Project Report
GAN Application for
Fake Image Detection and Generation

Group Member:

Le Nguyen Phuoc Thanh
Nguyen Khac Viet Anh
Nguyen Cong Son
Hoang Van Nhan
Tran Quang Trong

Contents

1 Introduction

This project develops a Flutter-based mobile application integrating Deep Convolutional Generative Adversarial Networks (DCGAN) and Progressive Growing GAN (PGAN) to address the growing challenge of fake images while enabling synthetic image generation. Fake images, such as deepfakes, pose significant risks like fraud and misinformation, necessitating robust detection methods to ensure digital content authenticity. Simultaneously, generating realistic images offers potential for research and future applications, such as graphic design through text-based synthesis. The application targets general users, allowing them to detect fake images and generate random human faces. PGAN, deployed on a server, serves as the primary model for high-quality output (512-pixel resolution), while DCGAN operates on-device (64-pixel resolution) for offline scenarios, using CelebA and CelebA-HQ datasets for training. Key challenges included dataset selection, model reimplementation, and server integration, which were addressed using base64 encoding via a RESTful API. This project demonstrates a practical approach to combating fake images and exploring AI-driven image generation.

2 Methods Used to Solve the Problem

2.1 General Approach

Generative Adversarial Networks (GANs) consist of two neural networks: a generator (G) that produces synthetic images from random latent vectors, and a discriminator (D) that distinguishes real images from generated ones. The two networks are trained adversarially, with G improving its output to deceive D, and D enhancing its ability to differentiate real and fake images.

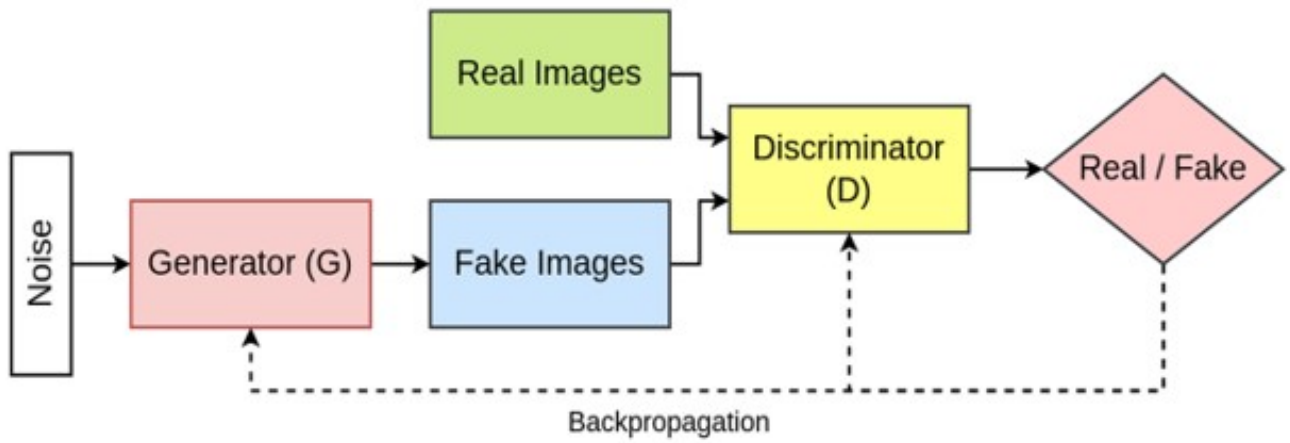

Figure 1: Pipeline of GANs

In this project, GANs serve dual purposes: the discriminator detects fake images by outputting a probability of authenticity, while the generator creates random human faces. Two GAN variants are employed: DCGAN for lightweight on-device operation and PGAN for high-quality server-based processing, addressing the challenges of offline accessibility and high-resolution generation.

2.2 DCGAN

2.2.1 Architecture

The Deep Convolutional GAN (DCGAN) follows the architecture proposed by Radford et al. [1], adapted for on-device deployment at 64×64 resolution. The generator transforms a latent vector of size 100 (nz) into an RGB image through five layers of transposed convolutions. It starts with a 1×1 feature map, progressively upsampling to 4×4, 8×8, 16×16, 32×32, and finally 64×64, with feature maps decreasing from 512 (ngf×8) to 3 (nc) channels. Each layer, except the last, uses BatchNorm2d and ReLU activation, with the final layer applying Tanh to produce outputs in [-1, 1].

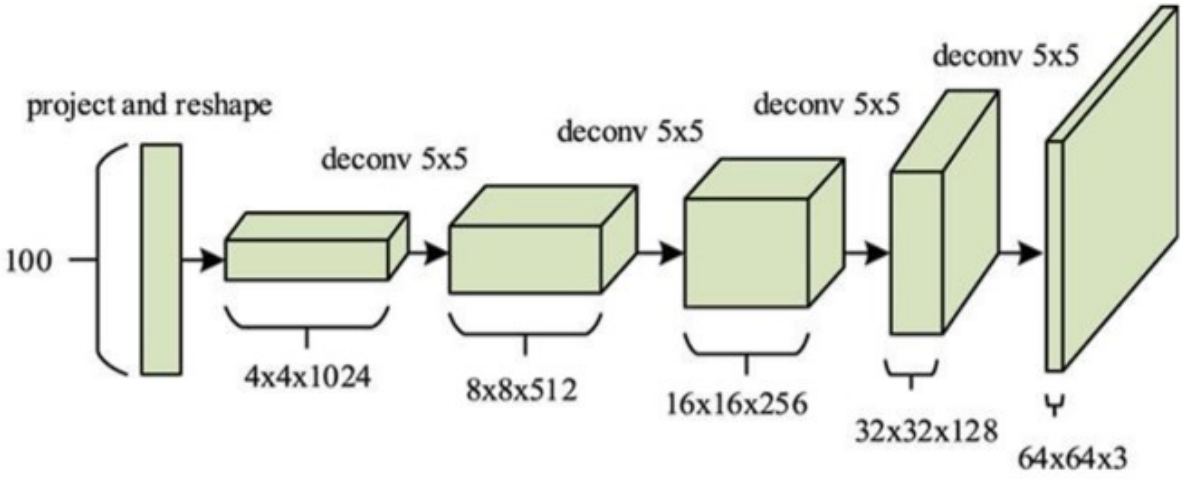

Figure 2: Generator Architecture

The discriminator mirrors this structure, taking a 3×64×64 RGB image and downsampling through five convolutional layers to a single scalar output. Feature maps increase from 64 (ndf) to 512 (ndf×8), using LeakyReLU (0.2) and BatchNorm2d (except in the first layer), with a Sigmoid output.

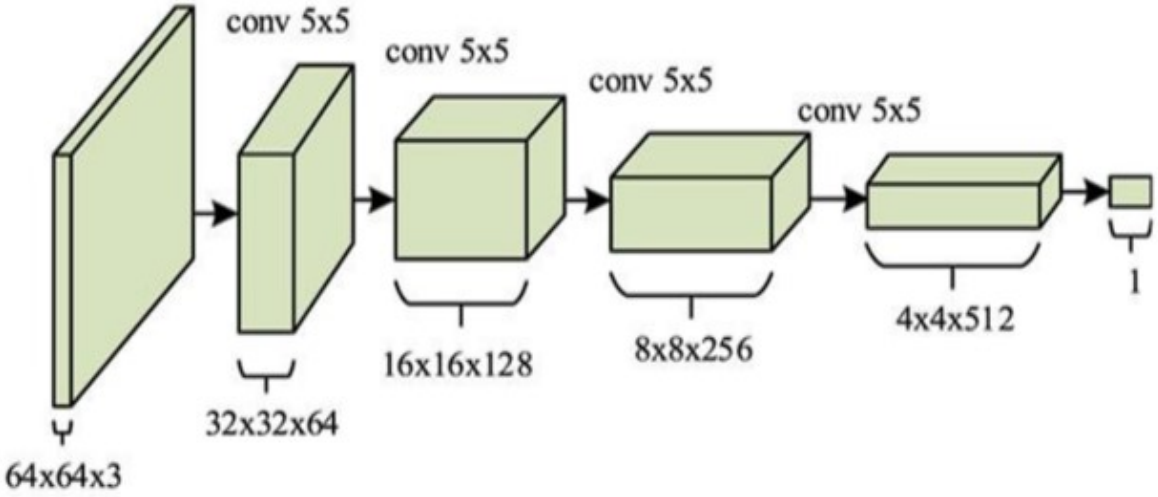

Figure 3: Discriminator Architecture

2.2.2 Training Process

DCGAN was trained on the CelebA dataset, preprocessed by resizing images to 64×64, center cropping, and normalizing to [-1, 1] with mean and standard deviation of 0.5 per channel. The training uses binary cross-entropy loss, defined as:

$$L_D = -\mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] - \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]$$

$$L_G = -\mathbb{E}_{z\sim p_z}[\log D(G(z))]$$

where $D(x)$ is the discriminator's output for real images x, and $G(z)$ is the generator's output for latent vectors z. The Adam optimizer was used with a learning rate of 0.0002, $\beta_1 = 0.5$, and $\beta_2 = 0.999$, over 250 epochs with a batch size of 4096. Training stability was ensured through BatchNorm2d, LeakyReLU (0.2) in the discriminator, a fixed random seed (999), and separate optimization steps for G and D. The training loop alternates between updating D with real and fake images and updating G to maximize D's error.

Training took approximately 8 hours and 30 minutes on RTX4060 GPU system with 20 worker processes for data loading. For on-device deployment, the model was converted to ONNX format and optimized using ONNX Runtime, ensuring efficient inference on resource-constrained devices.

2.3 PGAN

2.3.1 Architecture

The Progressive Growing GAN (PGAN) follows the methodology proposed by Karras et al. [2], employing a set of core techniques to enhance the quality, stability, and variation of generated images. The architecture is designed to progressively increase resolution from 4×4 to 512×512, with the generator and discriminator growing in synchrony. The following key techniques are integral to PGGAN's performance:

- **Progressive Growing**: Training begins at a low resolution of 4×4 pixels, with new convolutional layers incrementally added to both the generator and discriminator as training progresses. This allows the model to first learn large-scale structures before refining finer details, stabilizing training and accelerating convergence.
- **Minibatch Standard Deviation**: To encourage diversity in generated images, the discriminator includes a minibatch standard deviation layer. This layer computes the standard deviation of each feature across the minibatch at each spatial location, averages these values into a single scalar, and concatenates it as an additional feature map. This mechanism, detailed in Section 3 of [2], ensures the discriminator accounts for variation within the minibatch.
- **Equalized Learning Rate**: Instead of relying on careful weight initialization, PGGAN uses a simple $\mathcal{N}(0, 1)$ initialization and scales weights dynamically at runtime using a normalization constant derived from He's initializer (Section 4.1 of [2]). Implemented via custom layers like `EqualizedLR_Conv2d`, this ensures all layers learn at a consistent rate, balancing the training process.
- **Pixelwise Feature Vector Normalization**: In the generator, feature vectors at each pixel are normalized to unit length after every convolutional layer. This prevents signal magnitudes from escalating due to competition between the generator and discriminator, maintaining stability as described in Section 4.2 of [2].

- **Smooth Fading of New Layers**: During resolution transitions, new layers are faded in gradually using a blending parameter α that increases linearly from 0 to 1. The output is computed as:

$$\text{Output} = (1 - \alpha) \cdot \text{Low-res output} + \alpha \cdot \text{High-res output}$$

This smooth integration avoids disrupting the already trained lower-resolution layers.

The generator and discriminator are mirrored architectures. Each generator block consists of convolutional layers, upsampling, and LeakyReLU (0.2) activations, followed by pixelwise normalization. The discriminator mirrors this with convolutional layers, downsampling, LeakyReLU (0.2) activations, and the minibatch standard deviation layer. Feature map sizes decrease from 512 channels at lower resolutions to 256 channels at 512×512.

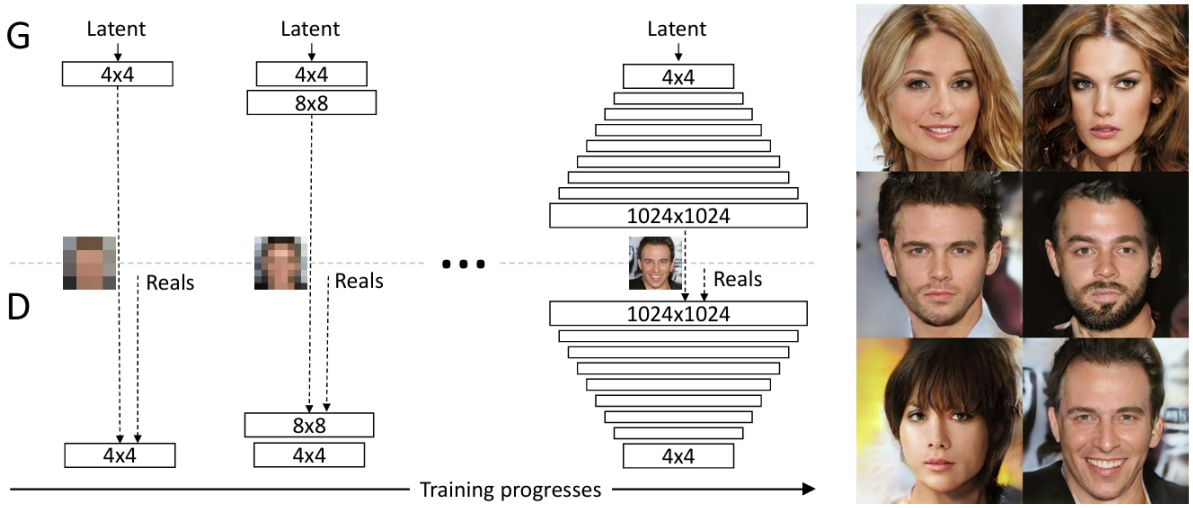

Figure 4: Architecture of the Progressive Growing GAN, illustrating the synchronized growth of generator and discriminator layers.

2.3.2 Training Process

The training of the Progressive Growing GAN (PGAN) is conducted in a progressive manner, starting from a low resolution of 4×4 pixels and incrementally increasing to the target resolution of 512×512 pixels.

Loss Function: The primary loss function employed in PGAN is the Wasserstein GAN with Gradient Penalty (WGAN-GP), as introduced by Gulrajani et al. [3]. The loss functions for the discriminator and generator are defined as:

$$L_D = \mathbb{E}_{x \sim p_{\text{data}}}[D(x)] - \mathbb{E}_{z \sim p_z}[D(G(z))] + \lambda \mathbb{E}_{\hat{x} \sim p_{\hat{x}}} \left[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2 \right]$$

$$L_G = -\mathbb{E}_{z \sim p_z}[D(G(z))]$$

Here, D represents the discriminator, G the generator, p_{data} the real data distribution, p_z the latent space distribution (typically $\mathcal{N}(0, 1)$ with 512 components), and $p_{\hat{x}}$ the distribution of interpolated samples between real and generated images. The gradient

penalty coefficient λ is set to 10, ensuring the discriminator's gradients remain well-behaved, which is particularly important as resolution increases and the complexity of distinguishing real from generated images grows.

Optimizer and Hyperparameters: The Adam optimizer is employed for both the generator and discriminator, with a learning rate of 1×10^{-4}, $\beta_1 = 0$, and $\beta_2 = 0.99$.

Training Schedule: The training advances through multiple resolution stages, with batch sizes adjusted to optimize computational efficiency and memory usage: 4×4 (batch size 1024), 8×8 (128), 16×16 and 32×32 (64), 64×64 and 128×128 (16), and 256×256 and 512×512 (4). Each stage consists of a fading phase, where new layers are gradually introduced, followed by a stabilization phase at the new resolution. The total training spans approximately 100 epochs across all stages, completed in about three days on an A100 GPU system.

Additional Details: The CelebA-HQ dataset, used for training, is normalized to the range $[-1, 1]$ with a per-channel mean and standard deviation of 0.5.

3 System Deployment and Model Integration

This section details the deployment of the server and the integration of the on-device model:

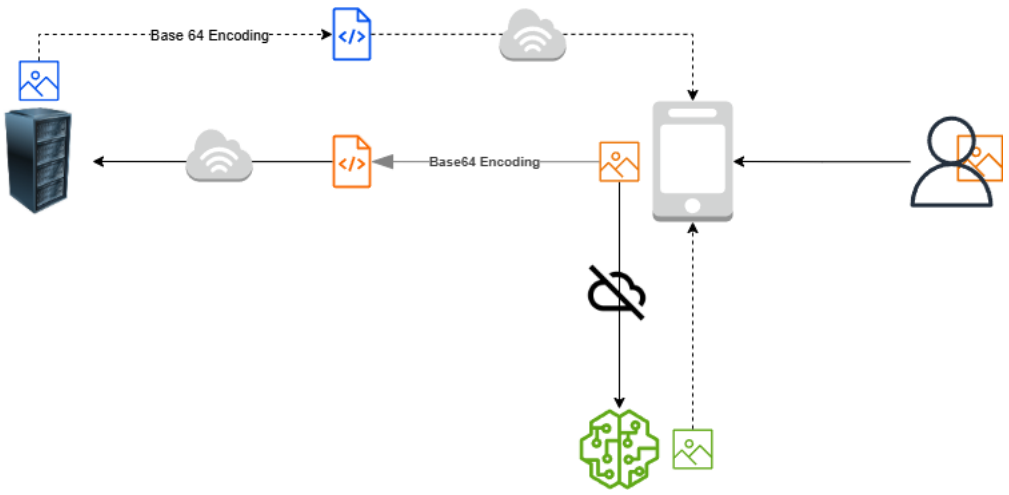

Figure 5: Client-Server interaction

3.1 Server-Side Deployment and API Integration

The server is a critical component of the system, hosting the Progressive Growing GAN (PGAN) model to provide high-quality image generation and accurate fake image detection.

3.1.1 APIs Provided

The server exposes two primary API endpoints to support the application's core functionalities, as defined in the 'server.py' script:

- **/api/generate-face:** This endpoint handles requests for face generation. The client sends a GET request with a JSON payload specifying the generation type ("random" or "text-to-image") and an optional description for text-to-image generation. The server processes the request using the PGAN model and returns a JSON response containing a base64-encoded image and metadata.
- **/api/process-image:** This endpoint processes fake image detection requests. The client sends a GET request with a base64-encoded image in the JSON payload. The server uses the PGAN discriminator to classify the image as "Real" or "Fake," returning a JSON response with the classification, confidence score, and analysis.

3.1.2 Role of FastAPI and Uvicorn

FastAPI was selected for its ability to create efficient and scalable RESTful APIs. It leverages Python's type hints and asynchronous capabilities (via 'asyncio') to provide high performance. **Uvicorn**, as an ASGI server, runs the FastAPI application on port 8000, supporting asynchronous request processing. This setup ensures the server can handle multiple simultaneous requests from mobile clients.

3.1.3 Static IP via ngrok

During development, the server was hosted on a local machine, necessitating a static IP address to allow the Flutter application to connect reliably from external devices. **ngrok** was used to create a secure tunnel to the local server, providing a public URL for client access.

3.2 On-Device Integration

To enable offline functionality, the DCGAN model was integrated into the Flutter app for on-device image generation and fake image detection using the `flutter_onnxruntime` library. The model was converted to ONNX format and quantized from 32-bit floating-point to 8-bit integers via dynamic quantization, reducing size and computational load while preserving accuracy. This ensured efficient inference on resource-constrained mobile devices.

4 Main Functions of the System and User Guide

4.1 Main Functions

The Flutter-based mobile application provides two core functions: fake image detection and random face generation, utilizing Deep Convolutional GAN (DCGAN) and Progressive Growing GAN (PGAN). These functions are designed to be user-friendly, catering to general users seeking to verify image authenticity or generate synthetic faces.

4.1.1 Fake Image Detection

This function allows users to determine whether an image is real or AI-generated. The application employs the discriminator component of the GAN models to analyze the image, classifying it as "Real" or "Fake" and displaying a confidence percentage (e.g., "Real image, Confidence: 51.5%"). The detection process operates in two modes:

- **Online Mode:** When an internet connection is available, the image is sent to a server where the PGAN discriminator, trained on high-resolution CelebA-HQ images, processes it for optimal accuracy.
- **Offline Mode:** If the server is unavailable or the device is offline, the application uses the on-device DCGAN discriminator, trained on 64×64 CelebA images, ensuring continuous functionality.

The application automatically switches between modes based on connectivity, prioritizing PGAN for better accuracy when possible.

4.1.2 Image Generation

The image generation function enables users to create random human face images using the generator component of the GAN models. Upon pressing the "Generate Image" button, the application offers two quality options:

- **High Quality (512×512 pixels):** Utilizes the server-based PGAN, requiring an internet connection, to produce detailed images.
- **Standard Quality (64×64 pixels):** Employs the on-device DCGAN, functioning offline, suitable for resource-constrained devices.

The generated image can be saved to the device or shared via the "Download Image" button, which opens sharing options to platforms like social media, Google Drive, or the Photos app. Additionally, a text-to-image generation feature is under development, allowing users to input descriptions (e.g., "blonde woman, blue eyes") to create customized faces, though it is not yet fully functional.

4.2 Usage Instructions

The application features an intuitive interface, ensuring ease of use across Android devices, optimized for both phones and tablets. Below are step-by-step instructions for utilizing its main features.

4.2.1 Launching the Application

1. Install the application by downloading and installing the APK file create from running the project repository (https://github.com/biabeogo147/GAN-Application).
2. Grant camera and storage permissions upon first launch to enable full functionality.
3. The main screen displays two buttons: "Detect Fake" for fake image detection and "Generate Image" for face generation.

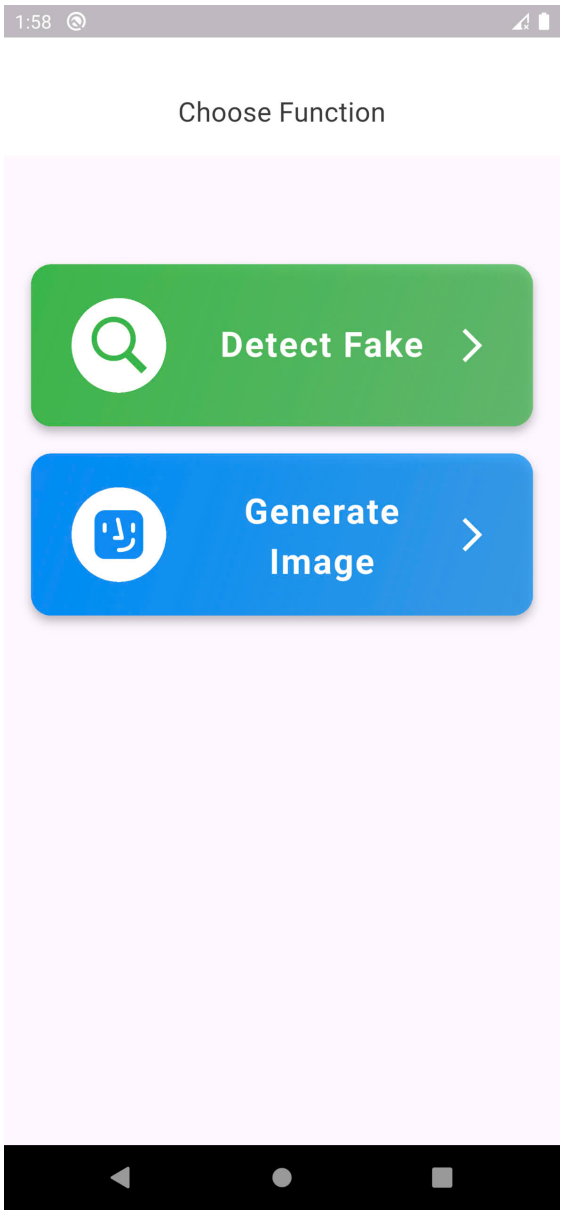

Figure 6: Main screen of the application with "Detect Fake" and "Generate Image" buttons.

4.2.2 Using Fake Image Detection

1. Tap the "Detect Fake" button on the main screen.
2. Choose an option:
   - Capture a new photo using the camera, which displays a live preview with a shutter button and a camera flip icon.
   - Select an existing image from the device gallery.

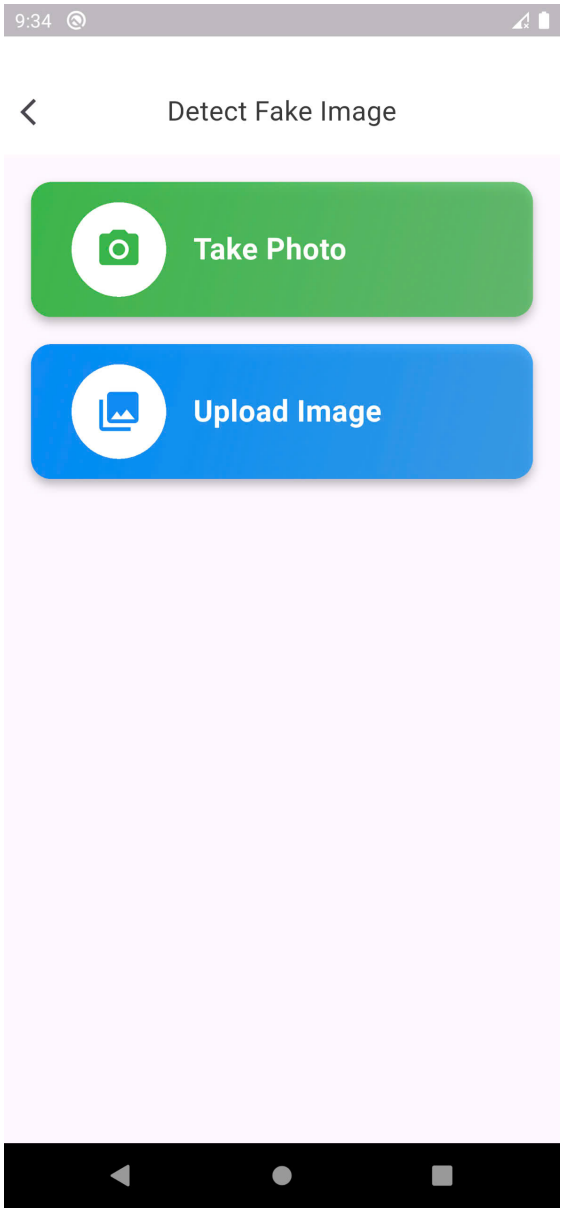

Figure 7: Take Photo or Upload Image

3. After capturing or selecting an image, tap the "Detect Fake" button to process the image.

4. The application displays the result, classifying the image as "Real" or "Fake" with a confidence percentage (e.g., "Real image, Confidence: 51.5%").

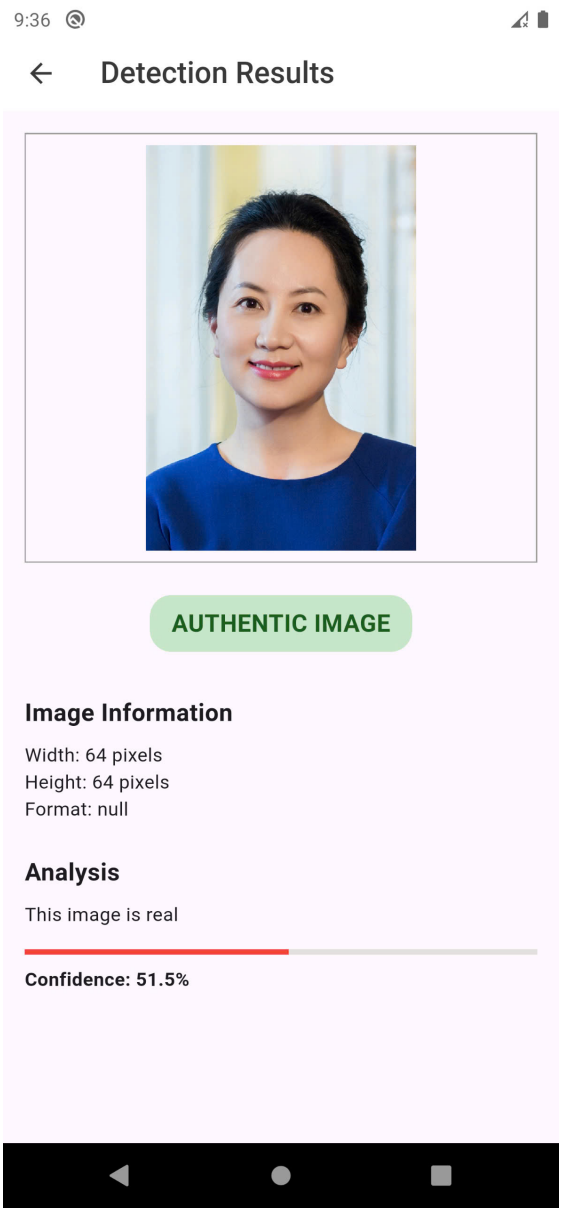

Figure 8: Detection result screen showing the classification and confidence percentage.

4.2.3 Using Image Generation

1. Tap the "Generate Image" button on the main screen.
2. Choose "Generate Random Face" or "Generate from Description".

   Note: A text-to-image generation feature is under development and currently non-functional.

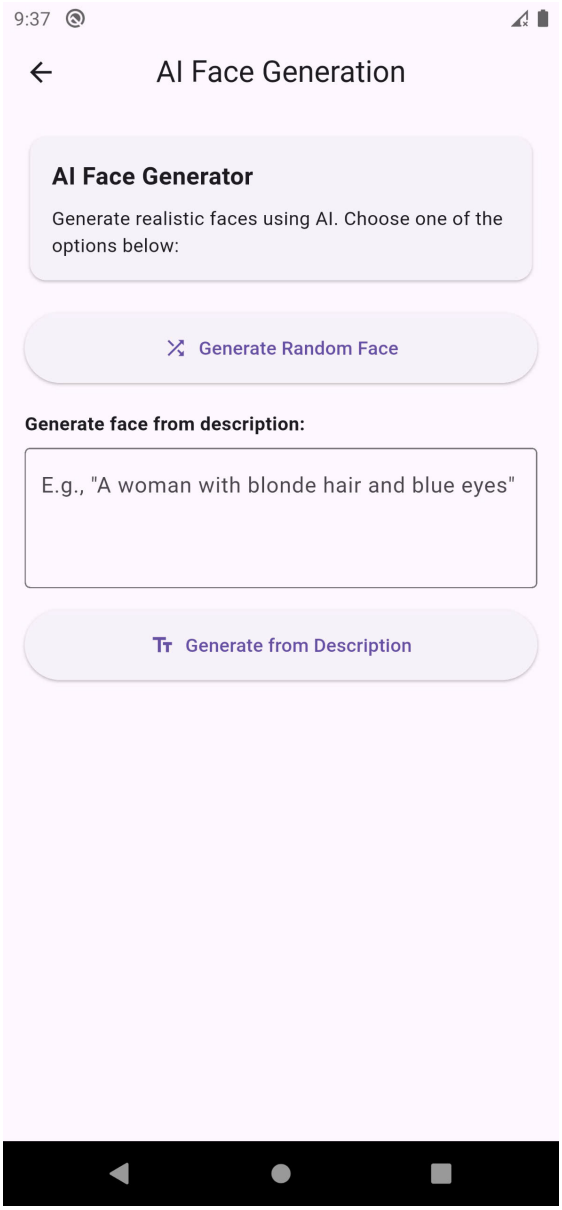

Figure 9: Navigation screen of Generation function

3. If you chose the first option, the application automatically generates a random human face, prioritizing the server-based PGAN if connected, or the on-device DCGAN if offline.

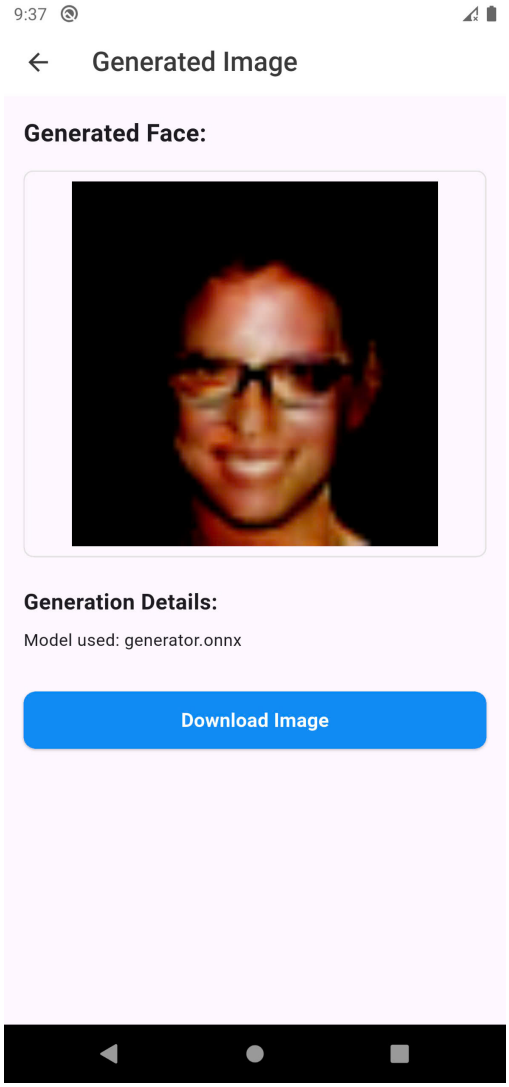

Figure 10: Image generation result screen displaying the generated face and "Download Image" button.

4. The generated image is displayed with a "Download Image" button, allowing users to save it to the device or share it via apps like social media, Google Drive, or Photos.

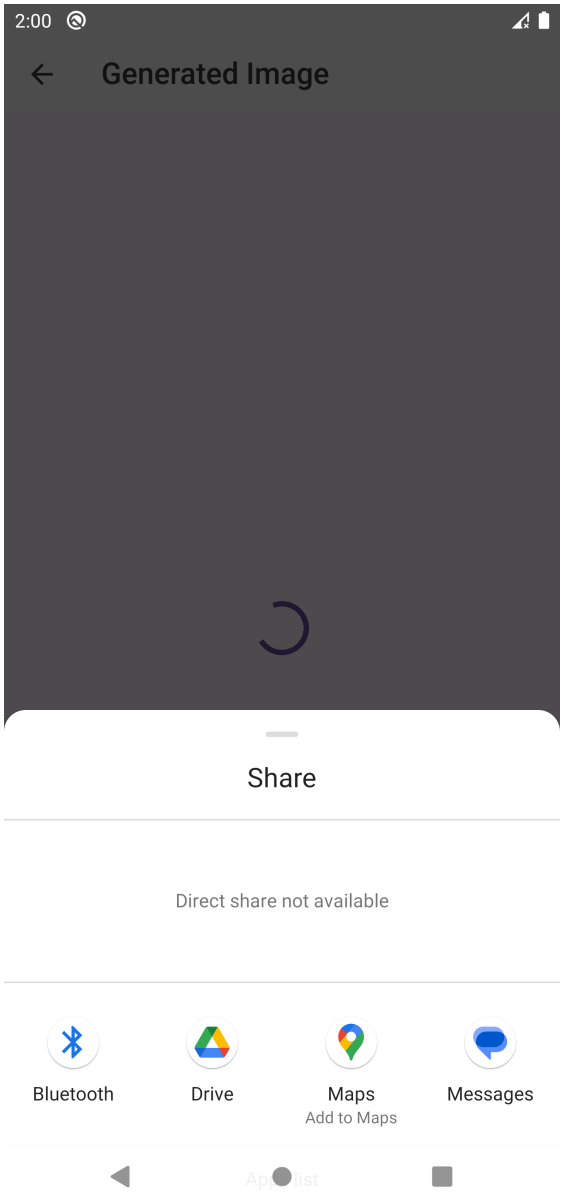

Figure 11: Download/Share Screen

5 Hardware Resources, Software Packages, Dataset

5.1 Hardware Resources

To support the computational demands of model training and inference, the project leverages the following hardware resources:

- NVIDIA RTX 4060: Utilized on personal computer for DCGAN model training
- NVIDIA A100: Accessed via Google Colab (Pro/Pro+) for PGAN model training and experimentation

5.2 Software Packages

Machine Learning and AI:

- PyTorch: Primary framework for developing and training GAN models
- ONNX\ONNX Runtime: For model conversion and cross-platform compatibility
- NumPy: For numerical operations and data manipulation
- OpenCV: For image processing and manipulation

Mobile Application Development:

- Flutter: Cross-platform framework for mobile application development
- Dart: Programming language for Flutter application logic
- Flutter Camera: For capturing images within the application
- Flutter Image Picker: For selecting images from device gallery
- Flutter ONNX Runtime: For integrating ML models into the mobile application

Server-side Technologies:

- FastAPI: Python web framework for RESTful API development
- Uvicorn: Asynchronous Server Gateway Interface for FastAPI
- Nginx: Web server for reverse proxy and load balancing
- Docker: For containerization and deployment server
- Base64: For encoding and decoding to transmit images between client and server

Development and Deployment Tools:

- Git: Version control system
- GitHub: Repository hosting and collaboration
- Android Studio: IDE for Android application development
- Visual Studio Code: Code editor for server-side development
- Jupyter Notebook: For model prototyping and experimentation

5.3 Dataset

Our GAN models were trained on carefully selected datasets to ensure high-quality output and effective detection capabilities:

CelebA Dataset:

- Contains over 200,000 celebrity face images
- Provides diverse facial attributes and expressions
- Used for training the DCGAN model (64×64 resolution)
- Preprocessed to standardize face alignment and cropping

CelebA-HQ Dataset:

- High-quality version of CelebA with 30,000 images
- Resolution up to 1024×1024 pixels
- Used for training the PGAN model (up to 512×512 resolution)
- Provides consistent high-quality facial details

Data Preprocessing:

- Face alignment using facial landmarks
- Center cropping to focus on facial features
- Resizing to target resolutions (64×64 for DCGAN, progressive resolutions for PGAN)
- Normalization to range [-1, 1] for model input

6 Challenges and Solution

6.1 Challenges Faced

Throughout the development of our GAN application, we encountered several significant challenges:

Dataset preprocessing and Model Training:

- Difficulties in finding a dataset suitable for the model
- Difficulties in replicating the PGAN model according to the paper
- Balancing generator and discriminator training: Preventing one network from overwhelming the other
- Hyperparameter sensitivity: Small changes causing significant training disruptions

Mobile Deployment Constraints:

- Model size limitations: Full PGAN model too large for mobile devices
- Computational resource constraints: Limited processing power for real-time generation
- Memory usage optimization: Preventing application crashes during model inference

Server Integration Issues:

- Latency in image transmission: Delays affecting user experience
- Server load balancing: Handling multiple simultaneous requests
- API reliability: Ensuring consistent service availability
- Security concerns: Protecting user data during transmission

Image Quality and Performance Tradeoffs:

- Balancing image quality with generation speed
- Optimizing detection accuracy without excessive false positives
- Managing user expectations between online and offline capabilities

6.2 Solutions Applied

To address these challenges, we implemented several technical solutions:

For GAN Training Stability:

- Applied batch normalization in DCGAN to stabilize training
- Employed learning rate scheduling to adapt throughout training
- Implemented early stopping based on Fréchet Inception Distance (FID) metrics

For Mobile Deployment:

- Utilized ONNX for model optimization and quantization
- Implemented asynchronous processing to prevent UI freezing
- Created hybrid online/offline approach with different quality options
- Optimized memory management during image processing

For Server Integration:

- Implemented efficient base64 encoding and decoding for image transmission
- Developed RESTful API with standardized request/response formats
- Optimized server-side processing for faster response times

These solutions enabled us to deliver a robust application that balances high-quality image generation and detection with practical performance considerations across various devices and network conditions.

7 Results and Evaluation

Qualitative Assessment:

- PGAN generated images display high-resolution details including:
  - Fine facial features
  - Realistic skin textures
  - Detailed hair rendering
  - Natural lighting effects
- DCGAN generated images show good quality at lower resolution:
  - Consistent facial structure
  - Recognizable features
  - Appropriate coloration
  - Some limitations in fine details

Performance Metrics: Processing Speed:

- Server-based detection: 4.2 seconds per image
- On-device detection: 0.5 seconds per image

8 Discussion

Comparative Analysis of DCGAN vs. PGAN:

- Image Quality: PGAN consistently produces higher quality images with more realistic details, particularly at higher resolutions. DCGAN generates acceptable results at lower resolutions but struggles with fine details.
- Training Stability: PGAN's progressive approach demonstrates significantly better training stability, with fewer instances of mode collapse and more consistent convergence.
- Computational Requirements: DCGAN is substantially more efficient in terms of both training and inference, making it suitable for on-device deployment. PGAN requires more substantial computational resources, necessitating server-side deployment.

Mobile AI Deployment Considerations:

- The hybrid approach (on-device DCGAN + server PGAN) provides an effective balance between quality, performance, and availability.
- Model quantization and optimization techniques can significantly reduce inference time without substantial quality degradation.
- Privacy considerations favor on-device processing where possible, particularly for sensitive image analysis.

9 Conclusion

9.1 Summary

This project has successfully developed a mobile application that harnesses the power of Generative Adversarial Networks to address the dual challenges of fake image detection and high-quality image generation. By implementing both DCGAN and PGAN architectures, we have created a solution that balances performance, quality, and accessibility.

- The Flutter-based mobile application provides an intuitive interface for both generating and detecting synthetic images
- Server-side PGAN implementation delivers high-resolution (512×512) face images with realistic details
- On-device DCGAN enables offline generation at moderate resolution (64×64)
- The discriminator components provide effective detection of manipulated and synthetic images
- A hybrid architecture ensures functionality across varying network conditions

9.2 Future Directions

Looking forward, this project establishes a foundation for several promising directions of future development:

- Integration of more advanced GAN architectures like StyleGAN3 for even higher quality and controllable generation
- Expansion to video generation and detection capabilities
- Development of more sophisticated on-device models to reduce server dependence
- Implementation of user-prompted generation features with attribute control

References

[1] Nathan Inkawhich. *DCGAN Tutorial*.

[2] Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen.
Progressive Growing of GANs for Improved Quality, Stability, and Variation.

[3] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, Aaron Courville. *Improved Training of Wasserstein GANs*.

This project is available on my github: GAN-Application