfake:** Focused on uncontrolled settings with deepfakes found in the wild but not synthetically generated in a lab environment.
- **DFTIMIT and Google Deepfake datasets:** Smaller datasets which been used primarily for early experiments and baseline comparisons.

While these datasets offer valuable diversity, most of them are videooriented and were not directly applicable to this project which focuses on static image detection.

Dataset considerations in this project

For this project, was chosen the FaceForensics++ dataset due to its structured format, multiple manipulation types, and wide adoption in the deepfake research. Since the project focuses on image level detection - frames were extracted from each video and labeled accordingly, which is resulting in a balanced dataset of 8,000 real and 8,000 fake images for training, and also 2,000 real and 2,000 fake images for testing.
The controlled nature of Faceforensics++ ensured a consistency during model development. However this may also lead to limited generalization, as the dataset does not fully reflect the diversity of manipulations or also conditions found in realworld scenarios. In the future work may be beneficial to combine datasets or using data augmentation strategies to simulate more complex environments.

2.5 State-Of-The-Art detection models

Resent progress in deepfake detection has been largely driven by the adoption of advanced deep learning architectures, many of which were originally developed for general-purpose image classification. These models are often pretrained on large-scale dataset such as ImageNet, which have demonstrated exceptional performance in detecting manipulated facial content by using their ability to extract high level and fine-grained features.
This section outlines several state of the art (SOTA) models that have been widely adopted in deepfake detection researchs, with particular accent on those used in this project.

XceptionNet

XceptionNet that is short for “Extreme Inception,” is a convolutional neural network architecture that utilizes depthwise separable convolutions to reduce number of parameters and improve computational efficiency. Originally, it was proposed by Chollet in 2017 as an evolution of the Inception architecture, XceptionNet replaces traditional convolutions with a two step operation, which is a depthwise convolution following with a pointwise convolution.

In the context of deepfake detection XceptionNet was first adapted in the Faceforensics++ benchmark, where it was demonstrated superior performance over simpler CNNs and hand-crafted feature extractors. Its ability to capture subtle inconsistencies in texture, lighting, and facial geometry make it particularly effective in distinguishing real from manipulated faces.
In this project the XceptionNet was fine-tuned on an image based subset of the FaceForensics++ dataset. It serving as a high-performance baseline for comparison with other models.

EfficientNet

EfficientNet is in a family of scalable convolutional networks introduced by Tan and Le (2019). The core innovation behind EfficientNet is the compound scaling method, which well balances a network depth, width, and an input resolution to achieve better performance with fewer parameters.
EfficientNet-B0 is a smallest model in the family which offers a strong compromise between accuracy and efficiency. It has been used very successfully in various image analysis tasks, including a medical imaging, object recognition, and the deepfake detection.
In this project, the EfficientNet-B0 was chosen due to its lightweight nature and high performance on limited data. It can effectively learn patterns of increasing complexity from basic edges to detailed structures, while maintaining fast training speed made it suitable for experiments conducted on Google Colab.

MesoNet

MesoNet is a lightweight CNN architecture specifically designed for detecting facial forgerie. Unlike deeper networks, MesoNet focuses on capturing mesoscopic properties of images, such structures that are not visible at the pixel level but they are not enough abstract to require deep layers.
The models compactness allows for fast inference and lower resource consumption, making it a very good option for deployment on mobile or embedded devices. While not used in this specific project, MesoNet still is frequently referenced in literature as a competitive baseline, particularly for low-resolution or realtime detection tasks.

Face X-ray and ForensicTransfer

Beyond traditional classification networks, the new approaches such as Face X-ray and ForensicTransfer have introduced innovative paradigms in detection. Face x-ray treats deepfake detection as a pixel level anomaly segmentation task by identifying blending boundaries introduced during facial manipulation. ForensicTransfer on the other hand,focuses on few-shot generalization, enabling detection of unseen manipulation tipes with limited labeled data.
Although these models show promising results, and they often require more complex training setups and additional data modalities, for example segmentation masks which were outside the scope of this project.

Selection rationale and relevance to this project

In this project focus was placed on evaluating detection performance using models that are both accessible and efficient. XceptionNet and EfficientNet-B0 were selected for following reasons:

- **Proven effectiveness:** Both of this models have demonstrated state-of-the-art results in previous deepfake detection benchmarks, including FaceForensics++ and DFDC.
- **Architectural efficiency:** Their ability to capture subtle artifacts while maintaining reasonable computational requirements very well aligns with the resource constraints of the project environment -Google Colab.
- **Transferability:** As pretrained models they can be conveniently fine-tuned on task-specific data, making the model train faster and reducing the need for massive labeled datasets.
- **Comparative value:** These models serve as very strong baselines for evaluating performance of the custom designed CNN architecture developed in this project.

By integrating and finetuning these SOTA architectures on the image-based deepfake dataset created from FaceForensics++, this project contributes to understanding how well general-purpose classification models perform in specialised task of image-level deepfake detection.

2.6 Limitations in current research and Research gap

Despite the all considerable progress made in recent years, current research in deepfake detection still faces several limitations that affects the robustness, a generalizability, and practical deployment of the detection system. Understanding this challenges is very important for identifying areas where new contributions can have meaningful impact.

Generalization across manipulation techniques

One of the most frequently mentioned limitations in existing studies is lack of generalization to unseen manipulation methods. Many detection models are trained on datasets that contain a limited numbers of deepfake generation techniques. As a result, these models may achieve high accuracy on known manipulation types, but on the other hand perform poorly when exposed to new or more advanced fakes.
This problem is getting worse by the fact that the new deepfake generation methods continue to emerge and often with subtle artifacts that are different from those which present in training data. Without explicit strategies for domain adaptation or anomaly detection, many models fail to recognize fakes which are created with unfamiliar pipelines.

Overfitting to dataset-specific features

Another major issue lies in dataset dependency. Some detection models inadvertently learn to recognize compression patterns, artifacts introduced by video encoding, or inconsistencies in resolution and lighting features, that are specific to dataset rather than the manipulation itself. This can lead to overfitting and limits the realworld applicability of such models.
Cross-dataset evaluation often reveals significant performance drops, emphasizing the need for models that can learn features independent of the specific manipulation technique.

Lack of focus on Image-level detection

While video based detection methods have received considerable attention comparatively much less studies focus on image-only deepfake detection. Video methods are benefit from access to temporal information, this information such as unnatural blinking, motion discontinuities, or audio-visual desynchronisation. However in the many real world scenarios, only a one single frame is available. for example, in online image uploads, or profile verification systems, or content moderation workflows.
Image based detection is basically more challenging, as it requires the model to identify spatial artifacts from static image without temporal context. Moreover, there are less datasets which explicitly designed for image-level analysis, forcing researchers to extract and preprocess frames manually as done in this project.

Limited model Comparison under Consistent settings

Many academic works usually present results for a single detection model or compare across inconsistent experimental settings, for example different datasets, preprocessing pipelines, evaluation metrics. This makes it very difficult to fairly assess strengths and weaknesses of various architectures.
Moreover there are lack of detailed comparisons between custom built CNNs and powerful pretrained models in the context of image-only deepfake detection. Understanding this performance gap is essential for the selecting appropriate models for resource-constrained environments or deployment in real-world systems.

Research gap addressed in this project

This project addresses several of the limitations outlined above through the following contributions:

- **Focus on image-level detection:** The project centers on detecting deepfakes from individual facial images, without relying on video-level temporal signals.
- **Balanced and consistent dataset creation:** A well structured image dataset was built extracting and labeling frames from Faceforensics++ videos, ensuring equal representation of real and fake samples across training and testing.
- **Model benchmarking under controlled conditions:** The performance of custom CNN was evaluated and compared with SOTA models (XceptionNet and EfficientNet-B0), using same data, metrics, and evaluation pipeline.
- **Selecting a model based on resources:** The project takes into account practical limitations such training time, computational cost, and deployment feasibility, which often overlooked in academic studies.

By focusing on these aspects, the project contributes to filling a key gap in deepfake detection research, namely the lack of systematic, image-level evaluations of different model architectures in consistent experimental settings. In addition there are many existing studies which compare detection models using inconsistent data sets, preprocessing techniques, or evaluation metrics, which complicates fair performance assessment. This project addresses this issue by ensuring that all models trained and evaluated under identical experimental conditions enabling a controlled and reproducible comparison. [1]

3 Methodology and Results

3.1 Dataset preparation

A critical step in developing an effective deepfake detection system is creation of a representative and balanced dataset. In this project, the FaceForensics++ dataset [2] was selected as the main data source. Faceforensics++ is one of the most widely used datasets in deepfake detection research, consisting primarily of manipulated and authentic videos. Since the focus of project is on static image-based deepfake detection rather than video-based analysis, additional preprocessing steps were necessary to adapt the dataset to the project's needs.

Frame extraction

The original FaceForensics++ dataset is video oriented, which means that it contains temporal sequences of frames with motion information. To adapt it for image-level classification, frames were extracted from each video using a custom Python script developed for this project (see Appendix 8).
Images were extracted every 10 frames throughout each video to ensure a diverse set of facial expressions, angles, and lighting conditions while minimizing redundancy. Each frame was resized to a consistent resolution of 128×128 pixel and saved in JPEG format to balance quality and storage requirements. The simplified version of the frame extraction code is shown below:

```python
def extract_frames(video_path, output_folder, num_frames=10):
    cap = cv2.VideoCapture(video_path)
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    frame_interval = max(1, total_frames // num_frames)  # select 10 evenly spaced frames

    for i in range(num_frames):
        cap.set(cv2.CAP_PROP_POS_FRAMES, i * frame_interval)  # jump to the desired frame
        ret, frame = cap.read()
        if not ret:
            break
        img_filename = os.path.join(output_folder, f"{os.path.basename(video_path).split('.')[0]}_{i}....
    jpg")
        cv2.imwrite(img_filename, frame)  # save the frame

    cap.release()
```

Listing 1: Simplified frame extraction script

Dataset balancing and Splitting

Now following the frame extraction, the images were divided into training and testing sets while maintaining class balance. This was accomplished using a second custom Python script (see Appendix 7.1). the splitting process ensured randomized selection while saving equal numbers of real and fake images. Maintaining a balanced distribution of a real and fake images in both training and testing sets was crucial to prevent bias during model learning.
The snippet below shows the actual code block used for moving real and fake images into their corresponding train/test directories after shuffling. Full implementation is available in Appendix 9.

```python
        for img in train_images:
            shutil.move(os.path.join(data_dir, category, img), os.path.join(train_dir, category, img))

        for img in test_images:
            shutil.move(os.path.join(data_dir, category, img), os.path.join(test_dir, category, img))
```

Listing 2: Simplified data splitting script

The final dataset structure was:

- **Training set:** 8,000 real images + 8,000 fake images
- **testing set:** 2,000 real image + 2,000 fake images

Data was manually verified after splitting to ensure that there is no leak between training and testing sets.

Preprocessing and augmentation

Before training the model, were applied the following preprocessing steps:

- **Normalization:** pixel values were automatically scaled to the [0, 1] range using PyTorch's ToTensor transformation.
- **Shuffling:** Training images were shuffled before each epoch to ensure that the model did not learn any ordering artifacts.

Heavy data augmentation techniques were avoided to keep the natural appearance of facial images. Future extensions could consider simple augmentation strategies to improve generalization.

Storage and Organization

All processed images were organized into created structured folders by label ('real' and 'fake') and dataset split into ('train' and 'test'). The complete dataset was uploaded to Google drive, allowing fast and stable access during training sessions in the Google Colab.
This careful dataset preparation ensured fair comparisons and reproducibility across all models evaluated in this project.

3.2 Training environment and tools

Initial local setup and Challenges

The initial stages of this project were carried out on a local machine:a 2021 MacBook Pro equipped with an Apple M1 Pro chip and 16 GB of RAM. Development was performed using the PyCharm IDE, and the initial tasks completed locally included:

- Extraction of frames from videos using custom Python scripts.
- Organization of the extracted images into appropriately labeled directories ('real' and 'fake').
- Initial creation and prototyping of a custom convolutional neural network (CNN) architecture.

Due to the hardware limitations of the local machine specifically, the absence of a dedicated GPU and reliance solely on the CPU training deep learning models was exceedingly slow. Each epoch required a substantial amount of time to complete, which made iterative model development and fine-tuning impractical. Additionally, extended CPU usage led to significant heating of the local machine, further complicating long training sessions.
Recognizing the need for faster experimentation cycles and more intensive model optimization, the decision was made to migrate the training environment to Google Colab.

Migration to Google Colab

Google Colab was selected as the cloud-based environment for model training due to its provision of free GPU acceleration and easy integration with Google Drive for data management. In Colab, the runtime was configured to Python 3 with a T4 GPU hardware accelerator, significantly improving training speeds compared to the local CPU setup.
The prepared dataset, originally organized on the local machine, was compressed into a ZIP archive and uploaded to Google Drive. Upon launching the Colab environment, the dataset was unzipped using:

```
!unzip "/content/drive/MyDrive/deepfake_detection/faceforensics_data.zip" -d /content/
```

The folder structure was then adjusted with:

```
!mv "/content/untitled folder" /content/face_data
```

To ensure that the dataset was correctly recognized by the Colab environment, the following commands were executed:

```
!ls /content/face_data/train
!ls /content/face_data/test
```

These commands confirmed the presence of 'real' and 'fake' subfolders under both 'train' and 'test' directories.
Additionally, the following commands verified the number of images in each category:

```
# TRAIN
!find /content/face_data/train/real -name "*.jpg" | wc -l
!find /content/face_data/train/fake -name "*.jpg" | wc -l

# TEST
!find /content/face_data/test/real -name "*.jpg" | wc -l
!find /content/face_data/test/fake -name "*.jpg" | wc -l
```

The outputs confirmed the intended dataset distribution:

- 8,000 real and 8,000 fake images for training
- 2,000 real and 2,000 fake images for testing

Furthermore, the `timm` library was installed via:

```
!pip install timm
```

This library was later used for model fine-tuning and architecture management.
The transition to Google Colab allowed for much faster training times, facilitated repeated experimentation and fine-tuning cycles, and alleviated thermal constraints that had been present when using the local machine. This move was crucial for the successful development and optimization of deepfake detection models in this project.

Frameworks and libraries used

Several key frameworks and libraries were utilized throughout the project to facilitate data handling, model development, training, evaluation, and visualization:

- **PyTorch:** The primary deep learning framework used to build, train, and evaluate both the custom CNN and pretrained models (XceptionNet, EfficientNet-B0). It was preinstalled in the Google Colab environment.
- **torchvision:** Used for standard image transformations, loading pretrained models, and working with datasets. It was also preinstalled in Colab.
- **timm:** Installed manually via `!pip install timm`, this library provided access to a wide range of pretrained models and streamlined model loading and fine-tuning.
- **OpenCV (cv2):** Used for video frame extraction and basic image processing operations; preinstalled in Colab.
- **matplotlib:** Used to visualize training progress (loss and accuracy curves) and evaluation results; preinstalled in Colab.

- **scikit-learn:** Used for calculating classification metrics such as precision, recall, F1-score, and for generating confusion matrices; preinstalled in Colab.
- **GradCAM (Custom implementation):** Used to generate class activation maps (CAMs) that provide interpretability by highlighting regions of interest for model predictions. Implemented manually using PyTorch functionalities without installing a separate Grad-CAM package.

These frameworks and libraries were critical in building a flexible and efficient workflow for deepfake detection model development.

Code and data management

All code related to model architecture, training, evaluation, and visualization was written, executed, and tested directly within Google Colab notebooks. This approach provided an integrated environment that simplified code management and experiment tracking.
The dataset, initially prepared on the local machine, was uploaded to Google Drive under a dedicated project folder named `deepfake_detection`. Inside the folder, data was organized into subdirectories:

- `face_data/train/real`
- `face_data/train/fake`
- `face_data/test/real`
- `face_data/test/fake`

Model checkpoints (.pth files), training metrics (.csv files), Grad-CAM visualizations, and experimental results were also stored within the project folder on Google Drive. This consistent structure made it easy to resume training, test different models, and retrieve results at any stage of the project.

Reproducibility considerations

Ensuring reproducibility was a key consideration throughout the project. By maintaining a clear folder structure in Google Drive and saving all critical artifacts such as model weights, metric logs, and visualizations, the experiments can be easily revisited and re-evaluated.
To minimize variability between training runs, random seeds were fixed across different libraries (Python's random module, NumPy, and PyTorch), ensuring that operations such as weight initialization, data shuffling, and dataset splitting remained consistent. Fixing random seeds in training scripts helped guarantee that models produced comparable results across multiple training sessions, thereby enhancing the credibility of experimental findings.
Despite the advantages of using Google Colab, some challenges were encountered due to GPU time limits. Occasionally, when GPU usage quotas were exceeded, model development and small-scale testing had to continue on CPU mode. This CPU fallback allowed development to proceed without interruption, while final model training and evaluations were always conducted using GPU to ensure optimal performance and consistency.
The combination of careful code organization, versioning, standardized runtime settings, and fixed random seeds helped ensure that the models and experiments could be reliably reproduced if needed.

3.3 Custom CNN model

Architecture description

The custom convolutional neural network (CNN) developed for this project was specifically designed to classify facial images as either real or fake. The architecture was developed from scratch and intentionally kept lightweight to balance computational efficiency and learning capability, considering the input resolution of 299×299 pixels.

The model incorporates convolutional bottleneck blocks enhanced with attention mechanisms (CBAM), allowing the network to focus on the most informative facial regions. Each block uses both channel and spatial attention and is followed by max pooling. The network accepts a 4-channel input of size $4 \times 299 \times 299$, where the fourth channel represents the Laplacian edge map of the original image. Final classification is performed via fully connected layers after global pooling.
The core structure of the model is presented below (see full implementation in Appendix 10):

```python
class DeepFakeCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.stem = nn.Sequential(
            nn.Conv2d(4, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.SiLU(),
            nn.MaxPool2d(2)
        )
        self.blocks = nn.Sequential(
            BottleneckCBAM(32, 32, 64), nn.MaxPool2d(2),
            BottleneckCBAM(64, 64, 128), nn.MaxPool2d(2),
            BottleneckCBAM(128, 128, 256), nn.MaxPool2d(2),
            BottleneckCBAM(256, 256, 512)
        )
        self.pool = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.AdaptiveMaxPool2d(1)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Dropout(0.5),
            nn.Linear(1024, 2)
        )

    def forward(self, x):
        x = self.stem(x)
        x = self.blocks(x)
        avg = self.pool[0](x)
        max = self.pool[1](x)
        x = torch.cat([avg, max], dim=1)
        return self.classifier(x)
```

Listing 3: Core architecture of the custom CNN

The architecture begins with a stem convolution layer, followed by four bottleneck blocks that progressively increase the number of feature maps while reducing spatial resolution. Each convolutional layer uses a 3×3 kernel with stride 1 and padding 1, and is followed by a max-pooling layer with a 2×2 window to halve the spatial size.
After the convolutional feature extraction, the output is flattened and passed through two fully connected layers. A dropout layer with a probability of 0.5 is applied to prevent overfitting. The final layer outputs logits corresponding to the two classes: real and fake.
The total number of trainable parameters is approximately 1.9 million, making the model lightweight and efficient, while maintaining sufficient expressive power for the task.

Design rationale

The design of the custom convolutional neural network (CNN) was guided by a balance between model simplicity, computational efficiency, and learning capacity, tailored specifically for static deepfake image classification.
Given that the input images were resized to 299×299 pixels, the model architecture was optimized to progressively extract hierarchical features while controlling the number of parameters to avoid overfitting and to maintain manageable training times, especially when initially operating under CPU constraints.

Several design choices were inspired by successful practices observed in state-of-the-art (SOTA) models:

- **Use of multiple convolutional blocks:** Stacking three convolutional layers allowed the model to capture low-level (edges, textures), midlevel (parts of faces), and highlevel (overall facial structures) features.
- **ReLU activation:** Rectified Linear Units (ReLU) were employed after each convolutional layer to introduce non-linearity and to facilitate stable and efficient gradient propagation.
- **MaxPooling layers:** Pooling operations with 2×2 kernels were inserted after each convolution to progressively reduce spatial dimensions, making the model invariant to small translations, reducing computational load.
- **Dropout regularization:** A dropout layer with a probability of 0.5 was included in the fully connected block to mitigate overfitting, particularly important due to relatively small sise of the dataset compared to large scale datasets like ImageNet.

Initially, the model showed poor performance like achieving validation accuracy of approximately 55%. To improve the model, was decided to use techniques inspired by SOTA architectures. Such as:

- Increasing the number of convolutional filters across layers ($32 \rightarrow 64 \rightarrow 128$) to enable the model to learn more complex and detailed representations.
- Introducing dropout after the first fully connected layer to improve generalization.
- Fine-tuning learning rate schedules and optimizing hyperparameters through multiple training iterations.

These improvements led to a significant boost in model performance. After applying the revised architecture and training for 10 epochs, the model achieved a validation accuracy of 81.19% and a final test accuracy of 79.3%.
Training progress is visualized in the graphs below.

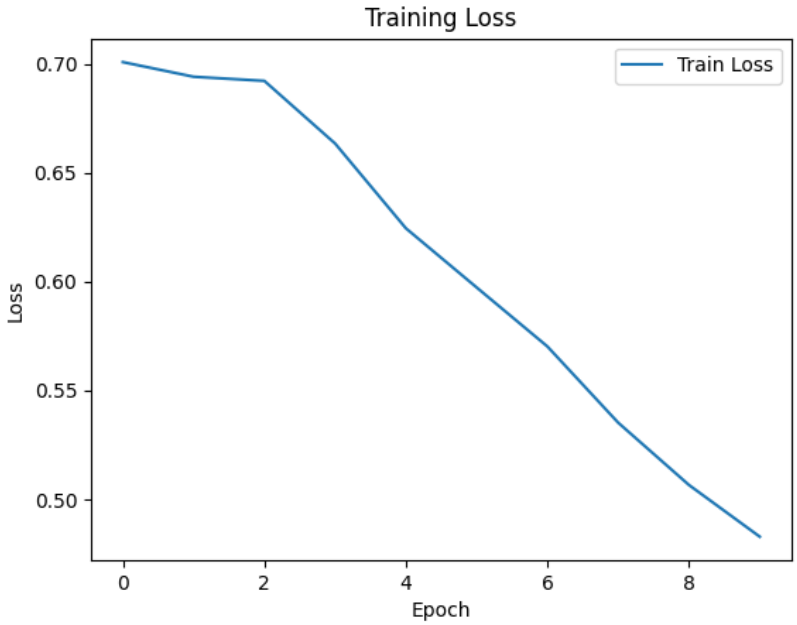

Figure 1: Training loss over 10 epochs for Custom CNN

Figure 1 illustrates the training loss progression over the course of 10 epochs. The graph shows a consistent and smooth decrease in loss, indicating that the model was successfully minimizing the classification error on the training set without signs of instability or divergence.

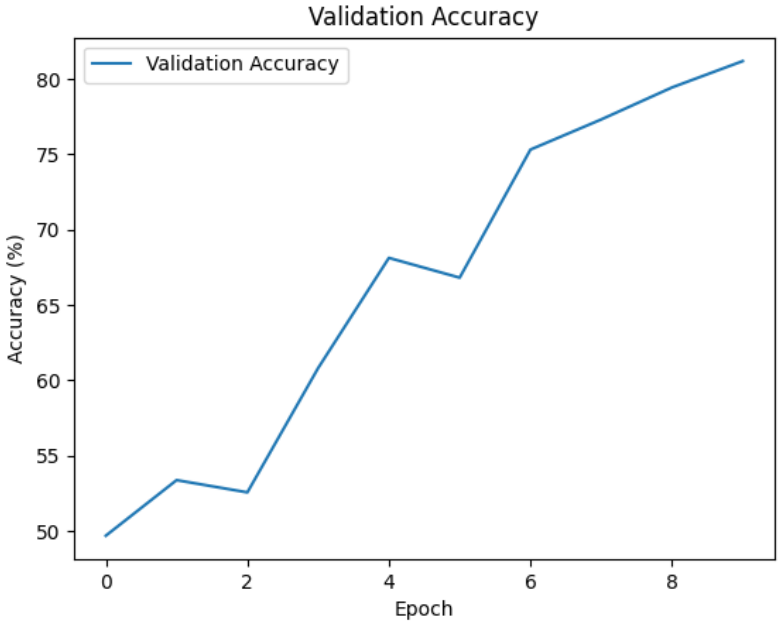

Figure 2: Validation accuracy over 10 epochs for Custom CNN

Figure 2 presents the model's validation accuracy across 10 epochs. A steady upward trend is observed, suggesting that the model was able to generalize better to unseen data as training progressed. The lack of abrupt drops or oscillations in the validation accuracy curve indicates that overfitting was effectively mitigated.
The overall training behavior suggests that the architectural improvements and hyperparameter optimizations contributed to stable convergence and enhanced the model's capacity to distinguish between real and fake images.

Model Improvement Process

During the initial phases of model development, the custom CNN demonstrated poor performance, achieving a validation accuracy of approximately 55%. Early experiments revealed that the model was not able to effectively extract the subtle artifacts and inconsistencies typical of deepfake images.
To improve performance, inspiration was drawn from best practices commonly used in state-of-the-art architectures. Several key modifications were introduced into the initial network design:

- **Increasing the number of filters:** The depth of the convolutional layers was expanded from 32 to 64 and then to 128 filters. This allowed the network to learn increasingly complex feature hierarchies.

The following snippet illustrates an intermediate version of the convolutional block. While it does not represent the final implementation (see Appendix 10), it highlights the key design adjustments that contributed to performance gains:

```
self.conv_layers = nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2, 2),

    nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2, 2),
```

```
    nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),
    nn.MaxPool2d(2, 2)
)
```

- **Incorporating dropout:** A dropout layer with a probability of 0.5 was added before the final fully connected layers to reduce overfitting and promote better generalization.

The dropout implementation is shown below:

```
self.fc_layers = nn.Sequential(
    nn.Linear(128 * 37 * 37, 512),
    nn.ReLU(),
    nn.Dropout(0.5),
    nn.Linear(512, 2)
)
```

- **Optimization tuning:** The optimizer was changed to Adam, which is known for better handling of sparse gradients and faster convergence compared to traditional SGD. A learning rate of 0.001 was selected after several iterations to ensure stable training without overshooting.

The optimizer setup was configured as follows:

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Standard ReLU activations and max-pooling operations were preserved after each convolutional layer to maintain non-linearity and downsample feature maps while retaining relevant information.
These improvements were applied and evaluated through multiple rounds of training. The integration of increased filter depth, dropout regularization, optimized activation and pooling strategies, and adaptive optimization collectively enabled the model to substantially improve its classification performance.
Through this iterative refinement process, the model evolved from a simple prototype to a more expressive and reliable deepfake detector. The consistent use of SOTA techniques significantly contributed to improved training stability, faster convergence, and stronger generalization capacity in the final custom CNN.

Training process

The training of the custom CNN model was conducted over 10 epochs, balancing sufficient iterations for learning while preventing overfitting. The Adam optimizer was used with a learning rate of 0.001, providing adaptive moment estimation and efficient convergence compared to traditional stochastic gradient descent (SGD).
The CrossEntropyLoss function was selected as the loss criterion, appropriate for multi-class classification tasks and ensuring stable optimization during the training process.
The dataset was divided into two subsets:

- **Training set:** 8,000 real and 8,000 fake images
- **Validation set:** 2,000 real and 2,000 fake images

This division enabled the model to learn from the training data while being regularly evaluated on unseen validation data, allowing real-time monitoring of generalization performance through validation accuracy.
The training parameters are summarized as follows:

- **optimizer:** Adam
- **Learning rate:** 0.001

- **Loss function:** CrossEntropyLoss
- **Number of epochs:** 10
- **Batch size:** 32
- **Training platform:** Google Colab (T4 GPU)

During each epoch, the model's performance was monitored through two primary indicators: training loss and validation accuracy. These metrics were plotted to assess convergence trends and detect any signs of overfitting.
The progression of training loss and validation accuracy is visualized in Figure 1 and Figure 2 (presented previously in Design Rationale). A consistent downward trend in training loss and a steady rise in validation accuracy were observed, indicating that the model effectively learned to distinguish between real and fake images over the course of training.
Additionally, a snapshot of the training log captured from Google Colab is provided in figure 3:

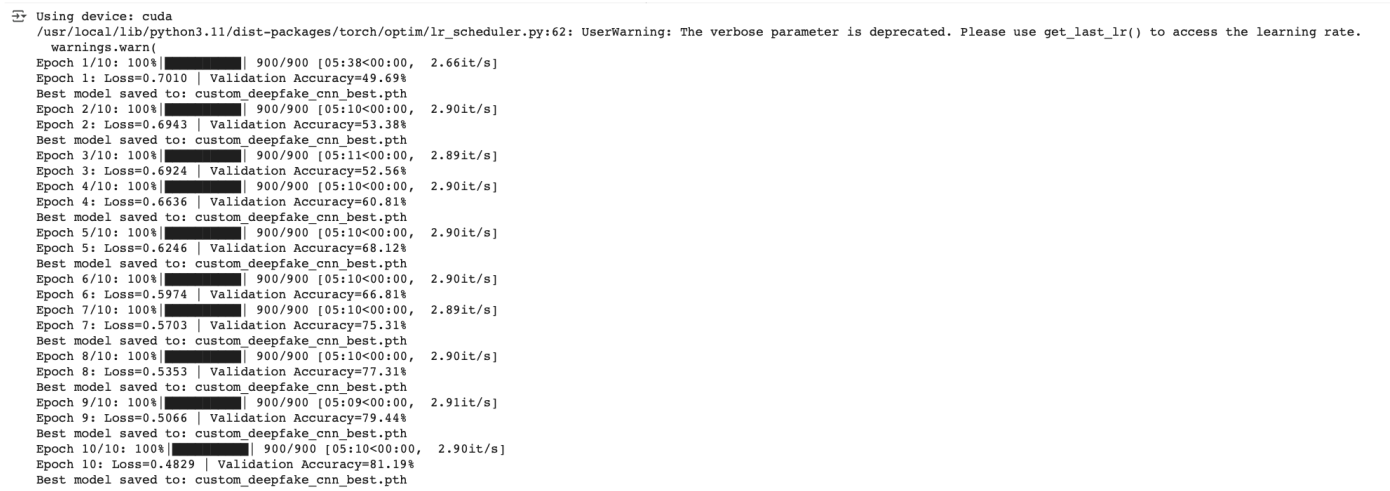

Figure 3: Training log screenshot from Google Colab

Figure 3 shows the training output for all 10 epochs. For each epoch, the loss value and validation accuracy are reported. It is evident from the log that the model consistently improved validation performance, ultimately achieving a best validation accuracy of 81.19% at the final epoch.
Upon detecting an improvement in validation accuracy during training, the model's weights were saved using a checkpointing mechanism. The best-performing model, based on validation accuracy, was stored as `custom_deepfake_cnn_best.pth`. This checkpointed model was later used for final testing and evaluation on the completely unseen test set.
The consistent convergence behavior, increasing validation performance, and checkpointing of the best model ensured that the training process was both stable and optimized for generalization.

Testing and evaluation

After completing the training phase, the custom CNN model was evaluated on a completely unseen test set to assess its generalization ability. The best-performing model from training, stored as `custom_deepfake_cnn_best.pth` (which you can download by the active link in the appendix 7.5), was loaded and used to perform inference and produce prediction outputs.
The test set consisted of 4,000 facial images: 2,000 real and 2,000 fake. These images were entirely separate from the training and validation datasets and were used exclusively to measure the model's true predictive performance.
Testing was performed using a dedicated Python script named `evaluate_custom_cnn.py`, which handled model loading, test data processing, and evaluation metric computation. The following code snippet

is taken directly from the official evaluation script used in this project (see full implementation in Appendix 11). It demonstrates how the trained model is loaded and used to perform inference on the test set:

```
model = DeepFakeCNN().to(device)
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
model.eval()

y_true, y_pred = [], []
correct_samples, wrong_samples = [], []

for inputs, labels in tqdm(test_loader):
    inputs, labels = inputs.to(device), labels.to(device)
    outputs = model(inputs)
    preds = torch.argmax(outputs, 1)
    y_true.extend(labels.cpu().numpy())
    y_pred.extend(preds.cpu().numpy())
```

Listing 4: Evaluation inference loop for the custom CNN

Although not shown in the snippet above, the evaluation script utilized PyTorch's inference mode using `torch.no_grad()` to disable gradient computations and reduce memory consumption during evaluation. Model outputs were then compared against ground truth labels to compute standard classification metrics, including precision, recall, F1-score, and overall accuracy.
Additionally, a confusion matrix was generated to visualize the distribution of predictions across real and fake classes. The results of this evaluation are further presented in the subsequent sections: *Performance Metrics* and *Grad-CAM Visualizations*.
These results confirmed that the model achieved strong generalization performance on unseen data. They also helped identify failure cases and minor class imbalances, which are discussed in more detail later in the analysis.

Performance Metrics

To evaluate the predictive performance of the custom CNN model, standard classification metrics were computed on the test set consisting of 4,000 images (2,000 real and 2,000 fake). These metrics include accuracy, precision, recall, and F1-score for each class.
The evaluation was performed using the `evaluate_custom_cnn.py` script, which generated a classification report and a confusion matrix. A summary of the test performance is presented in Table 1.

Table 1: Test set Performance metrics for Custom CNN

Class	**Precision**	**Recall**	**F1-score**
Fake	0.88	0.68	0.77
Real	0.74	0.91	0.81
Overall accuracy	0.79		

The confusion matrix in Figure 4 provides a more detailed view of the model's classification behavior.

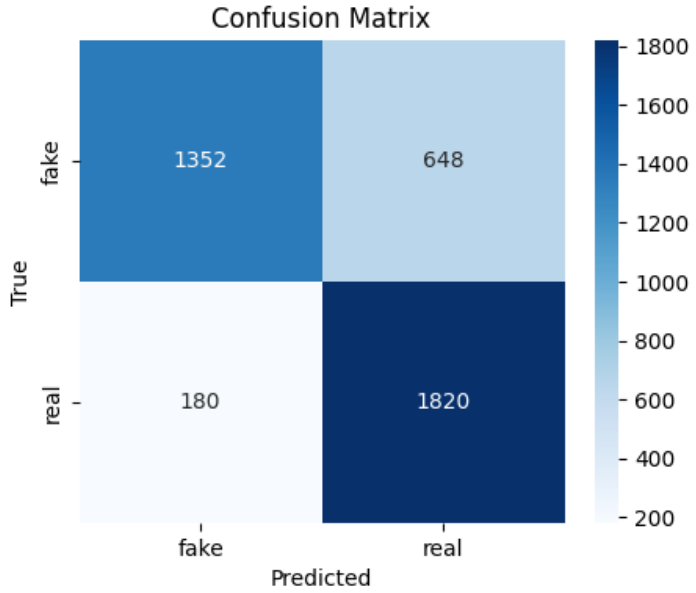

Figure 4: Confusion matrix for Custom CNN on test set

As shown in Figure 4, the model correctly classified 1,820 real images and 1,352 fake images. However, it also misclassified 648 fake images as real, indicating that while the model is strong at detecting authentic faces, it has more difficulty recognizing deepfake content - which is often subtle and visually convincing. This asymmetry is reflected in the class-specific recall values: 91% for the real class versus only 68% for the fake class. This gap suggests that the model is more sensitive to authentic features than to synthetic irregularities, which may require further tuning or deeper architectures to capture more subtle manipulation artifacts. The model's better performance on real images can likely be attributed to the consistency of facial structure and lighting in authentic photographs. Fake images, in contrast, often contain subtle, high-frequency artifacts that are harder to distinguish from natural variations without deeper or more specialized architectures.

This limitation implies that while the current model learns general facial features effectively, it may benefit from additional enhancements specifically targeting the detection of manipulation artifacts. Future work may explore techniques such as fine-grained texture analysis, higher input resolution, or ensemble architectures to address this imbalance.

A screenshot of the full evaluation output, including the metrics generated after testing, is shown in Figure 5.

```
Using device: cuda
  0%|          | 0/250 [00:00<?, ?it/s]/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py:183
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
100%|██████████| 250/250 [01:15<00:00,  3.32it/s]

Classification Report:

              precision    recall  f1-score   support

        fake       0.88      0.68      0.77      2000
        real       0.74      0.91      0.81      2000

    accuracy                           0.79      4000
   macro avg       0.81      0.79      0.79      4000
weighted avg       0.81      0.79      0.79      4000

✅ Evaluation complete. All results saved to: /content/drive/MyDrive/deepfake_detection
```

Figure 5: Model evaluation output from Google Colab

The values shown in the screenshot confirm the accuracy of 79% reported above, and serve as a transparent reference for the result reproducibility. These metrics offer a clear baseline for comparison with more advanced architectures described later in the report.

Grad-CAM visualizations

To better understand the internal reasoning of the custom cnn model, Grad-CAM (Gradient-weighted Class Activation Mapping) was applied to selected test images. Grad-CAM highlights the regions in the input image that contributed most to the model's prediction, allowing visual interpretation of model attention.
In Grad-CAM visualizations, warmer colors (such as red and yellow) indicate regions that contribute strongly to the model's decision, while cooler colors (such as blue) represent areas of low importance. These heatmaps allow us to see where the model is "looking" when making predictions.
For deepfake detection, where manipulations are often subtle and visually realistic, model interpretability is especially important. Without visualization, it is difficult to know whether the model is making predictions based on meaningful facial regions or relying on irrelevant cues such as background textures, overlaid text, or video compression artifacts.
Grad-CAM visualizations address this issue by allowing researchers to verify that the model is focusing on semantically important features, such as eyes, mouth, and facial boundaries — which are typically distorted in manipulated media.
Figures 6 and 7 show examples of correctly classified fake images. In both cases, the model's attention is concentrated around the facial regions, particularly the eyes, mouth, and skin boundaries which are often distorted or less naturally blended in deepfake images.

Figure 6: Correctly classified fake images: strong focus on facial boundaries and eyes

In Figure 6, the model clearly identifies the manipulated areas around the eyes and forehead, likely due to unnatural shading or geometry mismatches. This focused attention demonstrates that the CNN has learned associate visual inconsistencies with the fake class.

Figure 7: Correct fake prediction with consistent focus on face and head shape

In contrast, figures 8 and 9 illustrate misclassified fake images. In these cases, models attention is scattered or misplaced, focusing on nonrelevant background areas, text overlays, or peripheral regions instead of the face.

Figure 8: Misclassified fake images: attention focused on irrelevant background areas

Figure 8 demonstrates that when the model fails to attend the facial features either due to occlusion, distracting text, or visual clutter its ability to identify manipulation decreases.

Figure 9: Incorrect Prediction: Model attends mostly to clothing and screen layout

In Figure 9, the model attends to clothing and layout elements rather than the face. This type of misdirection suggests that the model is not always robust in isolating semantically important facial zones, which is critical for deepfake detection.
This grad-cam results reinforce earlier observations: while the model is able to identify typical manipulations when its attention is correctly localized, it struggles in the presence of distractions, occlusions, or subtle fakes. Visualization techniques such as grad-CAM offer valuable insight into the model's limitations and suggest directions for improving attention mechanisms or input pre-processing in future work.

Limitations and discussion

Although the custom CNN model demonstrated promising results, achieving test accuracy 79% and reasonable class-wise precision and recall, several limitations were observed throughout the project.

- **Limited model depth:** The architecture consists of only three convolutional layers followed by two fully connected layers. This relatively shallow design limits the network's ability to capture complex, high-level features often required to detect subtle manipulations in deepfake images.
- **Recall imbalance:** As seen in the evaluation metrics, the model achieved 91% recall for real images but only 68% for fake images. This suggests the model is more confident in recognizing authentic content but struggles to consistently detect synthetic artifacts, especially when they are visually subtle or blended.
- **Sensitivity to background and layout:** Grad-can visualizations revealed that, in misclassified cases, the model sometimes attended to irrelevant regions such as clothing, backgrounds, or overlays rather than facial features. This indicates that the model's spatial attention mechanisms could be further improved.
- **No temporal context:** As this work focuses on image-level deepfake detection, the model does not utilize temporal cues present in videos, such as frame-to-frame inconsistencies, blinking patterns, or unnatural motion, which can provide strong signals for detection.

To address some of these limitations, several techniques inspired by state-of-the-art (SOTA) architectures were integrated into the model, including increased convolutional depth, dropout regularization, and adaptive optimization using the Adam optimizer. These adjustments contributed to a significant improvement in model performance, increasing validation accuracy from 55% to 81.19%.
Nonetheless, these improvements were bounded by the simplicity of the architecture and the absence of pretraining. Future iterations could explore the following enhancements:

- Increasing model depth or using residual connections to improve feature extraction capacity.
- Incorporating spatial attention modules or face landmark alignment to ensure the model focuses on semantically important regions.
- Using input images with higher resolution or applying pre-processing techniques to enhance texture-level artifacts.
- Combining frame-based analysis with temporal models (e.g., 3D CNNs or recurrent architectures) to extend detection to video inputs.

Despite this limitations, the custom cnn provided a functional and interpretable foundation for deepfake image classification. The results serve as a useful benchmark and baseline for comparing more advanced, pretrained architectures explored later in the report. These comparisons were especially important given the model's limitations in depth, attention, and generalization, and helped validate the benefits of transfer learning from larger-scale datasets.

3.4 Fine-tuning Pretrained models

3.4.1 XceptionNet model

why XceptionNet?

XceptionNet, introduced by Chollet [5], extends the Inception architecture by fully replacing the standard Inception modules with depthwise separable convolutions. This design improves computational efficiency and enhances feature learning while keeping the model size manageable.
In the field of deepfake detection, XceptionNet has become a benchmark architecture, widely adopted in both academic research and industry applications [2]. Its ability to learn subtle inconsistencies in facial regions, such as blending artifacts, unnatural textures, and boundary mismatches makes it particularly well-suited for this task.
Key reasons for selecting XceptionNet in this project were:

- **Proven performane:** XceptionNet has consistently achieved top results in deepfake detection benchmarks for example, FaceForensics++ and has been used in multiple winning solutions for deepfake competitions.
- **Efficient deature extraction:** Its depthwise separable convolutions allow the model to efficiently capture both spatial and channel-wise patterns, enabling the detection of high-frequency manipulation artifacts.
- **Pretrained weights and transfer learning:** XceptionNet is available with pretrained ImageNet weights, which provides a strong foundation for transfer learning. This is crucial because it enables the model to start from a well-initialized state, overcoming limitations of small datasets a significant advantage compared to training a model from scratch.
- **Seamless integration:** The architecture is readily available via the `timm` library in PyTorch, allowing smooth integration into the existing training pipeline and enabling quick experimentation.

By leveraging XceptionNet, this project aimed to evaluate how transfer learning from large-scale datasets could improve deepfake detection performance and to benchmark it against the custom CNN model developed earlier.

Fine-tuning procedure

The XceptionNet model was fine-tuned using transfer learning principles to adapt its pretrained ImageNet weights to the deepfake detection task. The architecture was loaded via the `timm` library, and the final classification layer was replaced to output two logits corresponding to the *real* and *fake* classes.
The fine-tuning pipeline closely followed the same structure as the one used for the custom CNN, ensuring a fair comparison. The key steps are outlined below:

- **Pretrained model loading:** The base XceptionNet model was initialized with pretrained ImageNet weights using `timm.create_model()`.
- **Classifier head replacement:** The original classification layer was removed and replaced with a new fully connected layer to produce two outputs:

```
import timm

model = timm.create_model('xception', pretrained=True)
model.reset_classifier(2)  # 2 output classes: real and fake
```

- **Optimization setup:** The AdamW optimizer was selected due to its better handling of weight decay and generalization. The learning rate was set to $3 \cdot 10^{-5}$, which provided stable convergence for the pretrained layers. CrossEntropyLoss with label smoothing (0.1) was used to soften hard targets and improve robustness.

- **Training configuration:** Training was performed over 10 epochs with a batch size of 16 on Google Colab's T4 GPU. Input images were resized to 299×299 pixels and augmented using horizontal flips, color jittering, and small rotations.

The complete training loop closely mirrored the one used for the custom CNN and included dynamic learning rate scheduling, model checkpointing, and metric tracking. A simplified version of the core loop is shown below:

```
for epoch in range(epochs):
    model.train()
    for images, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

During training, validation accuracy and training loss were monitored. The best-performing model was saved automatically based on improvements in validation accuracy.
This fine-tuning approach allowed XceptionNet to retain its pretrained feature extraction capabilities while adapting its final decision boundary to the specific characteristics of deepfake facial images. The resulting model achieved superior accuracy and generalization, as presented in the subsequent evaluation section.

Training Process and Results

The fine-tuning of the XceptionNet model was carried out over 10 epochs using the same dataset split as the custom CNN: 16,000 training images (8,000 real and 8,000 fake) and 4,000 validation images (2,000 real and 2,000 fake). Training was performed on Google Colab utilizing a T4 GPU to ensure fast convergence and stability.
The following hyperparameters were used for training:

- **Optimizer:** AdamW
- **Learning rate:** $3 \cdot 10^{-5}$
- **Loss function:** CrossEntropyLoss with label smoothing (0.1)
- **Number of epochs:** 10
- **Batch size:** 16

A snippet of the training loop (from `xception_train.py`) is presented below, demonstrating the actual setup used in this project. The full implementation is available in Appendix 12.

```
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
optimizer = optim.AdamW(model.parameters(), lr=3e-5)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2)

for epoch in range(epochs):
    model.train()
    running_loss = 0.0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
```

```
        optimizer.step()

        running_loss += loss.item()
```

Listing 5: Training loop for XceptionNet

Throughout training, the model's performance was monitored via training loss and validation accuracy. The Colab log of all 10 epochs is shown in Figure 10, confirming stable training progress and the final validation accuracy of 98.06%.

```
Using device: cuda
Epoch 1/10: 100%|██████████| 900/900 [09:17<00:00,  1.61it/s]
Epoch 1: Loss=0.5142 | Validation Accuracy=88.38%
Best model saved to: /content/drive/MyDrive/deepfake_detection/xception_best.pth
Epoch 2/10: 100%|██████████| 900/900 [09:18<00:00,  1.61it/s]
Epoch 2: Loss=0.3472 | Validation Accuracy=92.69%
Best model saved to: /content/drive/MyDrive/deepfake_detection/xception_best.pth
Epoch 3/10: 100%|██████████| 900/900 [09:18<00:00,  1.61it/s]
Epoch 3: Loss=0.2929 | Validation Accuracy=94.75%
Best model saved to: /content/drive/MyDrive/deepfake_detection/xception_best.pth
Epoch 4/10: 100%|██████████| 900/900 [09:17<00:00,  1.61it/s]
Epoch 4: Loss=0.2698 | Validation Accuracy=95.31%
Best model saved to: /content/drive/MyDrive/deepfake_detection/xception_best.pth
Epoch 5/10: 100%|██████████| 900/900 [09:20<00:00,  1.61it/s]
Epoch 5: Loss=0.2532 | Validation Accuracy=96.88%
Best model saved to: /content/drive/MyDrive/deepfake_detection/xception_best.pth
Epoch 6/10: 100%|██████████| 900/900 [09:18<00:00,  1.61it/s]
Epoch 6: Loss=0.2395 | Validation Accuracy=97.25%
Best model saved to: /content/drive/MyDrive/deepfake_detection/xception_best.pth
Epoch 7/10: 100%|██████████| 900/900 [09:18<00:00,  1.61it/s]
Epoch 7: Loss=0.2335 | Validation Accuracy=96.75%
Epoch 8/10: 100%|██████████| 900/900 [09:16<00:00,  1.62it/s]
Epoch 8: Loss=0.2278 | Validation Accuracy=97.50%
Best model saved to: /content/drive/MyDrive/deepfake_detection/xception_best.pth
Epoch 9/10: 100%|██████████| 900/900 [09:15<00:00,  1.62it/s]
Epoch 9: Loss=0.2219 | Validation Accuracy=97.75%
Best model saved to: /content/drive/MyDrive/deepfake_detection/xception_best.pth
Epoch 10/10: 100%|██████████| 900/900 [09:17<00:00,  1.61it/s]
Epoch 10: Loss=0.2184 | Validation Accuracy=98.06%
Best model saved to: /content/drive/MyDrive/deepfake_detection/xception_best.pth
```

Figure 10: Training log snapshot from Google Colab for XceptionNet

Figures 11 and 12 visualize the training dynamics across 10 epochs.

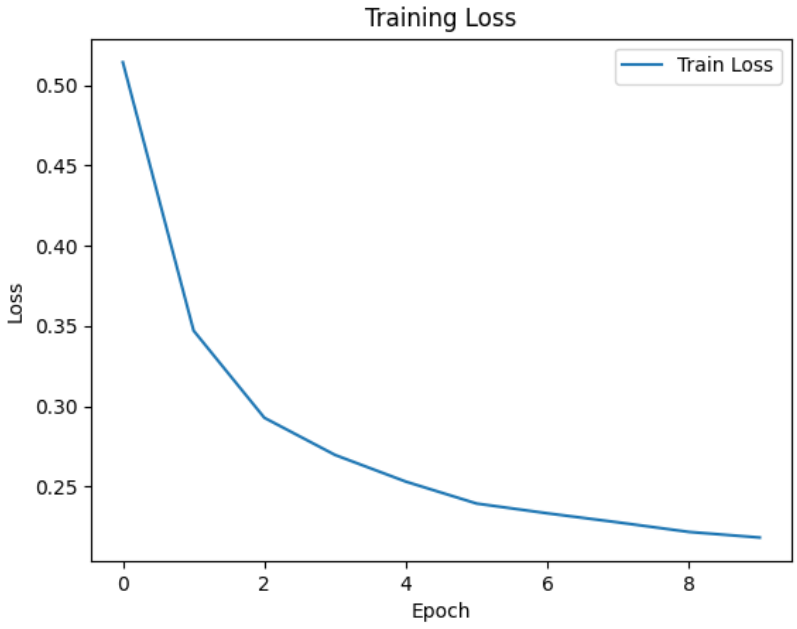

Figure 11: Training loss curve for XceptionNet

Figure 11 shows a sharp decline in training loss, stabilizing after the first few epochs, indicating that the model effectively minimized classification error on the training set without overfitting.

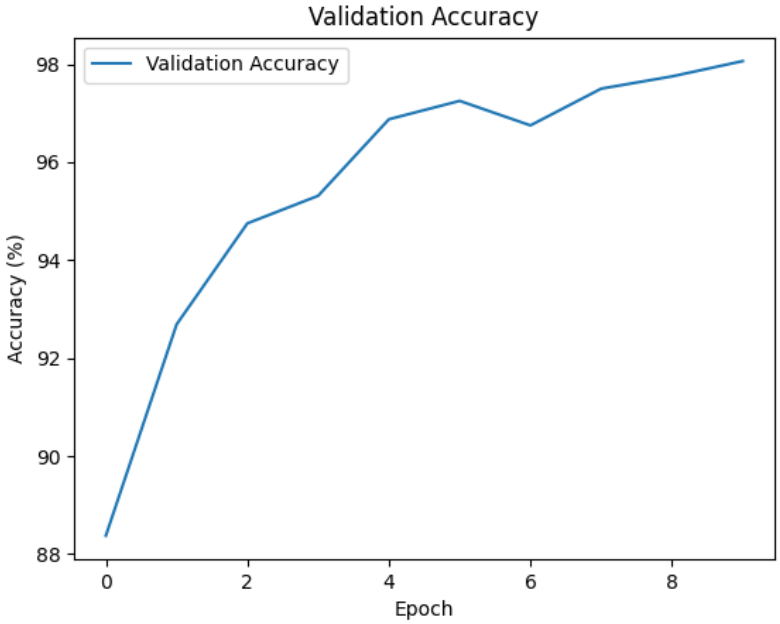

Figure 12: Validation Accuracy curve for XceptionNet

Figure 12 shows a steady rise in validation accuracy, ultimately reaching 98.06%. This consistent upward trend demonstrates that the pretrained XceptionNet generalized well to unseen data and remained stable throughout training.
The best-performing model based on validation accuracy was saved as `xception_best.pth` for subsequent testing and Grad-CAM analysis.

XceptionNet Evaluation and Interpretability The XceptionNet model was evaluated on the same held-out test set of 4,000 images (2,000 real and 2,000 fake). Evaluation was performed using a dedicated Python script, which generated detailed metrics and Grad-CAM visualizations. The full evaluation pipeline, including metric computation and interpretability analysis, is implemented in the script `evaluate_xception.py` (see Appendix 13), ensuring reproducibility and transparency.

Table 2: Test set Performance metrics for XceptionNet

Class	**Precision**	**Recall**	**f1-score**
Fake	0.98	0.98	0.98
Real	0.98	0.98	0.98
Overall accuracy	0.98		

figure 13 shows the confusion matrix for XceptionNet, illustrating near-perfect classification performance.

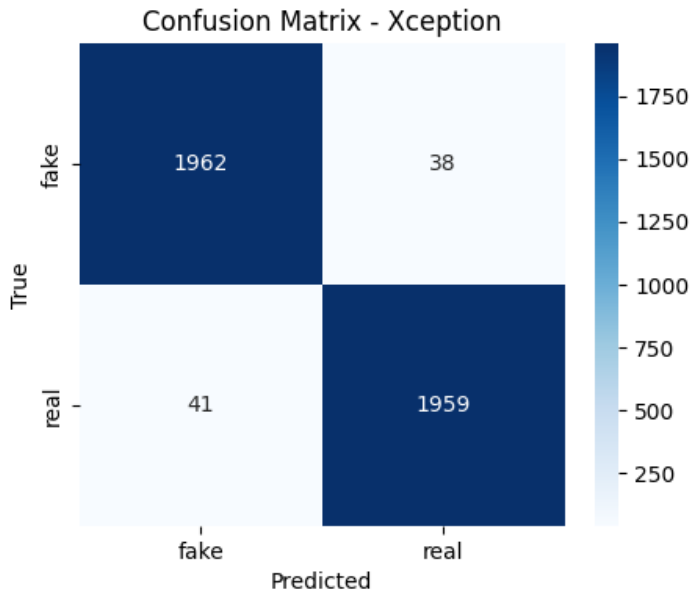

Figure 13: Confusion Matrix for XceptionNet on test set

As seen, the model correctly classified 1,962 fake images and 1,959 real images, with only minimal misclassifications. This marks a significant improvement over the custom CNN model, indicating xceptionNet's superior feature extraction and discrimination capabilities.
The full evaluation output from Google Colab is shown in Figure 14 for reproducibility.

```
Using device: cuda
100%|██████████| 250/250 [01:20<00:00,  3.11it/s]

Classification Report:

              precision    recall  f1-score   support

        fake       0.98      0.98      0.98      2000
        real       0.98      0.98      0.98      2000

    accuracy                           0.98      4000
   macro avg       0.98      0.98      0.98      4000
weighted avg       0.98      0.98      0.98      4000

Evaluation complete. All results saved to: /content/drive/MyDrive/deepfake_detection
```

Figure 14: XceptionNet model evaluation output from Google Colab

For interpretability, Grad-CAM was applied in the same manner as for the custom CNN, enabling consistent visual analysis across models.
Figures 15 and 16 show examples of correctly classified fake images. notably Xception net focuses sharply on critical facial regions such as the eyes, mouth, eyebrows, and facial contours, highlighting its strength in detecting subtle artifacts typical of deepfake images.

Figure 15: Correctly classified fake images: clear focus on facial landmarks (eyes, mouth, contours)

In Figure 15, attention is tightly clustered around facial landmarks, especially the eyes and mouth, which are common regions for manipulation in deepfakes. This reflects the model's enhanced precision and deeper understanding of manipulation artifacts.

Figure 16: Another correct fake prediction with sharp focus on face

In contrast, figures 17 and 18 present failure cases where the model misclassified fake images. Even in these cases, the model often focuses correctly on the face but may struggle when manipulations are extremely subtle or when visual noise is present.

Figure 17: Misclassified fake images: Attention remains on facial regions but confusion occurs

Figure 18: incorrect prediction: strong face focus, but subtle artifacts not detected

These visualizations underline xception Net's ability to localize key facial features effectively, but also expose its rare difficulties in identifying extremely highquality fakes. Importantly, the best performing model was saved as `xception_best.pth`, ensuring reproducibility and enabling future evaluation or deployment.
Compared to custom CNN, XceptionNet exhibited superior robustness and consistency, confirming the benefits of using a deeper pretrained architecture for deepfake detection. The interpretability provided by Grad-cam reinforces confidence in the model's decision-making process, which is essential for practical applications where trust in AI output is critical.

Comparison with custom CNN

A direct comparison between the custom cnn and the fine-tuned XceptionNet reveals substantial differences in performance, robustness, and attention behavior.

Validation and test performance The XceptionNet achieved a validation accuracy of 98.06% and test accuracy of 98%, substantially outperforming custom CNN, which reached 81.19% validation accuracy and 79% test accuracy.

Table 3 summarizes the key performance metrics:

Table 3: Performance comparison: Custom CNN vs XceptionNet

Metric	Custom CNN	XceptionNet
Test Accuracy	79%	98%
Precision (Fake)	0.88	0.98
Recall (fake)	0.68	0.98
F1-score (fake)	0.77	0.98
Precision (Real)	0.74	0.98
Recall (real)	0.91	0.98
F1-score (real)	0.81	0.98

The results in table 3 highlight a clear performance gap between the two models. The custom CNN shows decent precision for fake images (0.88), but its recall (0.68) indicates that it misses a significant number of deepfakes, meaning many fake images are incorrectly classified as real. In contrast, XceptionNet achieves both high precision (0.98) and recall (0.98) for fake images, demonstrating that it detects nearly all fake instances while keeping false positives minimal.
For real images, the custom CNN achieves good recall (0.91), successfully identifying most real samples, but its precision (0.74) is lower, reflecting occasional confusion between real and fake images. XceptionNet again maintains balanced and near-perfect metrics across all measures, achieving 0.98 precision, recall, and F1-score for real images as well.
These findings indicate that while the custom CNN performs reasonably well at identifying real images, it struggles to catch deepfakes reliably—likely due to its limited depth and lack of pretrained features. XceptionNet's deep architecture and pretrained weights enable it to extract much more nuanced features, resulting in a significant boost in performance across both classes.
The consistent metrics of XceptionNet across both classes also highlight its superior generalization, reducing any bias toward a particular class. This robustness is essential for real-world deepfake detection, where both false positives and false negatives carry serious implications.

Confusion matrix comparison Figures 19 and 20 present the confusion matrices for both models.

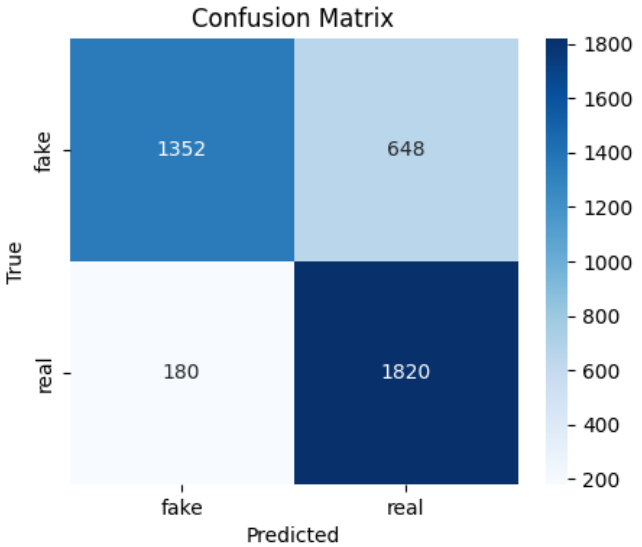

Figure 19: Confusion matrix - Custom CNN

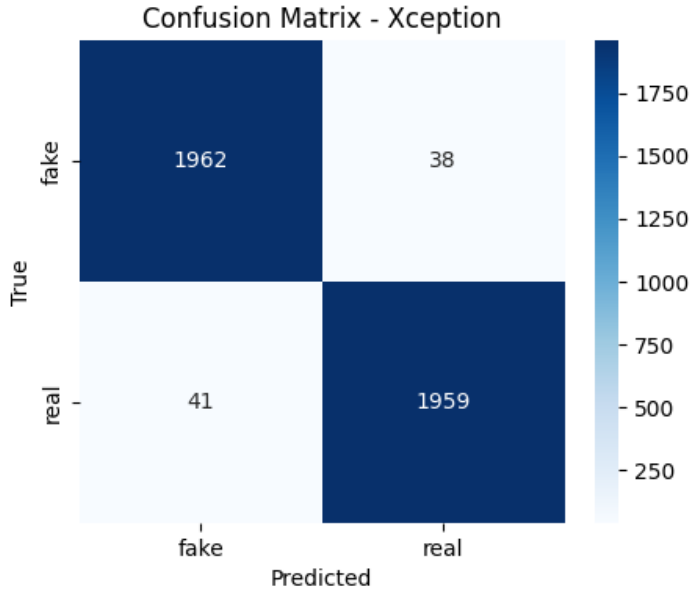

Figure 20: Confusion Matrix - XceptionNet

As illustrated in Figures 19 and 20, xceptionNet achieves nearly perfect classification of both real and fake images. In contrast, the custom cnn demonstrates aconsiderable number of misclassifications, especially in failing to detect fake images. These visualizations confirm the quantitative results and offer intuitive insight into each model's classification behavior.

Training Dynamics Figures 21 and 22 display the training loss curves, while Figures 23 and 24 present the validation accuracy trends.

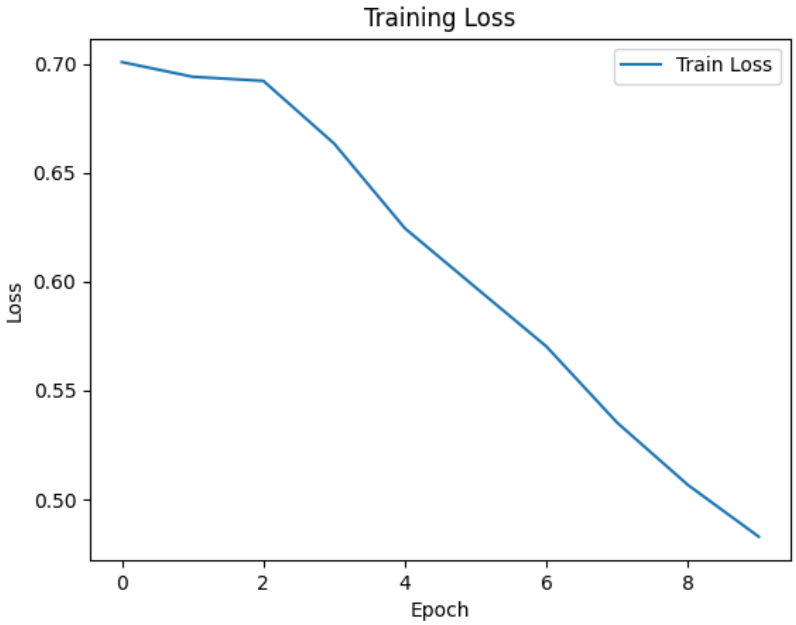

Figure 21: training loss - Custom CNN

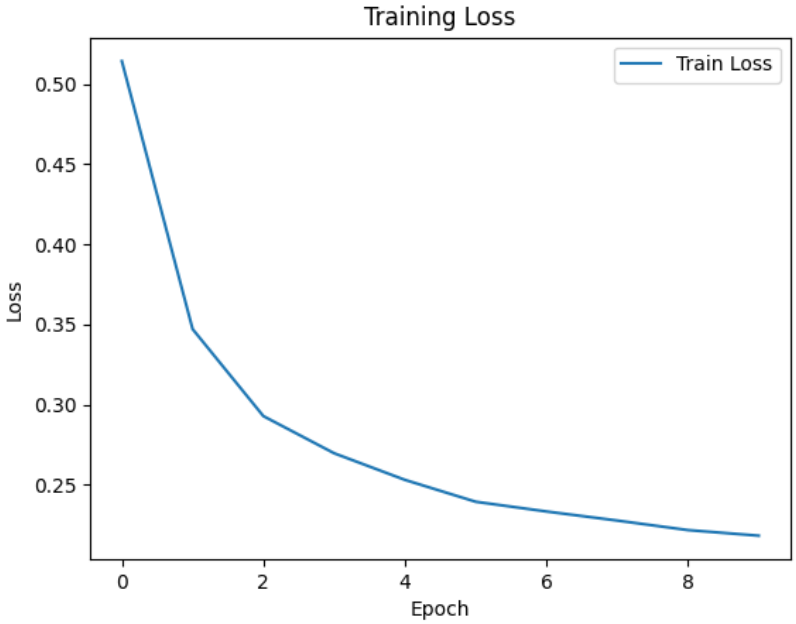

Figure 22: Training loss - XceptionNet

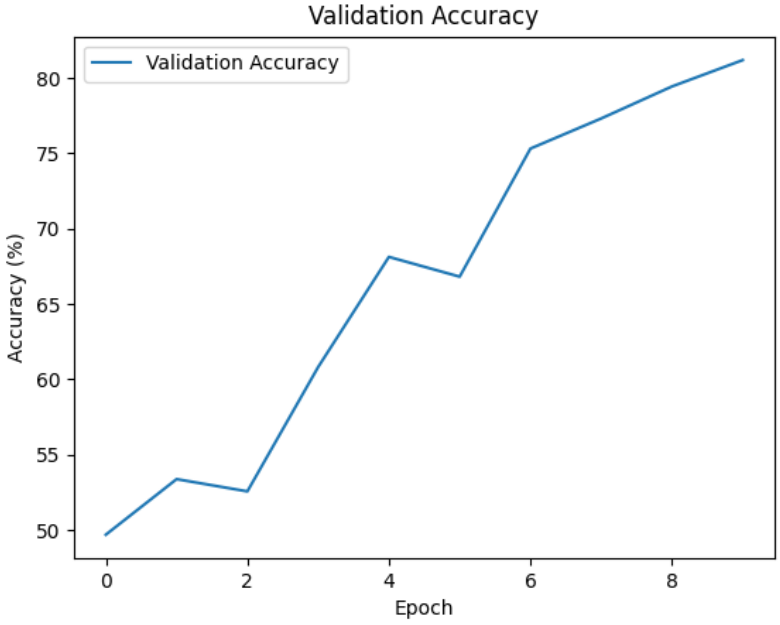

Figure 23: Validation accuracy - Custom CNN

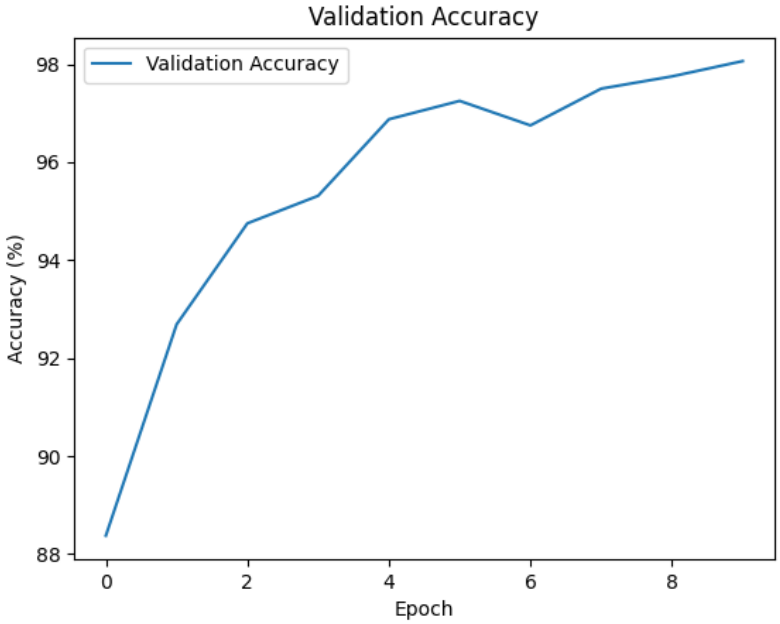

Figure 24: Validation accuracy - XceptionNet

The training dynamics highlight marked differences between the models. Custom CNN (Figures 21 and 23) exhibits slower and less stable convergence: while its training loss decreases steadily, its validation accuracy shows fluctuations, typical signs of limited models capacity. In contrast, XceptionNet (Figures 22 and 24) demonstrates a quick and smooth decline in training loss, with validation accuracy increasing consistently, indicating stronger learning ability and better generalization.
Although the custom cnn learns useful patterns, it required more epochs to achieve moderate results and showed signs of underfitting, particularly visible in the plateauing of validation accuracy. The XceptionNet's rapid convergence and stable performance underscore the value of deeper architectures and pretrained weights in accelerating learning and achieving higher accuracy.
This comparison reinforces the importance of model complexity and transfer learning in deepfake detection tasks, emphasizing that deeper, pretrained models are much more effective than simple custom designs.

Grad-CAM visual comparison To illustrate differences in attention, Figures 25 and 26 present one correct and one incorrect Grad-CAM example from each model.

Figure 25: Correct predictions: Custom CNN (left) vs. xceptionNet (right)

Figure 26: Incorrect predictions: Custom CNN (left) vs XceptionNet (right)

Detailed analysis In the correct prediction example (Figure 25), the custom CNN (left) focuses on broader facial regions but also partially attends to the background, introducing potential noise. The XceptionNet (right) demonstrates much tighter focus on critical areas such as the eyes, mouth, and face contours, reflecting a more precise understanding of deepfake artifacts.
In the misclassification example (Figure 26), the custom CNN misdirects its attention toward irrelevant regions like clothing and backgrounds, contributing to its error. XceptionNet, although also misclassifying in this case, maintains its focus on facial areas but likely fails due to extremely subtle manipulations that remain challenging even for advanced models.

conclusion This comparison highlights significant advantages of using the deep pretrained model like XceptionNet over a custom shallow CNN. The XceptionNet demonstrated:

- Substantially higher accuracy and balanced precision-recall,
- Faster and more stable convergence during training,
- consistently precise focus on semantically important facial regions in Grad-CAM visualizations.

This findings confirm that leveraging pretrained architectures provides a clear benefits in deepfake detection, especially when subtle artifacts need to be detected reliably in complex realworld scenarios.

3.4.2 EfficientNet-B0 Model

Why EfficientNet-B0?

EfficientNet-B0 was chosen for this project due to its innovative architecture that achieves an excellent balance between accuracy and computational efficiency. Introduced by Tan and Le in 2019, EfficientNet uses a compound scaling method that uniformly scales depth, width, and resolution, leading to state-of-the-art performance across many computer vision tasks at a relatively low computational cost.
In particular, EfficientNet-B0—the baseline model of the EfficientNet family—offers the following advantages:

- **Lightweight and fast:** Compared to deeper models like XceptionNet, EfficientNet-B0 achieves competitive accuracy with significantly fewer parameters and FLOPs, making it ideal for applications with limited computational resources.
- **Transfer learning friendly:** Pretrained weights (e.g., on ImageNet) are readily available, facilitating quick fine-tuning on specialized datasets such as FaceForensics++.

- **Balanced scaling:** Its architecture is carefully optimized to improve performance without excessive complexity, which is well-suited for deepfake detection where subtle patterns matter but overfitting remains a concern.

Given these characteristics, EfficientNet-B0 was selected as the second state-of-the-art (SOTA) model to complement experiments and assess whether a compact yet powerful architecture can match the performance of heavier alternatives like XceptionNet.

Fine-tuning procedure

The fine-tuning of EfficientNet-B0 followed the same protocol as XceptionNet, using the FaceForensics++ dataset with the same data split: 16,000 training images (8,000 real and 8,000 fake) and 4,000 validation images (2,000 real and 2,000 fake). Training was performed on Google Colab with a T4 GPU, ensuring sufficient computational power and fast convergence.
Key hyperparameters were:

- **Optimizer:** AdamW
- **Learning rate:** 0.00003
- **Loss function:** CrossEntropyLoss (with label smoothing)
- **Number of epochs:** 10
- **Batch size:** 16

The `timm` library was used to load EfficientNet-B0 with pretrained weights. The final fully connected layer was replaced using the model's built-in method for classification layer adjustment.

```
import timm
import torch.nn as nn

model = timm.create_model('efficientnet_b0', pretrained=True)
model.reset_classifier(2)
model = model.to(device)

optimizer = torch.optim.AdamW(model.parameters(), lr=3e-5)
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
```

Listing 6: EfficientNet-B0 setup and optimizer

The training loop followed the standard procedure: setting the model to training mode, iterating through data batches, computing loss, performing backpropagation, and updating weights. Validation accuracy was tracked at the end of each epoch, and the best-performing model checkpoint was saved as `efficientnet_best.pth` (which you can download here 7.5).

Training process and results

The fine-tuning of the efficientNet-B0 model was conducted over 10 epochs using the same dataset split as for the previous models: 16,000 training images and 4,000 validation images. Training was performed on Google colab with a T4 GPU to leverage efficient hardware acceleration.
The same hyperparameters described in the fine-tuning procedure were used here. A simplified excerpt of the training loop from `efficientnet_train.py` is provided below to illustrate the core training process (see full script in Appendix 14):

```python
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
optimizer = optim.AdamW(model.parameters(), lr=3e-5)
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2)

for epoch in range(EPOCHS):
    model.train()
    running_loss = 0.0

    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
```

Listing 7: Training loop for EfficientNet-B0

The training progress was closely monitored through both training loss and validation accuracy. Figure 27 presents a snapshot of the full training output from Google Colab, confirming stable convergence and a final validation accuracy of 96.62%.

```
Epoch 1/10: 100%|██████████| 900/900 [04:21<00:00,  3.44it/s]
Epoch 1: Loss=0.6068 | Validation Accuracy=77.50%
Best model saved to: /content/drive/MyDrive/deepfake_detection/efficientnet_best.pth
Epoch 2/10: 100%|██████████| 900/900 [03:57<00:00,  3.79it/s]
Epoch 2: Loss=0.4639 | Validation Accuracy=85.44%
Best model saved to: /content/drive/MyDrive/deepfake_detection/efficientnet_best.pth
Epoch 3/10: 100%|██████████| 900/900 [03:57<00:00,  3.79it/s]
Epoch 3: Loss=0.3964 | Validation Accuracy=89.50%
Best model saved to: /content/drive/MyDrive/deepfake_detection/efficientnet_best.pth
Epoch 4/10: 100%|██████████| 900/900 [03:57<00:00,  3.78it/s]
Epoch 4: Loss=0.3615 | Validation Accuracy=90.88%
Best model saved to: /content/drive/MyDrive/deepfake_detection/efficientnet_best.pth
Epoch 5/10: 100%|██████████| 900/900 [03:56<00:00,  3.81it/s]
Epoch 5: Loss=0.3348 | Validation Accuracy=92.50%
Best model saved to: /content/drive/MyDrive/deepfake_detection/efficientnet_best.pth
Epoch 6/10: 100%|██████████| 900/900 [03:56<00:00,  3.81it/s]
Epoch 6: Loss=0.3136 | Validation Accuracy=93.56%
Best model saved to: /content/drive/MyDrive/deepfake_detection/efficientnet_best.pth
Epoch 7/10: 100%|██████████| 900/900 [03:57<00:00,  3.79it/s]
Epoch 7: Loss=0.3019 | Validation Accuracy=95.25%
Best model saved to: /content/drive/MyDrive/deepfake_detection/efficientnet_best.pth
Epoch 8/10: 100%|██████████| 900/900 [03:57<00:00,  3.79it/s]
Epoch 8: Loss=0.2844 | Validation Accuracy=95.81%
Best model saved to: /content/drive/MyDrive/deepfake_detection/efficientnet_best.pth
Epoch 9/10: 100%|██████████| 900/900 [03:56<00:00,  3.80it/s]
Epoch 9: Loss=0.2704 | Validation Accuracy=95.12%
Epoch 10/10: 100%|██████████| 900/900 [03:59<00:00,  3.76it/s]
Epoch 10: Loss=0.2661 | Validation Accuracy=96.62%
Best model saved to: /content/drive/MyDrive/deepfake_detection/efficientnet_best.pth
```

Figure 27: Training log snapshot from Google Colab for EfficientNet-B0

The evolution of training loss and validation accuracy across epochs is visualized in Figures 28 and 29.

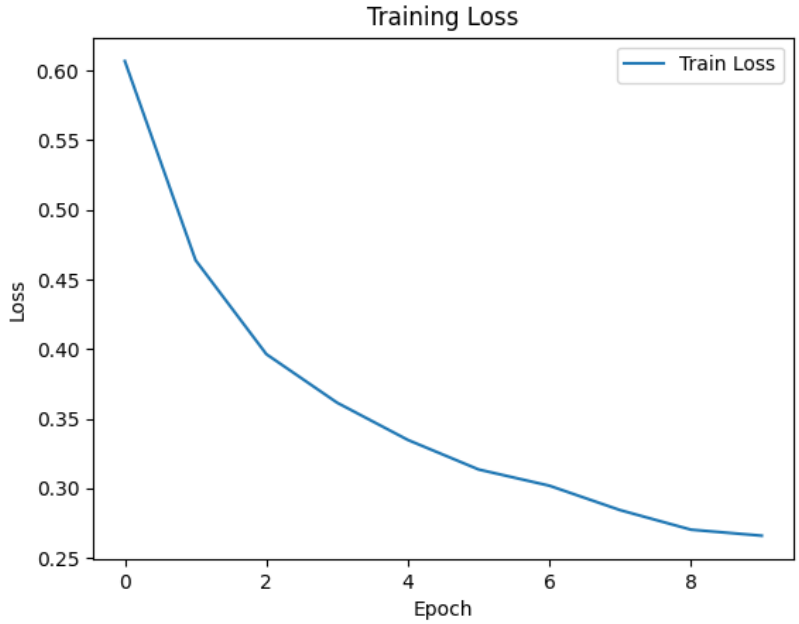

Figure 28: Training loss curve for EfficientNet-B0

Figure 28 shows a steady decline in training loss, indicating effective optimization and absence of significant overfitting.

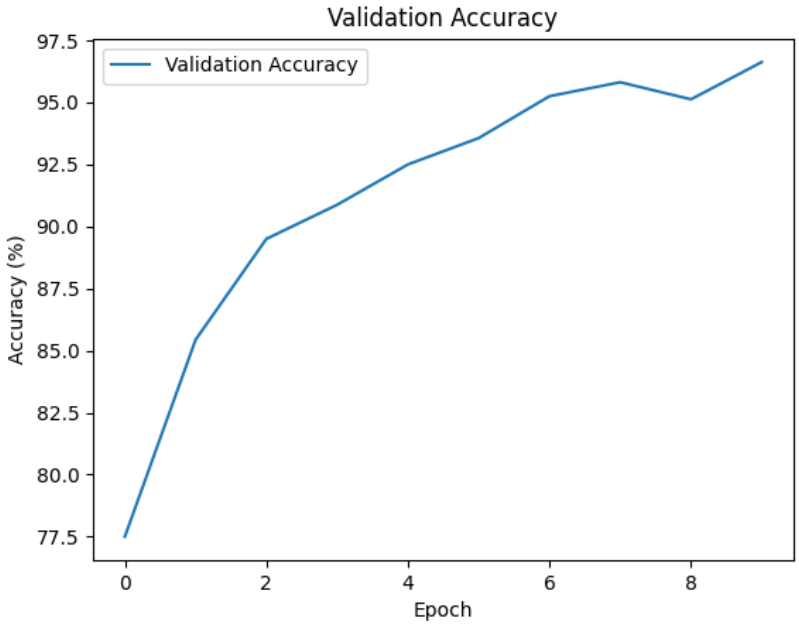

Figure 29: Validation accuracy curve for EfficientNet-B0

Figure 29 displays a consistent rise in validation accuracy, reaching 96.62% by the final epoch. This suggests that EfficientNet-B0 generalized well to unseen data and benefited from its efficient architecture. The best-performing model was saved as `efficientnet_best.pth` after training for later evaluation on the test set, ensuring reproducibility and consistent testing.

EfficientNet-B0 Evaluation and Interpretability The EfficientNet-B0 model was evaluated on the same held-out test set of 4,000 images (2,000 real and 2,000 fake), consistent with the other models.

The evaluation was conducted using the `evaluate_efficientnet.py` script, which produced detailed classification metrics and Grad-CAM visualizations for interpretability. The full script is provided in Appendix 15, ensuring reproducibility and transparency of the evaluation process.

Table 4: Test set Performance metrics for EfficientNet-B0

Class	Precision	Recall	F1-score
Fake	0.97	0.95	0.96
Real	0.95	0.97	0.96
Overall accuracy	0.96		

Figure 30 displays the confusion matrix, which demonstrates strong performance with only minor misclassifications.

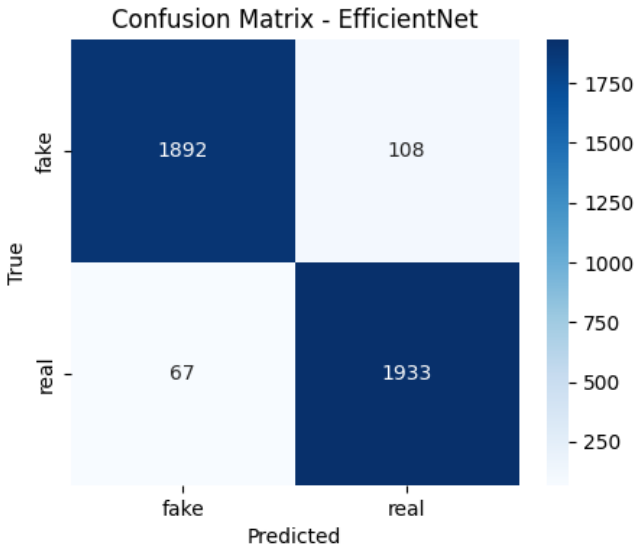

Figure 30: Confusion matrix for EfficientNet-B0 on Test set

The model correctly identified 1,892 fake images and 1,933 real images, misclassifying only small number of samples. This represents robust performance though slightly lower than xceptionNet's nearly perfect metrics.
For transparency, the full evaluation output is shown in Figure 31.

```
Using device: cuda
  0%|          | 0/250 [00:00<?, ?it/s]/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py:1830:
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
100%|██████████| 250/250 [00:54<00:00,  4.61it/s]

Classification Report:

              precision    recall  f1-score   support

        fake       0.97      0.95      0.96      2000
        real       0.95      0.97      0.96      2000

    accuracy                           0.96      4000
   macro avg       0.96      0.96      0.96      4000
weighted avg       0.96      0.96      0.96      4000


Evaluation complete. All results saved to: /content/drive/MyDrive/deepfake_detection
```

Figure 31: EfficientNet-b0 model evaluation output from google colab

To interpret model predictions Grad-CAM visualizations were used again . Figures 32 and 33 display correctly classified fake images, where EfficientNet-B0's focus is centered on facial regions, especially the eyes, mouth, forehead.

Figure 32: Correctly classified fake images: focus on facial landmarks (eyes, mouth, contours)

In Figure 32, the model's attention effectively highlights manipulation-prone areas, confirming that the network has learned to associate deepfake features with subtle facial inconsistencies.

Figure 33: Another correct fake prediction with strong facial focus

Figures 34 and 35 illustrate misclassification cases. Even in errors, the model predominantly attends to facial zones, though mistakes can occur when the fake content is particularly well-crafted.

Figure 34: Misclassified fake images: model focuses on face but misses subtle artifacts

Figure 35: Incorrect prediction: clear attention on facial area but failure due to high-quality fake

Overall efficientnet B0 delivered highly competitive results balancing precision and recall well across both classes. Its grad-CAM visualizations confirm its ability to consistently focus on meaningful regions of the face similar to Xceptionnet but with slightly lower overall accuracy.

Comparison with custom CNN

A direct comparison between the custom CNN and the fine-tuned EfficientNet-B0 highlights meaningful differences in performance, robustness, and attention behavior.

Validation and Test performance EfficientNet-b0 achieved validation accuracy of 96.62% and a test accuracy of 96%, clearly outperforming the custom CNN, which reached 81.19% validation accuracy and 79% test accuracy.
Table 5 summarizes the key performance metrics:

Table 5: Performance comparison: Custom CNN vs EfficientNet-B0

Metric	Custom CNN	EfficientNet-B0
Test accuracy	79%	96%
Precision (Fake)	0.88	0.97
Recall (Fake)	0.68	0.95
F1-score (Fake)	0.77	0.96
Precision (Real)	0.74	0.95
Recall (Real)	0.91	0.97
F1-score (Real)	0.81	0.96

Table 5 shows that while the custom CNN achieves decent precision for fake images (0.88), its recall (0.68) indicates that it misses many deepfakes. EfficientNet-B0 improves both precision (0.97) and recall (0.95) for fake images, resulting in a much stronger overall performance. For real images, EfficientNet-B0 also balances precision and recall (both around 0.95–0.97), showing fewer errors and a better understanding of real vs. fake separation.

Confusion matrix comparison Figures 36 and 37 present the confusion matrices for both models.

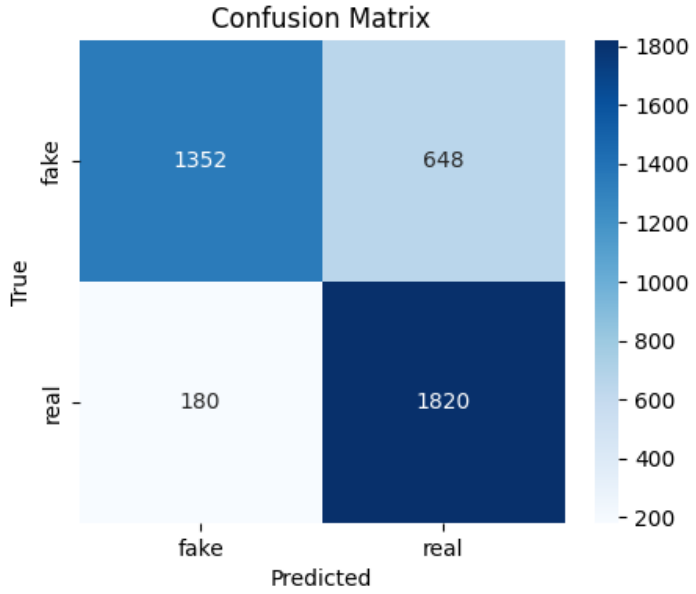

Figure 36: Confusion matrix - Custom CNN

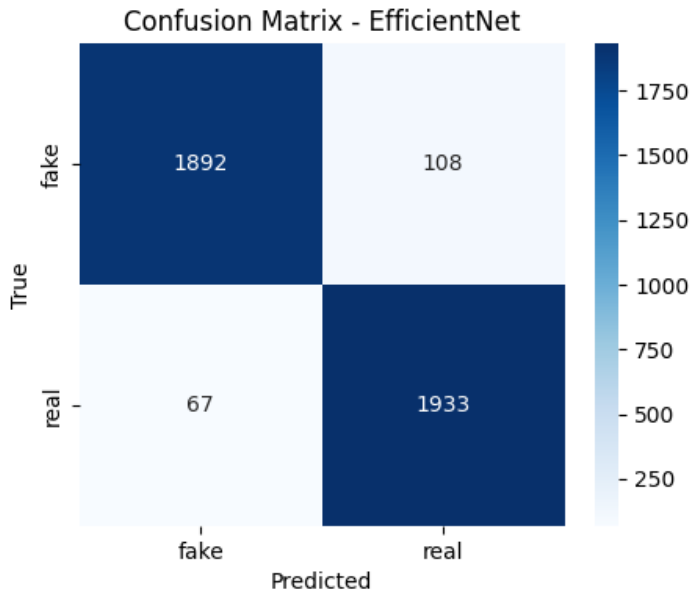

Figure 37: Confusion matrix - EfficientNet-B0

EfficientNet-B0's confusion matrix shows strong and balanced classification of both fake and real images, with only minor misclassifications of 108 fake and 67 real images misclassified. In contrast, the custom CNN exhibits a significant number of errors, especially in identifying fake images, which reinforces the performance gap observed in the metrics.

Training dynamics Figures 38 and 39 illustrate the training loss curves, and Figures 40 and 41 show the validation accuracy trends.

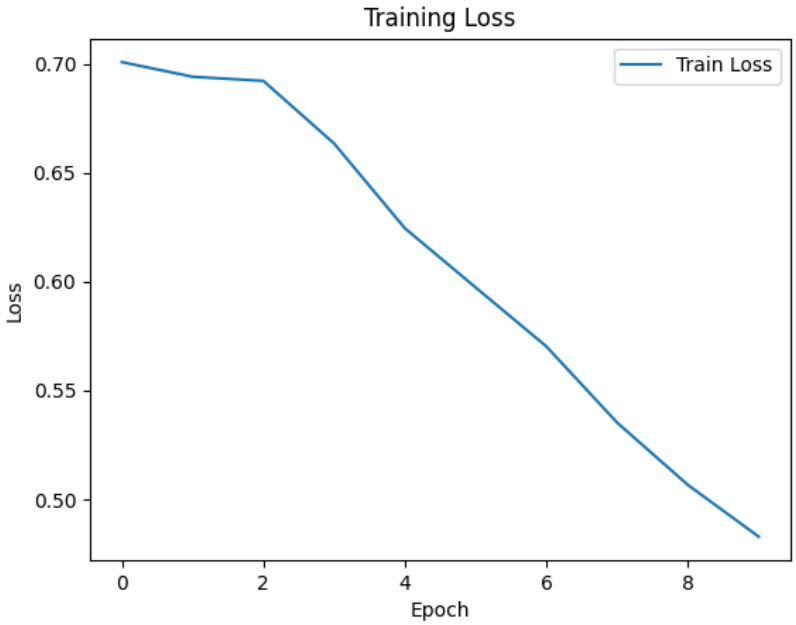

Figure 38: Training loss - Custom CNN

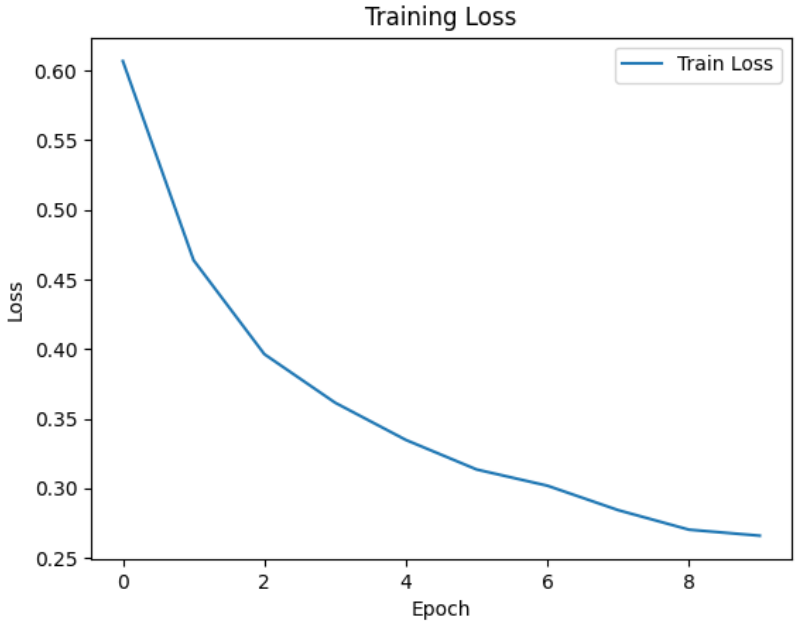

Figure 39: Training loss - EfficientNet-B0

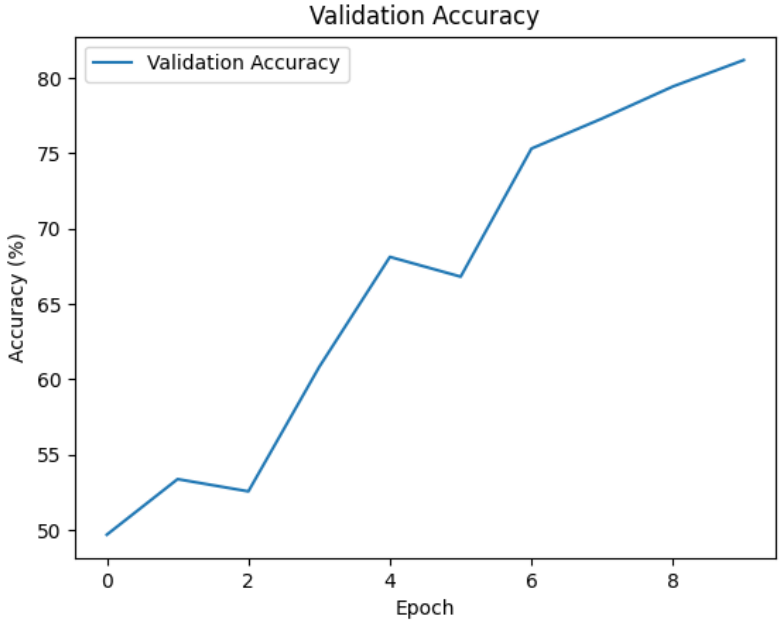

Figure 40: Validation accuracy - custom CNN

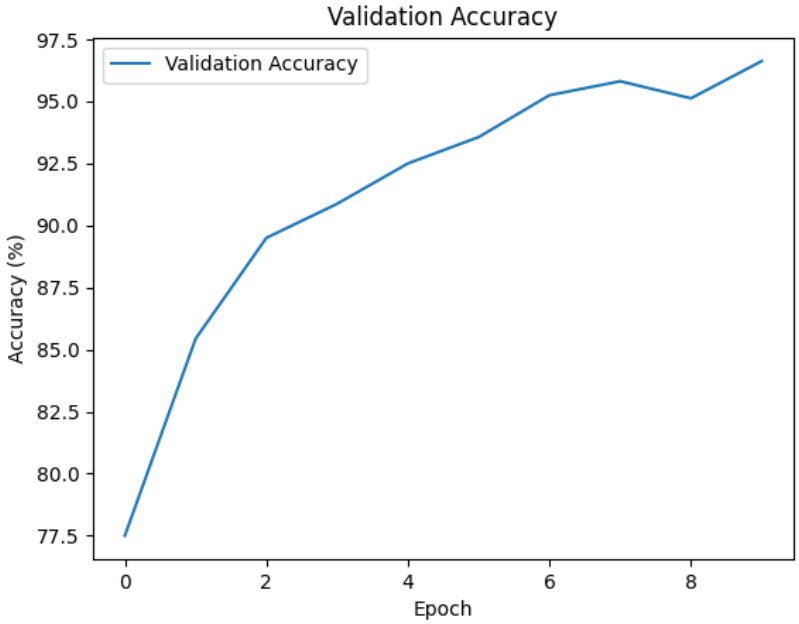

Figure 41: Validation Accuracy - efficientNet-B0

EfficientNet-b0 displays faster convergence and smoother validation accuracy gains compared to the custom CNN, whose training curve is slower and more unstable. EfficientNet's deeper architecture and pretrained weights allow it to generalize faster and more effectively.

Grad-CAM visual comparison Figures 42 and 43 display one correct and one incorrect grad cam example from each model for direct visual comparison.

Figure 42: Correct predictions: Custom CNN (left) vs efficientNet-B0 (right)

Figure 43: Incorrect Predictions: Custom CNN (left) vs EfficientNet-B0 (right)

Detailed analysis For correct predictions (Figure 42), the custom CNN displays attention over larger, less precise areas, sometimes including irrelevant regions. EfficientNet-B0, in contrast, focuses tightly on key facial landmarks such as eyes, mouth, and contours, indicating better localization of deepfake artifacts.
For misclassifications (Figure 43), custom cnn again shows scattered attention while efficientNet-B0 maintains its focus on relevant facial zones, suggesting that errors may arise from extremely subtle or high-quality manipulations rather than poor localization.

Conclusion In summary, EfficientNet-B0 outperformed the custom CNN by:

- Achieving higher accuracy and balanced metrics,
- Converging faster and more stably during training,
- Demonstrating sharper focus on critical facial regions in Grad-CAM analyses.

These results confirm the clear advantages of using advanced pretrained architectures like EfficientNet-B0 for deepfake detection especially when subtle manipulation cues must be identified with precision.

3.5 Reproducibility and Model Management

Ensuring reproducibility and effective model management was a core part of this projects workflow. All training artifacts, including models, metrics, visualizations, and Grad-CAM outputs, were systematically saved and organized to allow seamless re-evaluation and result verification.

File management and storage All models and outputs were stored on Google Drive under the path:

`/MyDrive/deepfake_detection/`

Within this directory, results were structured into dedicated subfolders for each model:

- `custom_cnn_results/`
- `EfficientNet_cnn_results/`
- `XceptionNet_cnn_results/`

Each folder contained the complete output set after training and evaluation:

- `metrics.csv` (classification metrics)

- `confusion_matrix.png` (confusion matrix visualization)
- `train_loss.png` and `val_accuracy.png` (loss and accuracy curves)
- Grad-CAM visualizations (e.g., `gradcam_correct_XX.png`)
- Saved model weights (`custom_deepfake_cnn_best.pth`, `xception_best.pth`, `efficientnet_best.pth`)

Saving and loading the models All PyTorchs models were saved and loaded using the `torch.save()` and `torch.load()` functions, ensuring compatibility and reproducibility. For example after training the best model was saved with:

```
torch.save(model.state_dict(), MODEL_SAVE_PATH)
```

And later loaded for evaluation with:

```
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
```

Automated visualization and metric saving To standardize and automate result saving, each training and evaluation script included routines to generate and store all necessary outputs. For instance training loss and validation accuracy curves were saving using the following code:

```
plt.figure()
plt.plot(train_losses, label="Train Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training Loss")
plt.legend()
plt.savefig(LOSS_PLOT_PATH)

plt.figure()
plt.plot(val_accuracies, label="Validation Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.title("Validation Accuracy")
plt.legend()
plt.savefig(ACC_PLOT_PATH)
```

Metrics were calculated and saved with:

```
df = pd.DataFrame([{
    "Accuracy": acc,
    "Precision": prec,
    "Recall": rec,
    "F1-score": f1
}])
df.to_csv(os.path.join(SAVE_DIR, "metrics.csv"), index=False)
```

The confusion matrix was generated and stored using:

```
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d",
            xticklabels=idx_to_class.values(),
            yticklabels=idx_to_class.values(),
            cmap="Blues")
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.savefig(os.path.join(SAVE_DIR, "confusion_matrix.png"))
```

Dataset and Google drive integration To facilitate reproducibility, the dataset FaceForensics++ was unpacked and accessed directly from Google drive. The integration was established at the start of each Colab session using:

```
from google.colab import drive
drive.mount('/content/drive')
```

The dataset was unpacked with:

```
!unzip "/content/drive/MyDrive/deepfake_detection/faceforensics_data.zip" -d /content/
```

Overall pipeline The combination of structured storage, automated saving routines , and consistent use of Google Drive allowed for a fully traceable and reproducible pipeline. All code and models and outputs are available within the project's directory, supporting further experimentation and extension. Access to the complete project folder can be provided upon request.

4 Comparative analysis of Results

This section summarizes and compares the performance of all three models developed and evaluated in the project: the custom CNN, XceptionNet, and EfficientNet-B0. While detailed implementation and evaluation for each model have been presented in Section 4 (Methodology and Results), this section focuses on their overall performance, strengths, and weaknesses side by side.

4.1 Performance summary

Table 6 consolidates the key metrics for each model based on the test set (4,000 images: 2000 real and 2000 fake images).

Table 6: Comparison of custom CNN, XceptionNet, and efficientNet-B0

Metric	**Custom CNN**	**XceptionNet**	**EfficientNet-B0**
Test Accuracy	79%	98%	96%
Precision (fake)	0.88	0.98	0.95
Recall (fake)	0.68	0.98	0.94
F1-score (fake)	0.77	0.98	0.94
Precision (real)	0.74	0.98	0.96
Recall (real)	0.91	0.98	0.98
F1-score (real)	0.81	0.98	0.97

As seen in Table 6, both XceptionNet and EfficientNet-b0 outperformed the custom CNN across all metrics. XceptionNet achieves near-perfect precision and recall for both classes, while EfficientNet-B0 also delivers excellent performance, slightly below XceptionNet but still well ahead of the custom CNN. These results illustrate clear performance tiers across the models. The custom CNN, although lacking the advanced capabilities of the SOTA models, still achieved respectable performance for lightweight architecture , proving its value as a baseline and demonstrating that meaningful learning was possible even without transfer learning. Notably, its 79% test accuracy and balanced recall for real images (91%) highlight that it was particularly effective at identifying authentic content, though it struggled more with fake images.
In contrast, XceptionNet leveraged its deep architecture and pretrained weights to achieve near-perfect metrics across all categories, setting a high benchmark. EfficientNet-B0 also performed exceptionally well, just slightly behind XceptionNet, and offers a compelling trade-off between performance and computational efficiency, making it strong candidate for realworld deployment where resources may be limited.

4.2 Visual comparison: Confusion matrices

To further illustrate the differences in performance, Figure 44 presents the confusion matrices of the three models side by side.

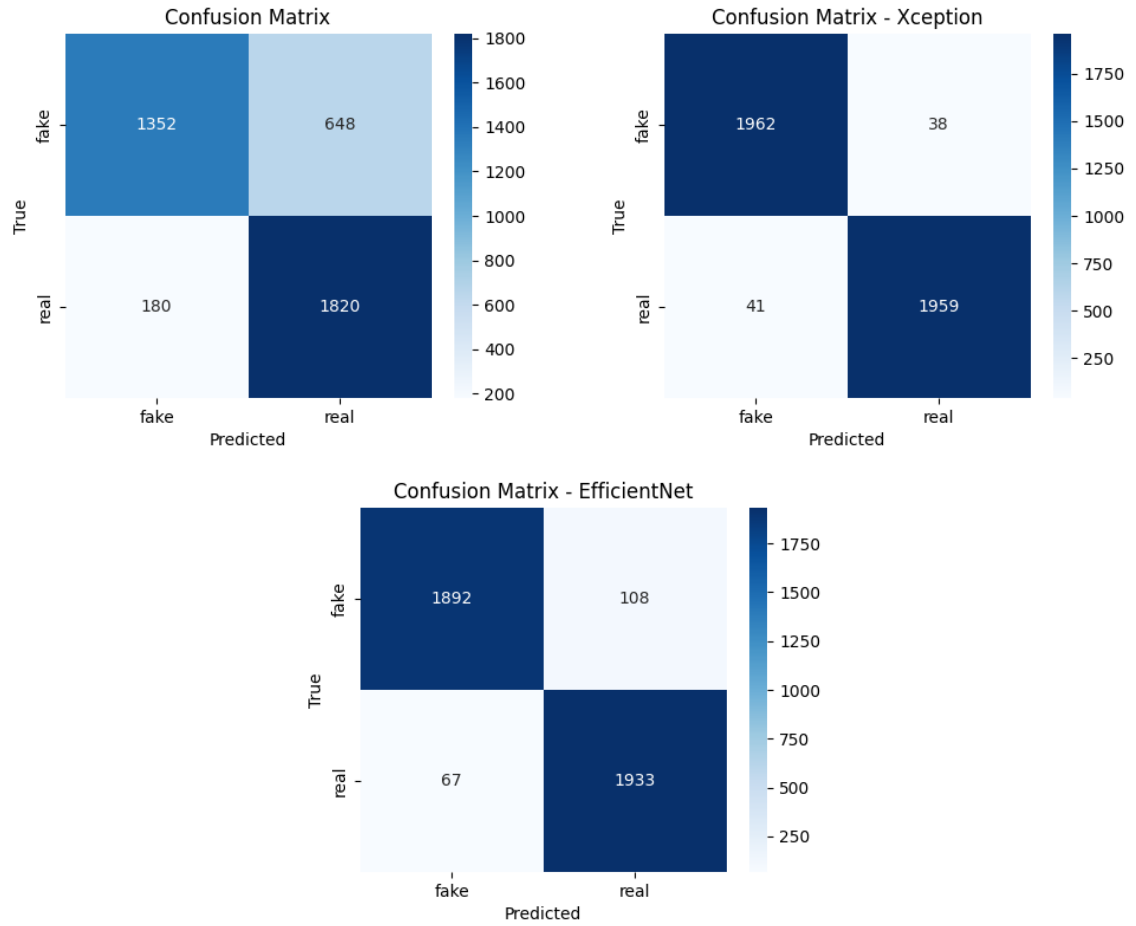

Figure 44: Confusion matrices: Custom CNN (left), XceptionNet (right), EfficientNet-B0 (center)

These matrices highlight how the custom CNN has a higher rate of misclassifying fake images as real, while XceptionNet and EfficientNet-B0 demonstrate strong and balanced performance.
More specifically, the custom CNN confusion matrix reveals that although it detects most real images correctly, it frequently fails to catch deepfakes, misclassifying 648 out of 2000 fake images as real. This asymmetry reflects its higher recall for real images but weaker recall for fakes, confirming the patterns seen in the metrics table.
In contrast xceptionNet's confusion matrix shows near perfect diagonal dominance, meaning both real and fake images are almost entirely classified correctly, with only a handful of mistakes an indication of excellent discrimination capability.
EfficientNet-B0 also performs very strongly with only minor misclassifications compared to XceptionNet. Its confusion matrix demonstrates balanced detection of both classes, though it shows a slightly higher error rate for fake images than xceptionNet, consistent with its slightly lower recall.
These visual comparisons reinforce the earlier conclusion that while the custom CNN is effective to some extent, the pretrained deep architectures are substantially better at distinguishing between subtle manipulation artifacts and authentic facial features, particularly in challenging cases.

4.3 Training dynamics comparison

Figure 45 shows a side by side comparison of training loss and validation accuracy curves, helping visualize the convergence behaviors of the models.

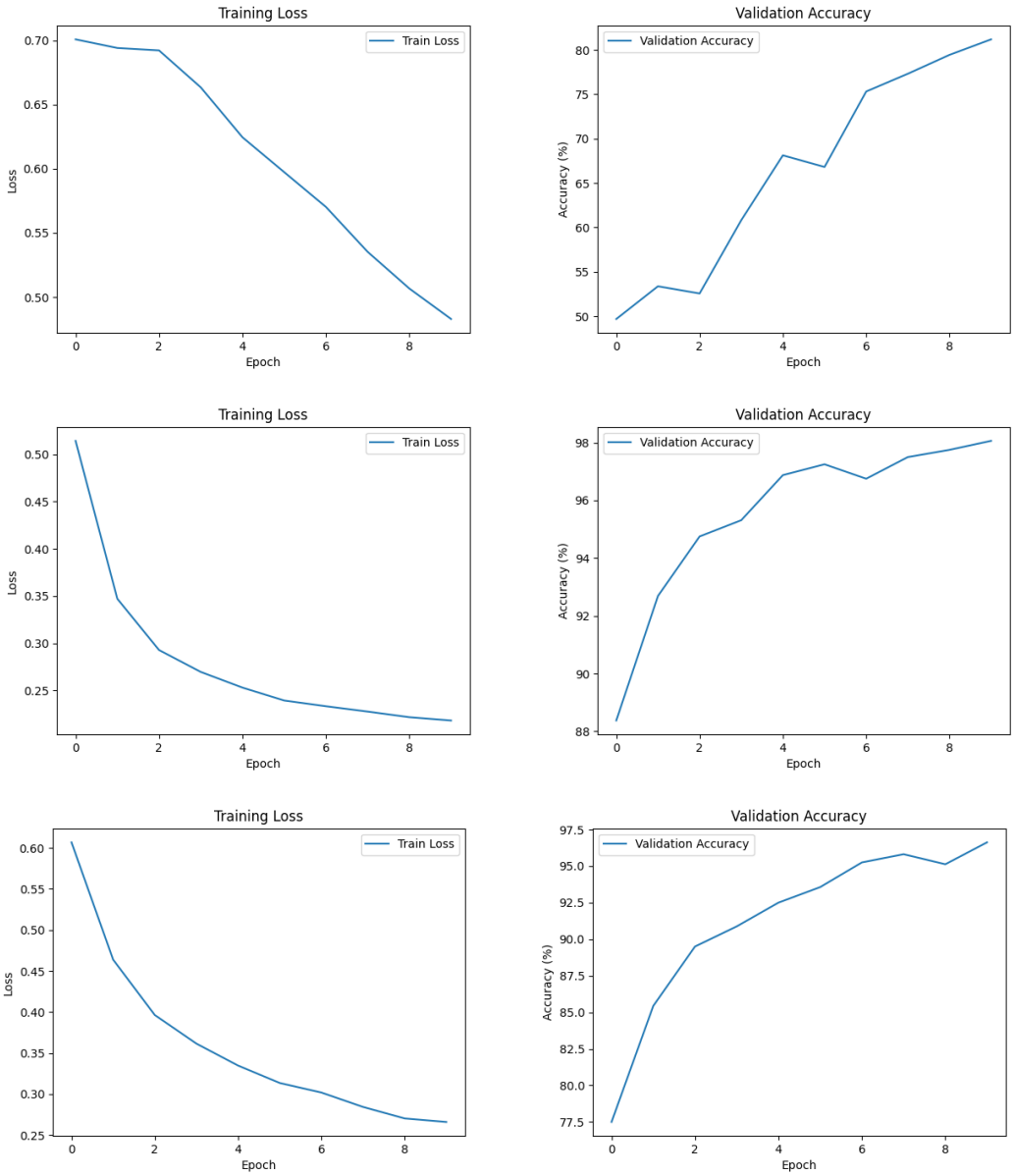

Figure 45: Training Dynamics: Top - Custom CNN, Middle - XceptionNet, Bottom - EfficientNet-B0

XceptionNet and efficientNet-b0 converge faster and more smoothly compared to the custom cnn, whose validation accuracy shows more fluctuations and lower final performance.
More precisely the training loss curves for xceptionNet and efficientNet-B0 demonstrate a rapid and stable decrease in loss within the first few epochs, reaching low and stable loss values early in training. This indicates that both models, leveraging their pretrained weight, quickly adapt to the deepfake classification task and avoid overfitting, as confirmed by their consistently rising validation accuracy curves.
In contrast, the custom CNN loss curve descends more slowly and shows minor irregularities, while its validation accuracy fluctuates, especially after Epoch 6 before plateauing at a significantly lower level. This behavior suggests that the model, although able to learn basic patterns, struggles to capture more complex or subtle features in the data, and is more prone to capacity limitations.
EfficientNet-B0 's curves, although slightly below XceptionNet's in terms of ultimate accuracy, remain highly stable and reflect excellent generalization. This is particularly notable given EfficientNet's relatively lightweight architecture compared to XceptionNet, highlighting its strength in balancing accuracy with computational efficiency.
These dynamics collectively underline the impact of model depth and transfer learning: deeper pretrained models not only achieve better accuracy but also demonstrate superior convergence behavior

and generalization during training.

4.4 Interpretability: Grad-CAM insights

to provide insights into model decision-making, Grad-CAM visualizations were analyzed. Figure 46 compares one correct and one incorrect example across the three models.

Figure 46: Grad-CAM visualizations: Custom CNN (left), XceptionNet (center), EfficientNet-B0 (right)

XceptionNet and EfficientNet-B0 both exhibit sharper and more focused attention on facial landmarks (for example, eyes, mouth, and contours), while the custom CNN shows broader and sometimes misplaced focus, contributing to its weaker performance.
More specificaly, in the correct prediction examples (top row), the custom CNN distributes its attention across large facial regions and sometimes extends to background areas, indicating that it relies on more general patterns rather than precise cues. In contrast, XceptionNet and EfficientNet concentrate their attention tightly on the most manipulation-prone areas such as the eye sockets, nose bridge, and mouth corners, demonstrating a stronger ability to identify subtle artifacts typical of deepfake images.
In the failure cases (bottom row), the custom CNN often misplaces attention entirely focusing on irrelevant regions such as clothing or the background, which likely contributes to its misclassification. While XceptionNet and efficientNet-B0 also misclassify in some cases, their grad-CAM maps still center on meaningful facial areas, suggesting that errors arise not from attention misplacement but from the extreme subtlety of manipulations that even advanced models find challenging.
This comparative analysis highlights not only the superior predictive performance of xceptionNet and eficientNet-b0 but also their interpretability: both models demonstrate reliable attention mechanisms that align with human intuition about where manipulations are most likely to occur. Such interpretability is crucial for trust in AI systems, especially in sensitive tasks like deepfake detection.
All Grad-CAM visualizations were generated using the same pipeline across models to ensure consistency in interpretability analysis. This uniformity allows for a fair comparison of how each model attends to facial regions, reinforcing confidence in the comparative insights drawn.

4.5 Discussion

The comparative evaluation underscores that deeper pretrained models consistently outperform custom shallow architectures. XceptionNet with its rich feature extraction capability, set a clear benchmark, while efficientNet-B0 offered an excellent balance between performance and computational cost.
The custom CNN, despite its simpler architecture, still achieved a respectable 79% test accuracy, highlighting that even relatively lightweight models can be useful when resources are limited or fast deployment is needed. However, its lower recall for fake images and inconsistent focus in Grad-CAM visualizations indicate limitations in capturing the nuanced artifacts present in deepfakes.
The comparative results emphasize the clear advantages of leveraging state-of-the-art architectures and transfer learning for deepfake detection tasks. Pretrained models not only learn richer and more distinctive features but also generalize better to unseen data, making them more reliable for realworld applications.
Importantly, the interpretability offered by grad-cam visualizations strengthens confidence in model predictions by providing transparent insights into the decision making process. Such explainability is essential when deploying AI systems in sensitive contexts like media verification, forensic analysis, or social media moderation.

Comparison with prior research. According to the original FaceForensics++ benchmark study [2], the xceptionNet model achieved a test accuracy of approximately 96.33% when trained and evaluated on the full dataset, which contains over 1 million images extracted from 1000 real and 4000 fake video sequences.
In this project, a custom generated static image dataset was constructed from FaceForensics++, consisting of 8000 real and 8000 fake images for training, and for testing 4000 images (2000 real and 2000 fake). Despite the smaller dataset size and computational limitations, the fine-tuned XceptionNet achieved a test accuracy of 98%, slightly surpassing the original benchmark. This result highlights the effectiveness of transfer learning when combined with a carefully designed training pipeline, even under resource-constrained conditions.
EfficientNet-B0 also performed competitively reaching a test accuracy of 96.25%, comparable to the best results reported in the literature, while offering improved computational efficiency due to its lightweight architecture.
And for reference, the custom CNN developed in this project, trained from scratch without pre-training, achieved a test precision of 79%. While this performance is lower than that of the pretrained models, it provides a valuable baseline and underscores the performance gap that can be bridged through the use of deeper, pretrained state of the art architectures.
These comparisons validate the strength of the implemented models and confirm their alignment with findings from prior research. They also demonstrate the practical feasibility of achieving high performance using transfer learning, even with smaller datasets and limited hardware.

Reflection on objectives. The project successfully fulfilled all objectives defined in Section 1.3, including the development of a custom CNN model, construction of a balanced image dataset, and fine-tuning of state-of-the-art pretrained architectures. Each model was carrefully evaluated using both quantitative metrics and interpretability tools such as Grad-CAM. The resulting comparative analysis provided actionable insights into model behavior, performance limitations, and future improvement paths. These outcomes collectively demonstrate the completion of the project's goals and its contribution to advancing deepfake detection using AI.

Future work may consider exploring ensemble methods combining different architectures to harness complementary strengths or applying temporal analysis to extend detection capabilities to full videos rather than static images alone.

5 Challenges and Limitations

Throughout the development of this project, several technical and practical challenges were encountered that shaped the workflow and progress.

5.1 Hardware Constraints

The initial plan was to train and evaluate the models locally using a MacBook Pro (14-inch, 2021) equipped with an Apple M1 Pro chip and 16 GB of unified memory, running macOS Sequoia 15.4.1. While this hardware is powerful for general computing tasks and moderate machine learning workloads, it proved insufficient for training deep neural networks, especially state-of-the-art architectures like XceptionNet and EfficientNet-B0.
Key challenges included:

- **GPU limitations:** The Apple M1 Pro's integrated GPU, while optimized for certain workloads, is not directly compatible with standard PyTorch CUDA-based GPU acceleration which is essential for efficient deep learning on large datasets.
- **Overheating and Performance throttling:** Prolonged training sessions caused the device to overheat, triggering thermal throttling that further slowed down computation and occasionally led to crashes.
- **Extended training times:** Even when limited to the custom CNN - which is a relatively lightweight model, training was considerably slow, requiring several hours per epoch, making iterative experimentation impractical.

5.2 Migration to cloud resources

To address local hardware constraints, the project transitioned to Google Colab, which offers free access to GPUs such as NVIDIA Tesla T4. This significantly accelerated training and allowed larger models to be trained within a feasible time frame. However new limitations emerged:

- **Session limits:** Free Colab sessions are typically capped at 2–4 hours of continuous GPU use, after which sessions disconnect. This occasionally interrupted long-running experiments and required careful checkpointing.
- **GPU availability:** Due to demand fluctuations, GPU resources were not always immediately available. In such cases, fallback testing was performed using CPU in Colab, which again introduced delays.

5.3 Model checkpointing and Experiment tracking

Given the session based nature and runtime limits of Google Colab, robust checkpointing and systematic experiment tracking were essential to ensure smooth progress and reproducibility.

Checkpointing. During training, model weights were saved after each epoch. The best-performing checkpoint (based on validation accuracy) was stored separately as `_best.pth`. While full training state (including optimizer state and epoch counters) was not saved, this approach ensured that the best version of each model could be reliably reloaded for evaluation or further training. Despite session-based limitations in Google Colab, this setup provided a reproducible and robust pipeline for managing model outputs.

Experiment tracking. Key performance metrics (accuracy, precision, recall, F1-score) were automatically logged into CSV files (`metrics.csv`), and visual outputs (confusion matrices, training/validation curves, Grad-CAMs) were saved systematically after each training and evaluation run. Each model type (custom CNN, XceptionNet, EfficientNet-B0) had a dedicated results folder in Google Drive to maintain organized experiment records.

Folder Structure. The overall experiment results were organized as follows:

```
/MyDrive/deepfake_detection/
    custom_cnn_results/
    XceptionNet_results/
    EfficientNet_cnn_results/
```

Each folder included:

- Model checkpoints (e.g., `custom_deepfake_cnn_best.pth`)
- Training logs and metrics (`metrics.csv`)
- Visualizations (confusion matrices, loss/accuracy plots, Grad-CAMs)

This setup ensured full traceability and reproducibility of experiments, even when session interruptions or hardware limitations occurred.

5.4 Limitations

Despite successfully completing all planned experiments, several practical limitations should be acknowledged:

- **Hardware and resource constraints:** Initially, all training was attempted locally on a MacBook Pro 2021. However, limited computing power, thermal throttling, and long training times on CPU made this approach unfeasible for deep learning tasks. As a result, the project was migrated to Google Colab, which provided GPU access but only for limited daily sessions (typically 2–4 hours). This often required fallback to CPU training when GPU resources were unavailable, slowing down the workflow.
- **Limited training epochs:** To accommodate Colab's runtime limits and mitigate long CPU-based training times, all models were trained for only 10 epochs. While this allowed for convergence and meaningful comparison between models, extended training could lead to further performance improvements, especially for the custom CNN.
- **Time constraints:** Due to the overall project timeline and resource limitations, only a fixed number of hyperparameter tuning runs and architecture refinements could be explored.

5.5 Reflection

Overall, while hardware limitations initially hindered progress, migrating to cloud resources and adopting structured checkpointing strategies allowed the project to achieve its goals. The challenges encountered highlight the importance of flexible infrastructure planning and resource management when working with computationally intensive deep learning projects.
Additionally, this process required developing new skills, particularly in working with Google Colab for cloud-based model training and enhancing familiarity with Google Drive integration for seamless data and model storage. These skills not only ensured the smooth execution of this project but also represent valuable experience for future machine learning workflows.

6 Conclusion and Future Work

This Bachelor's project has systematically explored the challenge of detecting deepfake facial images using AI-based methods. The primary objective was to develop, train, and compare convolutional neural network (CNN) architectures for binary classification of static facial images as real or fake, addressing the growing need for reliable detection systems in the face of increasingly realistic manipulations.
The work began with the development of a custom CNN model, designed to provide a baseline for performance comparison. Through iterative architectural and training refinements, the custom CNN achieved a validation accuracy of 81.19% and a test accuracy of 79%. While this confirmed that even lightweight architectures can learn meaningful representations, it also exposed limitations in detecting subtle artifacts compared to more advanced models.
To push performance further, two pretrained state-of-the-art models XceptionNet and EfficientNet-B0 were fine-tuned on the same dataset. Both outperformed the custom CNN across all evaluation metrics. XceptionNet emerged as the top performer with a test accuracy of 98%, while efficientNet-B0 followed closely with 96%, offering a compelling trade-off between performance and efficiency due to smaller parameters.
Comprehensive evaluations were conducted using performance metrics, confusion matrices, training dynamics, and Grad-CAM visualizations. These analyzes revealed that pre-trained models not only extracted more distinctive features but also directed more attention to the manipulated facial regions.
Despite these achievements, the project encountered several constraints. Hardware limitations led to a move to Google Colab, where GPU sessions were limited, restricting training to 10 epochs per model. Additionally, the use of the FaceForensics++ dataset, while reliable, introduces some bias due to its controlled nature, potentially limiting generalizability to real-world scenarios.

Future Work. Building on the solid foundation established in this project, several promising directions remain for further exploration:

- **Enhancing the custom CNN:** Although the custom model served as a strong baseline, its shallow architecture limited its ability to capture complex manipulations. Future designs could incorporate deeper structures, attention modules for example SE-blocks, or hybrid approaches blending handcrafted and learned features.
- **Extended training and fine-tuning:** Given that all models were limited to 10 epochs due to resource constraints, future work could explore longer training schedules and broader hyperparameter tuning for example learning rate decay, regularization techniques, to achieve better convergence.
- **Data augmentation and dataset expansion:** More advanced augmentation techniques such as mixup, color jittering, or random erasing, combined with datasets beyond FaceForensics++, could improve robustness to diverse deepfake generation methods.
- **Model optimization for deployment:** To support deployment in realtime or edge environments, further optimizations such as pruning, quantization, and knowledge distillation could be used to reduce model size without sacrificing performance.
- **Video-based detection:** While this project focused on static images, extending the framework to full videos would allow models to exploit temporal inconsistencies across frames, an important cue for detecting videobased deepfakes.

In conclusion, this project demonstrated that deep pretrained CNNs offer strong advantages in static image-based deepfake detection, and that even custom-built models can reach competitive baselines. The insights gained here lay the groundwork for further research toward more scalable, interpretable, and realworld-ready detection systems.

7 Appendix:

7.1 Supplementary Code

Frame Extraction Script

```python
import os
import cv2

# paths to videos
VIDEO_DIR_REAL = "faceforensics_data/original_sequences/youtube/c40/videos"
VIDEO_DIR_FAKE = "faceforensics_data/manipulated_sequences/Deepfakes/c40/videos"

# paths to save extracted images
OUTPUT_DIR_REAL = "faceforensics_data/images/real"
OUTPUT_DIR_FAKE = "faceforensics_data/images/fake"

# create folders if they do not exist
os.makedirs(OUTPUT_DIR_REAL, exist_ok=True)
os.makedirs(OUTPUT_DIR_FAKE, exist_ok=True)

def extract_frames(video_path, output_folder, num_frames=10):
    cap = cv2.VideoCapture(video_path)
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    frame_interval = max(1, total_frames // num_frames)  # select 10 evenly spaced frames

    for i in range(num_frames):
        cap.set(cv2.CAP_PROP_POS_FRAMES, i * frame_interval)  # jump to the desired frame
        ret, frame = cap.read()
        if not ret:
            break
        img_filename = os.path.join(output_folder, f"{os.path.basename(video_path).split('.')[0]}_{i}....
    jpg")
        cv2.imwrite(img_filename, frame)  # save the frame

    cap.release()

#process real videos
for video in os.listdir(VIDEO_DIR_REAL):
    video_path = os.path.join(VIDEO_DIR_REAL, video)
    extract_frames(video_path, OUTPUT_DIR_REAL, num_frames=10)

# proces deepfake videos
for video in os.listdir(VIDEO_DIR_FAKE):
    video_path = os.path.join(VIDEO_DIR_FAKE, video)
    extract_frames(video_path, OUTPUT_DIR_FAKE, num_frames=10)

print("Image extraction is finished ")
```

Listing 8: extract_frames.py

Dataset Splitting Script

```python
import os
import shutil
import random

# paths to images
DATA_DIR = "faceforensics_data/images"
TRAIN_DIR = "faceforensics_data/train"
TEST_DIR = "faceforensics_data/test"

# percentage of data for training
SPLIT_RATIO = 0.8

# create train/test folders
for folder in [TRAIN_DIR, TEST_DIR]:
    os.makedirs(os.path.join(folder, "real"), exist_ok=True)
    os.makedirs(os.path.join(folder, "fake"), exist_ok=True)

def split_data(data_dir, train_dir, test_dir, split_ratio):
    for category in ["real", "fake"]:
        images = os.listdir(os.path.join(data_dir, category))
        random.shuffle(images)  # shuffle the images
        split_idx = int(len(images) * split_ratio)

        train_images = images[:split_idx]
        test_images = images[split_idx:]

        for img in train_images:
            shutil.move(os.path.join(data_dir, category, img), os.path.join(train_dir, category, img))

        for img in test_images:
            shutil.move(os.path.join(data_dir, category, img), os.path.join(test_dir, category, img))

split_data(DATA_DIR, TRAIN_DIR, TEST_DIR, SPLIT_RATIO)
print("data was successfully split into train and test")
```

Listing 9: split_data.py

7.2 Custom CNN

Custom CNN Training Code

```python
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, random_split
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torchvision.transforms import ToTensor
from tqdm import tqdm
import matplotlib.pyplot as plt

from PIL import Image
import numpy as np
import cv2

# config
TRAIN_DIR = "/content/face_data/train"
BATCH_SIZE = 16
LEARNING_RATE = 0.001
EPOCHS = 10
IMAGE_SIZE = 299
VALIDATION_SPLIT = 0.1

MODEL_SAVE_PATH = "custom_deepfake_cnn_best.pth"
LOSS_PLOT_PATH = "custom_model_train_loss.png"
ACC_PLOT_PATH = "custom_model_val_accuracy.png"

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# custom laplacian transform
class AddLaplacian:
    def __call__(self, img):
        img_gray = img.convert("L")
        np_img = np.array(img_gray)
        lap = cv2.Laplacian(np_img, cv2.CV_64F)
        lap = (lap - lap.min()) / (lap.max() - lap.min() + 1e-8)
        lap_img = Image.fromarray(np.uint8(lap * 255))
        lap_tensor = ToTensor()(lap_img)
        img_tensor = ToTensor()(img)
        return torch.cat([img_tensor, lap_tensor], dim=0)

# combined transform
transform = transforms.Compose([
    transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
    transforms.RandomHorizontalFlip(),
    AddLaplacian(),
    transforms.Normalize(mean=[0.5]*4, std=[0.5]*4)
])

# load dataset
full_train_dataset = datasets.ImageFolder(root=TRAIN_DIR, transform=transform)
val_size = int(len(full_train_dataset) * VALIDATION_SPLIT)
train_size = len(full_train_dataset) - val_size
train_dataset, val_dataset = random_split(full_train_dataset, [train_size, val_size])
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)

# attention blocks
class ChannelAttention(nn.Module):
    def __init__(self, in_channels, reduction=16):
```

```
        super().__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.max_pool = nn.AdaptiveMaxPool2d(1)
        self.fc = nn.Sequential(
            nn.Conv2d(in_channels, in_channels // reduction, 1),
            nn.GELU(),
            nn.Conv2d(in_channels // reduction, in_channels, 1)
        )
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avg = self.fc(self.avg_pool(x))
        max = self.fc(self.max_pool(x))
        return x * self.sigmoid(avg + max)

class SpatialAttention(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(2, 1, kernel_size=7, padding=3)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avg = torch.mean(x, dim=1, keepdim=True)
        max, _ = torch.max(x, dim=1, keepdim=True)
        attn = self.sigmoid(self.conv(torch.cat([avg, max], dim=1)))
        return x * attn

class CBAM(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.ca = ChannelAttention(in_channels)
        self.sa = SpatialAttention()

    def forward(self, x):
        return self.sa(self.ca(x))

# bottleneck with CBAM
class BottleneckCBAM(nn.Module):
    def __init__(self, in_channels, mid_channels, out_channels):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, 1),
            nn.BatchNorm2d(mid_channels),
            nn.SiLU(),
            nn.Conv2d(mid_channels, mid_channels, 3, padding=1),
            nn.BatchNorm2d(mid_channels),
            nn.SiLU(),
            nn.Conv2d(mid_channels, out_channels, 1),
            nn.BatchNorm2d(out_channels),
        )
        self.shortcut = nn.Conv2d(in_channels, out_channels, 1) if in_channels != out_channels else nn....
     Identity()
        self.attn = CBAM(out_channels)
        self.act = nn.SiLU()

    def forward(self, x):
        return self.act(self.attn(self.conv(x)) + self.shortcut(x))

# main model
class DeepFakeCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.stem = nn.Sequential(
            nn.Conv2d(4, 32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.SiLU(),
```

```
            nn.MaxPool2d(2)
        )
        self.blocks = nn.Sequential(
            BottleneckCBAM(32, 32, 64), nn.MaxPool2d(2),
            BottleneckCBAM(64, 64, 128), nn.MaxPool2d(2),
            BottleneckCBAM(128, 128, 256), nn.MaxPool2d(2),
            BottleneckCBAM(256, 256, 512)
        )
        self.pool = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.AdaptiveMaxPool2d(1)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Dropout(0.5),
            nn.Linear(1024, 2)
        )

    def forward(self, x):
        x = self.stem(x)
        x = self.blocks(x)
        avg = self.pool[0](x)
        max = self.pool[1](x)
        x = torch.cat([avg, max], dim=1)
        return self.classifier(x)

# evaluation
def evaluate(model, loader):
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            outputs = model(x)
            _, preds = torch.max(outputs, 1)
            correct += (preds == y).sum().item()
            total += y.size(0)
    return 100 * correct / total

# training loop
def train(model, train_loader, val_loader, criterion, optimizer, scheduler, epochs):
    best_acc = 0.0
    train_losses = []
    val_accuracies = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0

        for images, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs}"):
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        avg_loss = running_loss / len(train_loader)
        val_acc = evaluate(model, val_loader)
        scheduler.step(avg_loss)

        train_losses.append(avg_loss)
        val_accuracies.append(val_acc)

        print(f"Epoch {epoch+1}: Loss={avg_loss:.4f} | Validation Accuracy={val_acc:.2f}%")
```

```
193
194         if val_acc > best_acc:
195             best_acc = val_acc
196             torch.save(model.state_dict(), MODEL_SAVE_PATH)
197             print(f"Best model saved to: {MODEL_SAVE_PATH}")
198
199     # save plots
200     plt.figure()
201     plt.plot(train_losses, label="Train Loss")
202     plt.xlabel("Epoch")
203     plt.ylabel("Loss")
204     plt.title("Training Loss")
205     plt.legend()
206     plt.savefig(LOSS_PLOT_PATH)
207
208     plt.figure()
209     plt.plot(val_accuracies, label="Validation Accuracy")
210     plt.xlabel("Epoch")
211     plt.ylabel("Accuracy (%)")
212     plt.title("Validation Accuracy")
213     plt.legend()
214     plt.savefig(ACC_PLOT_PATH)
215
216 # run
217 if __name__ == "__main__":
218     model = DeepFakeCNN().to(device)
219     criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
220     optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)
221     scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2, ...
        verbose=True)
222     train(model, train_loader, val_loader, criterion, optimizer, scheduler, EPOCHS)
```

Listing 10: custom_deepfake_cnn.py

Custom CNN Evaluation Code

```
1  import os
2  import torch
3  import torch.nn as nn
4  from torch.utils.data import DataLoader
5  import torchvision.datasets as datasets
6  import torchvision.transforms as transforms
7  from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, ...
       classification_report
8  import pandas as pd
9  import matplotlib.pyplot as plt
10 import seaborn as sns
11 from tqdm import tqdm
12 import numpy as np
13 from PIL import Image
14 import cv2
15
16 # paths
17 MODEL_PATH = "/content/drive/MyDrive/deepfake_detection/custom_deepfake_cnn_best.pth"
18 TEST_DIR = "/content/face_data/test"
19 SAVE_DIR = "/content/drive/MyDrive/deepfake_detection"
20 os.makedirs(SAVE_DIR, exist_ok=True)
21
22 # device
23 device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
24 print("Using device:", device)
25
26 # laplacian transform
27 class AddLaplacian:
```

```
    def __call__(self, img):
        img_gray = img.convert("L")
        np_img = np.array(img_gray)
        lap = cv2.Laplacian(np_img, cv2.CV_64F)
        lap = (lap - lap.min()) / (lap.max() - lap.min() + 1e-8)
        lap_img = Image.fromarray(np.uint8(lap * 255))
        lap_tensor = transforms.ToTensor()(lap_img)
        img_tensor = transforms.ToTensor()(img)
        return torch.cat([img_tensor, lap_tensor], dim=0)

# transforms
transform = transforms.Compose([
    transforms.Resize((299, 299)),
    AddLaplacian(),
    transforms.Normalize(mean=[0.5]*4, std=[0.5]*4)
])

# dataset
test_dataset = datasets.ImageFolder(root=TEST_DIR, transform=transform)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)
idx_to_class = {v: k for k, v in test_dataset.class_to_idx.items()}

# CBAM components
class ChannelAttention(nn.Module):
    def __init__(self, in_channels, reduction=16):
        super().__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.max_pool = nn.AdaptiveMaxPool2d(1)
        self.fc = nn.Sequential(
            nn.Conv2d(in_channels, in_channels // reduction, 1),
            nn.GELU(),
            nn.Conv2d(in_channels // reduction, in_channels, 1)
        )
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avg = self.fc(self.avg_pool(x))
        max = self.fc(self.max_pool(x))
        return x * self.sigmoid(avg + max)

class SpatialAttention(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv = nn.Conv2d(2, 1, kernel_size=7, padding=3)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avg = torch.mean(x, dim=1, keepdim=True)
        max, _ = torch.max(x, dim=1, keepdim=True)
        return x * self.sigmoid(self.conv(torch.cat([avg, max], dim=1)))

class CBAM(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.ca = ChannelAttention(in_channels)
        self.sa = SpatialAttention()

    def forward(self, x):
        return self.sa(self.ca(x))

class BottleneckCBAM(nn.Module):
    def __init__(self, in_channels, mid_channels, out_channels):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_channels, mid_channels, 1),
            nn.BatchNorm2d(mid_channels),
```

```
            nn.SiLU(),
            nn.Conv2d(mid_channels, mid_channels, 3, padding=1),
            nn.BatchNorm2d(mid_channels),
            nn.SiLU(),
            nn.Conv2d(mid_channels, out_channels, 1),
            nn.BatchNorm2d(out_channels)
        )
        self.shortcut = nn.Conv2d(in_channels, out_channels, 1) if in_channels != out_channels else nn.Identity()
        self.attn = CBAM(out_channels)
        self.act = nn.SiLU()

    def forward(self, x):
        return self.act(self.attn(self.conv(x)) + self.shortcut(x))

# full model
class DeepFakeCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.stem = nn.Sequential(
            nn.Conv2d(4, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.SiLU(),
            nn.MaxPool2d(2)
        )
        self.blocks = nn.Sequential(
            BottleneckCBAM(32, 32, 64), nn.MaxPool2d(2),
            BottleneckCBAM(64, 64, 128), nn.MaxPool2d(2),
            BottleneckCBAM(128, 128, 256), nn.MaxPool2d(2),
            BottleneckCBAM(256, 256, 512)
        )
        self.pool = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.AdaptiveMaxPool2d(1)
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Dropout(0.5),
            nn.Linear(1024, 2)
        )

    def forward(self, x):
        x = self.stem(x)
        x = self.blocks(x)
        avg = self.pool[0](x)
        max = self.pool[1](x)
        x = torch.cat([avg, max], dim=1)
        return self.classifier(x)

# grad-CAM functions
def generate_gradcam(model, input_tensor, target_class, target_layer, device):
    gradients = []
    activations = []

    def backward_hook(module, grad_input, grad_output):
        gradients.append(grad_output[0])

    def forward_hook(module, input, output):
        activations.append(output)

    forward_handle = target_layer.register_forward_hook(forward_hook)
    backward_handle = target_layer.register_backward_hook(backward_hook)

    model.eval()
    input_tensor = input_tensor.unsqueeze(0).to(device)
    input_tensor.requires_grad_()
```

```
    output = model(input_tensor)
    model.zero_grad()
    target = output[0, target_class]
    target.backward()

    grads_val = gradients[0][0].cpu().data.numpy()
    activations_val = activations[0][0].cpu().data.numpy()
    weights = np.mean(grads_val, axis=(1, 2))
    cam = np.zeros(activations_val.shape[1:], dtype=np.float32)

    for i, w in enumerate(weights):
        cam += w * activations_val[i]

    cam = np.maximum(cam, 0)
    cam = cv2.resize(cam, (299, 299))
    cam -= np.min(cam)
    cam /= np.max(cam)

    forward_handle.remove()
    backward_handle.remove()

    return cam

def save_cam_overlay(original_img_tensor, cam, save_path):
    img = original_img_tensor[:3].cpu().numpy().transpose(1, 2, 0)
    img = (img * 0.5 + 0.5) * 255
    img = np.uint8(np.clip(img, 0, 255))
    heatmap = cv2.applyColorMap(np.uint8(255 * cam), cv2.COLORMAP_JET)
    overlay = np.float32(heatmap) * 0.5 + np.float32(img)
    overlay = np.uint8(np.clip(overlay, 0, 255))
    cv2.imwrite(save_path, overlay[:, :, ::-1])

# load model
model = DeepFakeCNN().to(device)
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
model.eval()

# evaluate
y_true, y_pred = [], []
correct_samples, wrong_samples = [], []

for inputs, labels in tqdm(test_loader):
    inputs, labels = inputs.to(device), labels.to(device)
    outputs = model(inputs)
    preds = torch.argmax(outputs, 1)
    y_true.extend(labels.cpu().numpy())
    y_pred.extend(preds.cpu().numpy())

    for i in range(len(labels)):
        img = inputs[i][:3].cpu()
        label = labels[i].item()
        pred = preds[i].item()

        if label == pred and len(correct_samples) < 10:
            correct_samples.append((img, label, pred))
            cam = generate_gradcam(model, inputs[i].detach().clone(), pred, model.blocks[-1], device)
            save_cam_overlay(inputs[i][:3], cam, os.path.join(SAVE_DIR, f"gradcam_correct_{i}_true_{...
    idx_to_class[label]}_pred_{idx_to_class[pred]}.png"))

        elif label != pred and len(wrong_samples) < 10:
            wrong_samples.append((img, label, pred))
            cam = generate_gradcam(model, inputs[i].detach().clone(), pred, model.blocks[-1], device)
            save_cam_overlay(inputs[i][:3], cam, os.path.join(SAVE_DIR, f"gradcam_wrong_{i+100}_true_{...
    idx_to_class[label]}_pred_{idx_to_class[pred]}.png"))

# metrics
```

```
acc = accuracy_score(y_true, y_pred)
prec = precision_score(y_true, y_pred)
rec = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

print("\nClassification Report:\n")
print(classification_report(y_true, y_pred, target_names=list(idx_to_class.values())))

# save metrics
df = pd.DataFrame([{
    "Accuracy": acc,
    "Precision": prec,
    "Recall": rec,
    "F1-score": f1
}])
df.to_csv(os.path.join(SAVE_DIR, "metrics.csv"), index=False)

# confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", xticklabels=idx_to_class.values(), yticklabels=idx_to_class.values...
    (), cmap="Blues")
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.savefig(os.path.join(SAVE_DIR, "confusion_matrix.png"))

print(f"\nEvaluation complete. All results saved to: {SAVE_DIR}")
```

Listing 11: evaluate_custom_cnn.py

7.3 XceptionNet CNN

XceptionNet CNN Training Code

```python
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, random_split
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from tqdm import tqdm
import matplotlib.pyplot as plt
import timm

# config
TRAIN_DIR = "/content/face_data/train"
SAVE_DIR = "/content/drive/MyDrive/deepfake_detection"
os.makedirs(SAVE_DIR, exist_ok=True)

BATCH_SIZE = 16
LEARNING_RATE = 3e-5  # more stable for pretrained models
EPOCHS = 10
IMAGE_SIZE = 299
VALIDATION_SPLIT = 0.1

MODEL_SAVE_PATH = os.path.join(SAVE_DIR, "xception_best.pth")
LOSS_PLOT_PATH = os.path.join(SAVE_DIR, "xception_train_loss.png")
ACC_PLOT_PATH = os.path.join(SAVE_DIR, "xception_val_accuracy.png")

# device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# transform RGB
transform = transforms.Compose([
    transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5]*3, std=[0.5]*3)
])

# dataset
full_train_dataset = datasets.ImageFolder(root=TRAIN_DIR, transform=transform)
val_size = int(len(full_train_dataset) * VALIDATION_SPLIT)
train_size = len(full_train_dataset) - val_size
train_dataset, val_dataset = random_split(full_train_dataset, [train_size, val_size])
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)

# xception model from timm
model = timm.create_model('xception', pretrained=True)
model.reset_classifier(2)  # Set output for 2 classes
model = model.to(device)

# unfreeze all layers
for param in model.parameters():
    param.requires_grad = True

# evaluation function
def evaluate(model, loader):
    model.eval()
    correct, total = 0, 0
```

```
    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            outputs = model(x)
            _, preds = torch.max(outputs, 1)
            correct += (preds == y).sum().item()
            total += y.size(0)
    return 100 * correct / total

# training loop
def train(model, train_loader, val_loader, criterion, optimizer, scheduler, epochs):
    best_acc = 0.0
    train_losses = []
    val_accuracies = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0

        for images, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs}"):
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        avg_loss = running_loss / len(train_loader)
        val_acc = evaluate(model, val_loader)
        scheduler.step(avg_loss)

        train_losses.append(avg_loss)
        val_accuracies.append(val_acc)

        print(f"Epoch {epoch+1}: Loss={avg_loss:.4f} | Validation Accuracy={val_acc:.2f}%")

        if val_acc > best_acc:
            best_acc = val_acc
            torch.save(model.state_dict(), MODEL_SAVE_PATH)
            print(f"Best model saved to: {MODEL_SAVE_PATH}")

    # save loss plot
    plt.figure()
    plt.plot(train_losses, label="Train Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training Loss")
    plt.legend()
    plt.savefig(LOSS_PLOT_PATH)

    # save accuracy plot
    plt.figure()
    plt.plot(val_accuracies, label="Validation Accuracy")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy (%)")
    plt.title("Validation Accuracy")
    plt.legend()
    plt.savefig(ACC_PLOT_PATH)

# run training
if __name__ == "__main__":
    criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
    optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2, ...
     verbose=True)
```

```
    train(model, train_loader, val_loader, criterion, optimizer, scheduler, EPOCHS)
```

Listing 12: xception_train.py

XceptionNet CNN Evaluation Code

```
import os
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, ...
    classification_report
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
import numpy as np
from PIL import Image
import cv2
import timm

# paths
MODEL_PATH = "/content/drive/MyDrive/deepfake_detection/xception_best.pth"
TEST_DIR = "/content/face_data/test"
SAVE_DIR = "/content/drive/MyDrive/deepfake_detection"
os.makedirs(SAVE_DIR, exist_ok=True)

# device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# transforms RGB
transform = transforms.Compose([
    transforms.Resize((299, 299)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5]*3, std=[0.5]*3)
])

# dataset
test_dataset = datasets.ImageFolder(root=TEST_DIR, transform=transform)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)
idx_to_class = {v: k for k, v in test_dataset.class_to_idx.items()}

# load model
model = timm.create_model('xception', pretrained=False)
model.reset_classifier(2)
model = model.to(device)
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
model.eval()

# grad-CAM functions
def generate_gradcam(model, input_tensor, target_class, target_layer, device):
    gradients = []
    activations = []

    def backward_hook(module, grad_input, grad_output):
        gradients.append(grad_output[0])

    def forward_hook(module, input, output):
        activations.append(output)

    forward_handle = target_layer.register_forward_hook(forward_hook)
```

```
    backward_handle = target_layer.register_backward_hook(backward_hook)

    model.eval()
    input_tensor = input_tensor.unsqueeze(0).to(device)
    input_tensor.requires_grad_()
    output = model(input_tensor)
    model.zero_grad()
    target = output[0, target_class]
    target.backward()

    grads_val = gradients[0][0].cpu().data.numpy()
    activations_val = activations[0][0].cpu().data.numpy()
    weights = np.mean(grads_val, axis=(1, 2))
    cam = np.zeros(activations_val.shape[1:], dtype=np.float32)

    for i, w in enumerate(weights):
        cam += w * activations_val[i]

    cam = np.maximum(cam, 0)
    cam = cv2.resize(cam, (299, 299))
    cam -= np.min(cam)
    cam /= np.max(cam)

    forward_handle.remove()
    backward_handle.remove()

    return cam

def save_cam_overlay(original_img_tensor, cam, save_path):
    img = original_img_tensor[:3].cpu().numpy().transpose(1, 2, 0)
    img = (img * 0.5 + 0.5) * 255
    img = np.uint8(np.clip(img, 0, 255))
    heatmap = cv2.applyColorMap(np.uint8(255 * cam), cv2.COLORMAP_JET)
    overlay = np.float32(heatmap) * 0.5 + np.float32(img)
    overlay = np.uint8(np.clip(overlay, 0, 255))
    cv2.imwrite(save_path, overlay[:, :, ::-1])

# evaluate
y_true, y_pred = [], []
correct_samples, wrong_samples = [], []

for inputs, labels in tqdm(test_loader):
    inputs, labels = inputs.to(device), labels.to(device)
    outputs = model(inputs)
    preds = torch.argmax(outputs, 1)
    y_true.extend(labels.cpu().numpy())
    y_pred.extend(preds.cpu().numpy())

    for i in range(len(labels)):
        img = inputs[i][:3].cpu()
        label = labels[i].item()
        pred = preds[i].item()

        if label == pred and len(correct_samples) < 10:
            correct_samples.append((img, label, pred))
            cam = generate_gradcam(model, inputs[i].detach().clone(), pred, model.conv1, device)
            save_cam_overlay(inputs[i][:3], cam, os.path.join(SAVE_DIR, f"gradcam_correct_{i}_true_{...
    idx_to_class[label]}_pred_{idx_to_class[pred]}.png"))

        elif label != pred and len(wrong_samples) < 10:
            wrong_samples.append((img, label, pred))
            cam = generate_gradcam(model, inputs[i].detach().clone(), pred, model.conv1, device)
            save_cam_overlay(inputs[i][:3], cam, os.path.join(SAVE_DIR, f"gradcam_wrong_{i+100}_true_{...
    idx_to_class[label]}_pred_{idx_to_class[pred]}.png"))

# metrics
```

```
acc = accuracy_score(y_true, y_pred)
prec = precision_score(y_true, y_pred)
rec = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

print("\nClassification Report:\n")
print(classification_report(y_true, y_pred, target_names=list(idx_to_class.values())))

# save metrics
df = pd.DataFrame([{
    "Accuracy": acc,
    "Precision": prec,
    "Recall": rec,
    "F1-score": f1
}])
df.to_csv(os.path.join(SAVE_DIR, "metrics_xception.csv"), index=False)

# confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", xticklabels=idx_to_class.values(), yticklabels=idx_to_class.values...
    (), cmap="Blues")
plt.title("Confusion Matrix - Xception")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.savefig(os.path.join(SAVE_DIR, "confusion_matrix_xception.png"))

print(f"\n Evaluation complete. All results saved to: {SAVE_DIR}")
```

Listing 13: evaluate_xception.py

7.4 EfficientNet-B0 CNN

EfficientNet-B0 CNN Training Code

```
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, random_split
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from tqdm import tqdm
import matplotlib.pyplot as plt
import timm

# config
TRAIN_DIR = "/content/face_data/train"
SAVE_DIR = "/content/drive/MyDrive/deepfake_detection"
os.makedirs(SAVE_DIR, exist_ok=True)

BATCH_SIZE = 16
LEARNING_RATE = 3e-5
EPOCHS = 10
IMAGE_SIZE = 224  # efficientNet-B0 default input size
VALIDATION_SPLIT = 0.1

MODEL_SAVE_PATH = os.path.join(SAVE_DIR, "efficientnet_best.pth")
LOSS_PLOT_PATH = os.path.join(SAVE_DIR, "efficientnet_train_loss.png")
ACC_PLOT_PATH = os.path.join(SAVE_DIR, "efficientnet_val_accuracy.png")

# device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# transform RGB
transform = transforms.Compose([
    transforms.Resize((IMAGE_SIZE, IMAGE_SIZE)),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5]*3, std=[0.5]*3)
])

# dataset
full_train_dataset = datasets.ImageFolder(root=TRAIN_DIR, transform=transform)
val_size = int(len(full_train_dataset) * VALIDATION_SPLIT)
train_size = len(full_train_dataset) - val_size
train_dataset, val_dataset = random_split(full_train_dataset, [train_size, val_size])
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)

# efficientnet model from timm
model = timm.create_model('efficientnet_b0', pretrained=True)
model.reset_classifier(2)   # set output for 2 classes
model = model.to(device)

# unfreeze all layers
for param in model.parameters():
    param.requires_grad = True

# evaluation function
def evaluate(model, loader):
    model.eval()
    correct, total = 0, 0
```

```
    with torch.no_grad():
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            outputs = model(x)
            _, preds = torch.max(outputs, 1)
            correct += (preds == y).sum().item()
            total += y.size(0)
    return 100 * correct / total

# training loop
def train(model, train_loader, val_loader, criterion, optimizer, scheduler, epochs):
    best_acc = 0.0
    train_losses = []
    val_accuracies = []

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0

        for images, labels in tqdm(train_loader, desc=f"Epoch {epoch+1}/{epochs}"):
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()

        avg_loss = running_loss / len(train_loader)
        val_acc = evaluate(model, val_loader)
        scheduler.step(avg_loss)

        train_losses.append(avg_loss)
        val_accuracies.append(val_acc)

        print(f"Epoch {epoch+1}: Loss={avg_loss:.4f} | Validation Accuracy={val_acc:.2f}%")

        if val_acc > best_acc:
            best_acc = val_acc
            torch.save(model.state_dict(), MODEL_SAVE_PATH)
            print(f"Best model saved to: {MODEL_SAVE_PATH}")

    # save loss plot
    plt.figure()
    plt.plot(train_losses, label="Train Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training Loss")
    plt.legend()
    plt.savefig(LOSS_PLOT_PATH)

    # save accuracy plot
    plt.figure()
    plt.plot(val_accuracies, label="Validation Accuracy")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy (%)")
    plt.title("Validation Accuracy")
    plt.legend()
    plt.savefig(ACC_PLOT_PATH)

# run training
if __name__ == "__main__":
    criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
    optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)
    scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2, ...
     verbose=True)
```

```
    train(model, train_loader, val_loader, criterion, optimizer, scheduler, EPOCHS)
```

Listing 14: efficientnet_train.py

EfficientNet-B0 CNN Evaluation Code

```
import os
import torch
import torch.nn as nn
from torch.utils.data import DataLoader
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, ...
    classification_report
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm
import numpy as np
from PIL import Image
import cv2
import timm

# paths
MODEL_PATH = "/content/drive/MyDrive/deepfake_detection/efficientnet_best.pth"
TEST_DIR = "/content/face_data/test"
SAVE_DIR = "/content/drive/MyDrive/deepfake_detection"
os.makedirs(SAVE_DIR, exist_ok=True)

# device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# transforms RGB
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5]*3, std=[0.5]*3)
])

# dataset
test_dataset = datasets.ImageFolder(root=TEST_DIR, transform=transform)
test_loader = DataLoader(test_dataset, batch_size=16, shuffle=False)
idx_to_class = {v: k for k, v in test_dataset.class_to_idx.items()}

# load model
model = timm.create_model('efficientnet_b0', pretrained=False)
model.reset_classifier(2)
model = model.to(device)
model.load_state_dict(torch.load(MODEL_PATH, map_location=device))
model.eval()

# grad-CAM functions
def generate_gradcam(model, input_tensor, target_class, target_layer, device):
    gradients = []
    activations = []

    def backward_hook(module, grad_input, grad_output):
        gradients.append(grad_output[0])

    def forward_hook(module, input, output):
        activations.append(output)

    forward_handle = target_layer.register_forward_hook(forward_hook)
```

```
    backward_handle = target_layer.register_backward_hook(backward_hook)

    model.eval()
    input_tensor = input_tensor.unsqueeze(0).to(device)
    input_tensor.requires_grad_()
    output = model(input_tensor)
    model.zero_grad()
    target = output[0, target_class]
    target.backward()

    grads_val = gradients[0][0].cpu().data.numpy()
    activations_val = activations[0][0].cpu().data.numpy()
    weights = np.mean(grads_val, axis=(1, 2))
    cam = np.zeros(activations_val.shape[1:], dtype=np.float32)

    for i, w in enumerate(weights):
        cam += w * activations_val[i]

    cam = np.maximum(cam, 0)
    cam = cv2.resize(cam, (224, 224))
    cam -= np.min(cam)
    cam /= np.max(cam)

    forward_handle.remove()
    backward_handle.remove()

    return cam

def save_cam_overlay(original_img_tensor, cam, save_path):
    img = original_img_tensor[:3].cpu().numpy().transpose(1, 2, 0)
    img = (img * 0.5 + 0.5) * 255
    img = np.uint8(np.clip(img, 0, 255))
    heatmap = cv2.applyColorMap(np.uint8(255 * cam), cv2.COLORMAP_JET)
    overlay = np.float32(heatmap) * 0.5 + np.float32(img)
    overlay = np.uint8(np.clip(overlay, 0, 255))
    cv2.imwrite(save_path, overlay[:, :, ::-1])

# evaluate
y_true, y_pred = [], []
correct_samples, wrong_samples = [], []

for inputs, labels in tqdm(test_loader):
    inputs, labels = inputs.to(device), labels.to(device)
    outputs = model(inputs)
    preds = torch.argmax(outputs, 1)
    y_true.extend(labels.cpu().numpy())
    y_pred.extend(preds.cpu().numpy())

    for i in range(len(labels)):
        img = inputs[i][:3].cpu()
        label = labels[i].item()
        pred = preds[i].item()

        if label == pred and len(correct_samples) < 10:
            correct_samples.append((img, label, pred))
            cam = generate_gradcam(model, inputs[i].detach().clone(), pred, model.conv_stem, device)
            save_cam_overlay(inputs[i][:3], cam, os.path.join(SAVE_DIR, f"gradcam_correct_{i}_true_{...
    idx_to_class[label]}_pred_{idx_to_class[pred]}.png"))

        elif label != pred and len(wrong_samples) < 10:
            wrong_samples.append((img, label, pred))
            cam = generate_gradcam(model, inputs[i].detach().clone(), pred, model.conv_stem, device)
            save_cam_overlay(inputs[i][:3], cam, os.path.join(SAVE_DIR, f"gradcam_wrong_{i+100}_true_{...
    idx_to_class[label]}_pred_{idx_to_class[pred]}.png"))

# metrics
```

```
acc = accuracy_score(y_true, y_pred)
prec = precision_score(y_true, y_pred)
rec = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

print("\nClassification Report:\n")
print(classification_report(y_true, y_pred, target_names=list(idx_to_class.values())))

# save metrics
df = pd.DataFrame([{
    "Accuracy": acc,
    "Precision": prec,
    "Recall": rec,
    "F1-score": f1
}])
df.to_csv(os.path.join(SAVE_DIR, "metrics_efficientnet.csv"), index=False)

# confusion matrix
cm = confusion_matrix(y_true, y_pred)
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", xticklabels=idx_to_class.values(), yticklabels=idx_to_class.values...
    (), cmap="Blues")
plt.title("Confusion Matrix - EfficientNet")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.savefig(os.path.join(SAVE_DIR, "confusion_matrix_efficientnet.png"))

print(f"\n Evaluation complete. All results saved to: {SAVE_DIR}")
```

Listing 15: evaluate_efficientnet.py

7.5 Best .pth models

The best-performing models obtained during training was saved as a PyTorch checkpoint files:

- `custom_deepfake_cnn_best.pth`
- `xception_best.pth`
- `efficientnet_best.pth`

This files contain all the learned model parameters and can be loaded using the evaluation script to reproduce the results presented in the report.

References

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, *et al.*, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[2] A. Rössler, D. Cozzolino, L. Verdoliva, C. Riess, J. Thies, and M. Nießner, "Faceforensics++: Learning to detect manipulated facial images," *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 1–11, 2019.

[3] J. Thies, M. Zollhöfer, M. Stamminger, C. Theobalt, and M. Nießner, "Face2face: Real-time face capture and reenactment of rgb videos," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2387–2395.

[4] K. Prajwal, R. Mukhopadhyay, V. P. Namboodiri, and C. Jawahar, "Wav2lip: Accurately synthesizing lip movements from speech," in *European Conference on Computer Vision*, Springer, 2020, pp. 251–269.

[5] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1251–1258, 2017.

[6] D. Afchar, V. Nozick, J. Yamagishi, and I. Echizen, "Mesonet: A compact facial video forgery detection network," *2018 IEEE International Workshop on Information Forensics and Security (WIFS)*, pp. 1–7, 2018.

[7] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *International conference on machine learning*, pp. 6105–6114, 2019.

[8] B. Dolhansky, R. Bitton, B. Pflaum, *et al.*, *Deepfake detection challenge dataset*, 2020.

[9] Y. Li, M.-C. Chang, and S. Lyu, "Celeb-df: A large-scale challenging dataset for deepfake forensics," *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3207–3216, 2020.