
# 2025 Jan – 60.001: Applied Deep Learning

## Team 3: ADL Deepfakers


| | |
|---|---|
| 1006662 | Lim Sing Thai Tiger |
| 1004931 | Tang Zhi-Ju Edward |
| 1006947 | Chow Liang Zhi |
| 1006878 | Gay Kai Feng Matthew |
| 1006866 | Joel Lim |


## GANFingerprint Deepfake detection model documentation

## Contents

# GANFingerprint: More about GANFingerprint and its relationship with deepfake recognition

## What are GAN Fingerprints?

GAN Fingerprints are distinctive patterns or traces that are unintentionally embedded in images generated by Generative Adversarial Networks (GANs). These GAN fingerprints are akin to real human fingerprints, with the comparison that humans unintentionally leave fingerprints on the items they touch, that can be used to trace their identities. Just like human fingerprints, these GAN Fingerprints are unique to the GAN architecture the images are generated from, due to these factors:

1. Each GAN architecture has its own unique way of generating images based on its specific design, loss functions, and optimization methods.
2. Even GANs with identical architectures but different training datasets, random initializations, or hyperparameters will produce images with subtly different characteristics.

## Objective of our project

With GAN image generation techniques becoming more advanced, there may be difficulties identifying deepfake images through existing methods, such as detecting distortions in facial features and image details. Through our project, we hope to create a deepfake detection model that can identify deepfake images reliably, no matter how realistic the generated images are to the human eye. By customizing and creating a model that can discriminate deepfake images from real ones through their GAN Fingerprint profiles, we hope to come up with a more sophisticated model which can capture details invisible to the human eye.

## Summary of GANFingerprint model

The GANFingerprint model implements a deepfake detection system based on multi-domain feature extraction and analysis. The architecture integrates a modified ResNet34 backbone with specialized components for frequency domain analysis, feature fusion, and fingerprint detection. The model's design prioritizes both effectiveness in detection accuracy and reproducibility of results through deterministic operations.

The model employs a weighted metrics approach during training that prioritizes recall, which is critical for reliable deepfake detection. It includes Grad-CAM visualization capabilities for model explainability, comprehensive experiment tracking, and probability calibration for more interpretable results. These features collectively enhance both the performance and transparency of the deepfake detection process.

## Model Directory

```
deepfake_detector/
├── config.py                # Configuration parameters
├── data_loader.py           # Dataset and dataloader implementation
├── models/
│   ├── __init__.py          # Module initialization
│   ├── fingerprint_net.py   # GANFingerprint model architecture
│   ├── layers.py            # Custom layers and blocks
├── train.py                 # Training script
├── evaluate.py              # Evaluation script
├── inference.py             # Inference on new images
├── GANFingerprint.ipynb     # Jupyter notebook file with step-by-step guidance on how to run the model
├── utils/
│   ├── __init__.py          # Utilities module initialization
│   ├── reproducibility.py   # Random seed and reproducibility utilities
│   ├── visualization.py     # Plotting and visualization tools
│   ├── metrics.py           # Performance metrics calculation
│   ├── augmentations.py     # Advanced augmentation techniques
│   ├── experiment.py        # Logging of information when training model
│   ├── gradcam.py           # Grad-CAM visualization of inference results
├── checkpoints/             # Directory for saved model checkpoints
├── logs/                    # TensorBoard logs and training records
├── requirements.txt         # Contains all required dependencies
```

## Files and Functions

### 1. config.py

This file defines all configuration parameters and settings for the project.

**Key Components:**

- **Dataset paths**: Defines directory structure for train/val/test splits and real/fake classes
- **Model parameters**: Input size, backbone architecture, embedding dimension, dropout rate
- **Training parameters**: Batch size, worker count, learning rate, weight decay, epoch count
- **Early stopping**: Patience parameters and warmup epochs
- **Mixed precision**: Flag for automatic mixed precision training
- **File paths**: Directories for checkpoints and logs
- **Reproducibility**: Seeds and deterministic operation flags

### 2. data_loader.py

Handles dataset creation, transformations, and loading with reproducibility guarantees.

**Functions in chronological order:**

1. worker_init_fn(worker_id):
    - Initializes each DataLoader worker with a separate but deterministic seed
    - Ensures reproducible data loading across different runs
2. DeepfakeDataset class:
    - Constructor: Sets up paths, applies transforms, handles seed setting
    - _valid_file(): Validates file extensions for images
    - **len**(): Returns dataset size
    - **getitem**(): Loads and transforms images with deterministic processing for validation/test
3. get_transforms(phase):
    - Creates different transformation pipelines for train/val/test phases
    - Training transforms: Random crops, flips, color jitter, custom augmentations
    - Validation/test transforms: Simple resize and normalization
4. get_dataloaders(seed=42):
    - Creates datasets with deterministic seed
    - Sets up samplers with fixed seeds
    - Returns reproducible data loaders for train, validation, and test sets
5. get_dataset_stats():
    - Prints statistics about dataset size for monitoring

## 3. models/layers.py

Contains custom neural network layers and blocks tailored for deepfake detection.

**Classes in order of dependency:**

1. SpatialAttention:
    - Applies channel-wise attention to focus on important spatial regions
    - forward(): Generates and applies attention maps to input features
2. SelfAttention:
    - Implements self-attention mechanism to model feature relationships

- forward(): Computes query, key, value projections and attention maps

3. FrequencyAwareness:
    - Enhances detection of frequency domain artifacts
    - forward(): Processes features through spatial branch and high-pass filter branch
4. FingerprintBlock:
    - Core block combining frequency awareness with spatial attention
    - forward(): Applies frequency-aware layers, spatial attention, and feature dropout
5. DCTLayer:
    - Approximates Discrete Cosine Transform using FFT
    - forward(): Performs FFT, extracts magnitude, and applies log scaling
6. FrequencyEncoder:
    - Encodes frequency domain information using CNN layers
    - Constructor: Sets up CNN layers for processing frequency features
    - forward(): Applies DCT to each channel, processes through CNNs

## 4. models/fingerprint_net.py

Defines the main FingerprintNet model architecture for deepfake detection.

**Classes and methods:**

1. FingerprintNet class:
    - Constructor: Initializes backbone network and specialized layers
        - Sets up multi-path feature extraction (low/mid/high)
        - Creates adaptation layers
        - Sets up frequency encoder
        - Creates fusion layer, attention, fingerprint block
        - Initializes embedding layer and classifier
    - _initialize_weights(): Initializes model weights deterministically

- forward(x): Main inference function
    - Extracts multi-level spatial features
    - Adapts features to common dimensions
    - Extracts frequency domain features
    - Fuses all features
    - Applies attention and fingerprint block
    - Performs global average pooling
    - Computes embedding and classification
- fingerprint_distance(x1, x2): Calculates distance between image fingerprints

2. EnhancedClassifier (nested class):
    - Constructor: Sets up layers and residual connection
    - forward(x): Processes input through layers with residual connection

## 5. utils/reproducibility.py

Utilities for ensuring reproducible model behavior.

**Functions in chronological order:**

1. set_all_seeds(seed=42):
    - Sets seeds for Python, NumPy, and PyTorch random generators
    - Configures deterministic operations for PyTorch
    - Sets environment variables for reproducibility
2. get_random_state():
    - Captures the current random state from all sources
    - Returns a dictionary with all random states
3. set_random_state(random_state):
    - Restores previously captured random states
    - Ensures consistent behavior when resuming experiments

## 6. utils/metrics.py

Provides functions for computing and analyzing model performance metrics.

**Functions in chronological order:**

1. compute_metrics(y_true, y_pred_probs, threshold=0.5):
    - Calculates classification metrics (accuracy, precision, recall, F1, AUC)
    - Converts probabilities to binary predictions based on threshold
2. compute_confusion_matrix(y_true, y_pred):
    - Computes confusion matrix for classification results
3. get_roc_curve_data(y_true, y_pred_probs):
    - Calculates ROC curve data points (false positive rate, true positive rate)
4. get_precision_recall_curve_data(y_true, y_pred_probs):
    - Calculates precision-recall curve data points
5. find_optimal_threshold(y_true, y_pred_probs, metric='f1'):
    - Finds optimal classification threshold by scanning over possible values
    - Optimizes for selected metric (F1-score, accuracy, precision or recall)

## 7. utils/visualization.py

Tools for visualizing model performance and predictions.

**Functions in chronological order:**

1. plot_training_curves(train_losses, val_losses, train_metrics, val_metrics, metric_name='accuracy'):
    - Creates plots for training and validation losses/metrics
2. plot_confusion_matrix(cm, classes, normalize=False, title='Confusion Matrix', cmap=plt.cm.Blues):
    - Visualizes confusion matrix with class labels
3. plot_roc_curve(y_true, y_pred_probs):
    - Generates ROC curve plot with AUC score
4. plot_precision_recall_curve(y_true, y_pred_probs):
    - Generates precision-recall curve plot with average precision
5. visualize_model_predictions(model, dataloader, device, num_images=8):
    - Shows model predictions on sample images with true/predicted labels
6. visualize_attention_maps(model, image_tensor, device):

- Visualizes attention maps from the model for a given image

## 8. utils/augmentations.py

Implements specialized augmentation techniques for deepfake detection.

**Classes in chronological order:**

1. JPEGCompression:
   - Simulates varying JPEG compression artifacts
   - **call**(): Applies random quality JPEG compression to images
2. AddNoiseGaussian:
   - Adds Gaussian noise to simulate camera sensor noise
   - **call**(): Converts image to tensor, adds noise, converts back to PIL
3. VariableBlur:
   - Applies varying levels of blur to simulate camera focus issues
   - **call**(): Applies Gaussian blur with random radius
4. ColorQuantization:
   - Simulates color quantization artifacts from image compression
   - **call**(): Quantizes color values to fewer bits

## 9. utils/experiment.py

Module that provides comprehensive experiment tracking and logging capabilities.

**Classes and methods:**

1. ExperimentTracker class:
   - Constructor: Initializes experiment directory with timestamped folder
   - save_config(): Saves configuration to JSON file
   - log(): Logs messages with timestamps
   - save_results(): Saves training results (epoch metrics or summary)

The ExperimentTracker enables structured logging of all training parameters, metrics, and outcomes, which facilitates experiment reproducibility and comparison.

## 10. utils/gradcam.py

Module that implements Gradient-weighted Class Activation Mapping for model explainability.

**Classes and functions:**

1. GradCAM class:
    - Constructor: Sets up model hooks for accessing gradients and activations
    - save_activation(): Callback to store activations from forward pass
    - save_gradient(): Callback to store gradients from backward pass
    - **call**(): Generates CAM heatmap by weighting activations with gradients
2. get_gradcam_layer(): Helper function to identify target layer for visualization
3. generate_gradcam(): Function to create complete Grad-CAM visualization
4. visualize_gradcam(): Create visualization with original image and heatmap overlay

This module allows visualization of which image regions most influenced the model's decision, providing crucial interpretability to the detection process.

## 11. train.py

Main training script for the GANFingerprint model.

**Functions in chronological order:**

1. get_lr_scheduler(optimizer, warmup_epochs, total_epochs):
    - Creates learning rate scheduler with warmup and cosine annealing
2. train(args):
    - Main training function:
        - Sets up seeds and directories
        - Initializes ExperimentTracker for logging
        - Loads data loaders and initializes model
        - Sets up loss function, optimizer, and scheduler
        - Initializes AMP scaler if using mixed precision
        - Handles checkpoint resumption if specified
        - Implements training loop with validation
        - Computes weighted metrics combination for model selection

- Logs comprehensive metrics to experiment tracker and TensorBoard
- Handles early stopping based on combined metric score
- Saves checkpoints periodically and for best models with random states

The training process uses a weighted metrics approach that prioritizes recall while balancing other metrics, providing better model selection for the deepfake detection task.

## 12. evaluate.py

Evaluation script for the trained model.

**Functions in chronological order:**

1. evaluate(checkpoint_path, output_dir):
    - Sets reproducibility seeds
    - Creates output directory
    - Loads test data loader
    - Initializes and loads model from checkpoint
    - Restores random state if available
    - Performs evaluation with prediction collection
    - Calculates metrics (accuracy, precision, recall, F1, AUC)
    - Generates and saves visualization plots
    - Outputs results to files

## 13. inference.py

Script for applying the trained model to new images with enhanced visualization capabilities.

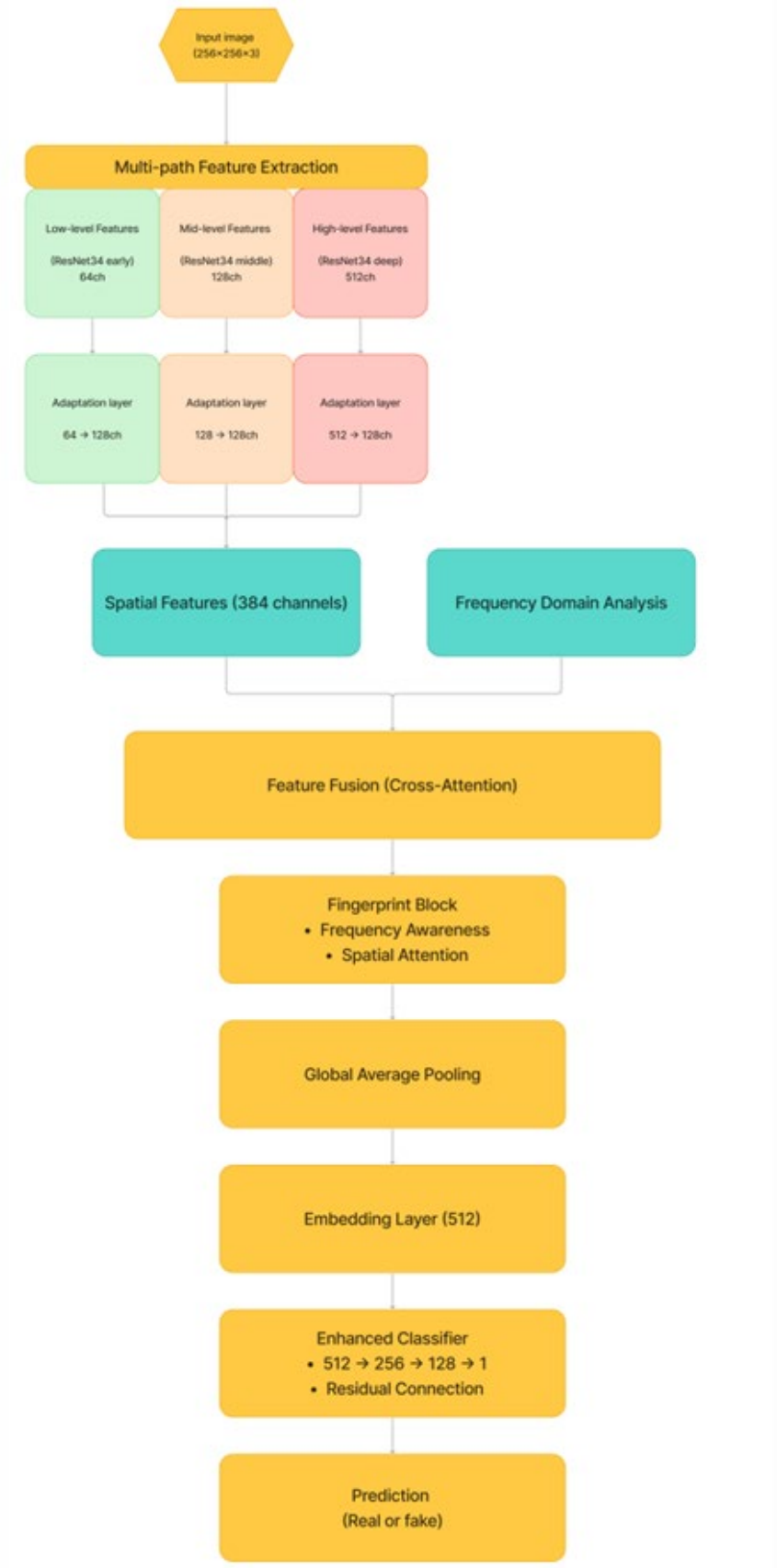
**Functions in chronological order:**

1. predict_image_calibrated(model, image_path, transform):
    - Loads and preprocesses a single image
    - Runs model inference
    - Applies probability calibration for more interpretable results
    - Returns calibrated prediction probability and class label

2. extract_true_label(filename):
    - Extracts true label from filename (if available)
3. calculate_metrics(true_labels, pred_labels):
    - Calculates performance metrics for batch inference
4. visualize_result(image_path, prob, pred_class, true_class=None, output_path=None):
    - Creates visualization of image with prediction
    - Highlights result based on prediction class
    - Integrates true label if available
5. run_inference(checkpoint_path, input_path, output_dir=None, batch_mode=False, use_gradcam=False):
    - Sets up reproducibility
    - Loads model from checkpoint
    - Handles single image or batch mode processing
    - Implements Grad-CAM visualization if requested
    - Generates comprehensive reports and visualizations
    - Calculates performance metrics for batches with known labels
    - Saves outputs to specified directory

The inference script provides multiple visualization options, batch processing capabilities, and integrated performance metrics analysis.

## Model Architecture



# Detailed Explanation of how the GANFingerprint model works

## 1. Data Preprocessing and Input Transformation

### 1.1 Input Standardization

The initial stage of processing involves transforming input images into a standardized format suitable for neural network analysis. The data_loader.py module implements this process with the following critical operations:

1. Image loading and conversion to RGB format
2. Resizing to a uniform input dimension (256×256 pixels)
3. Application of appropriate transformations based on operational context
4. Normalization using ImageNet statistics (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

### 1.2 Context-Specific Transformations

The model applies distinct transformation pipelines depending on the operational phase:

**Training phase transformations:**

- Random resized cropping and horizontal flipping for data augmentation
- Color jitter and random affine transformations
- Specialized augmentations that simulate real-world artifacts:
    - JPEG compression with variable quality
    - Gaussian noise addition to simulate sensor noise
    - Variable blur to simulate focus variations
    - Random erasing for robustness against partial occlusions

**Validation/Testing phase transformations:**

- Deterministic resizing and center cropping
- Standardized normalization without augmentation

These transformations are implemented with reproducibility in mind, using fixed seeds to ensure consistent results across multiple runs.

## 2. Multi-Path Feature Extraction

### 2.1 Backbone Segmentation

Upon receiving the normalized input tensor, the model initiates a multi-path feature extraction process through the segmented ResNet34 backbone:

1. **Low-level feature extraction path:** Utilizes early layers of ResNet34 (up to layer1) to capture fundamental visual elements such as edges, textures, and color transitions, producing feature maps with 64 channels.
2. **Mid-level feature extraction path:** Employs intermediate layers (layer2) to extract more complex patterns and structural information, generating feature maps with 128 channels.
3. **High-level feature extraction path:** Utilizes deeper layers (layer3 and layer4) to extract semantic and contextual information, yielding feature maps with 512 channels.

### 2.2 Feature Adaptation

Each feature extraction path produces representations with different dimensionalities. To enable effective fusion, the model applies adaptation layers:

```
low_adapted = self.low_adapter(F.adaptive_avg_pool2d(low_features,
high_features.shape[2:]))

mid_adapted = self.mid_adapter(F.adaptive_avg_pool2d(mid_features,
high_features.shape[2:]))

high_adapted = self.high_adapter(high_features)
```

These adaptation layers perform:

1. Spatial dimension matching through adaptive pooling
2. Channel dimension standardization to 128 channels per path
3. Feature normalization and non-linear activation

This standardization enables the subsequent fusion of features extracted at different levels of abstraction.

## 3. Frequency Domain Analysis

### 3.1 Spectral Transformation

Parallel to spatial feature extraction, the model performs frequency domain analysis through the FrequencyEncoder component. This analysis is crucial for detecting GAN-specific artifacts that may be imperceptible in the spatial domain but evident in the frequency spectrum.

The frequency analysis process involves:

1. Application of Discrete Cosine Transform (DCT) approximation using Fast Fourier Transform (FFT):

```
x_fft = torch.fft.rfft2(x)
x_magnitude = torch.abs(x_fft)
x_magnitude = torch.log(x_magnitude + 1e-10)
```

2. Processing of spectral representations through convolutional layers to extract frequency domain features:

```
x_freq = self.conv1(x_dct)
x_freq = self.conv2(x_freq)
x_freq = self.conv3(x_freq)
```

   This process yields 256 channels of frequency domain features that complement the spatial analysis.

### 3.2 Significance of Frequency Analysis

The frequency domain analysis is particularly significant because GAN-generated images often contain characteristic spectral signatures resulting from the generation process. These signatures differ from the natural frequency distributions found in authentic photographs, which are governed by optical physics and sensor properties.

## 4. Feature Fusion and Attention Mechanisms

### 4.1 Multi-Domain Integration

After extracting features from both spatial and frequency domains, the model performs feature fusion through concatenation followed by dimensional reduction:

```
fused = torch.cat([low_adapted, mid_adapted, high_adapted, freq_features], dim=1)
fused = self.fusion_layer(fused)
```

This fusion combines four distinct feature types (three spatial scales plus frequency analysis) into a unified representation.

### 4.2 Self-Attention Mechanism

The fused features then undergo self-attention processing to weight spatial locations and feature channels according to their relevance:

```
fused = self.attention(fused)
```

The SelfAttention module implements a dot-product attention mechanism:

1. Query, key, and value projections are generated from the input features
2. Attention scores are computed through matrix multiplication of query and key projections
3. Softmax normalization is applied to obtain attention weights
4. The value projection is weighted by the attention weights
5. A residual connection with a learnable parameter combines the attention output with the original input

This attention mechanism enables the model to emphasize discriminative patterns while suppressing less relevant information.

## 5. Fingerprint Feature Enhancement

### 5.1 The Fingerprint Block

The model employs a specialized FingerprintBlock to enhance patterns characteristic of GAN-generated images:

```
features = self.fingerprint_block(fused)
```

The FingerprintBlock consists of:

1. A FrequencyAwareness module that processes features through two branches:
    - Spatial branch: Standard convolutional processing
    - Frequency branch: High-pass filtering to enhance frequency components
2. A SpatialAttention module that generates and applies channel-wise attention maps
3. Feature dropout for regularization

This specialized block accentuates subtle artifacts and inconsistencies that differentiate synthetic images from authentic photographs.

## 6. Global Feature Aggregation and Embedding

### 6.1 Global Pooling

Following the fingerprint enhancement, the model applies global average pooling to reduce spatial dimensions while preserving channel information:

```
features = self.gap(features)
features = features.view(features.size(0), -1)
```

This operation condenses the spatial information into a compact feature vector.

### 6.2 Embedding Layer

The pooled features are then processed through an embedding layer:

```
embedding = self.embedding(features)
```

This layer transforms the features into a 512-dimensional embedding vector through:

1. Linear transformation
2. Batch normalization
3. ReLU activation
4. Dropout regularization

The resulting embedding vector can be conceptualized as a "fingerprint" of the input image, capturing the essential characteristics that differentiate authentic images from GAN-generated ones.

## 7. Classification Process

### 7.1 Enhanced Classifier Architecture

The final classification occurs through an enhanced multi-layer classifier:

```
logits = self.classifier(embedding)
```

The classifier implements:

1. Three fully connected layers (512→256→128→1) with batch normalization and LeakyReLU activations
2. A residual connection to preserve information flow

3. Graduated dropout rates for effective regularization

### 7.2 Prediction Generation

The classifier outputs a scalar logit value that is converted to a probability through the sigmoid function:

```
return logits.squeeze() # Sigmoid applied during loss calculation or post-processing
```

This probability represents the model's assessment of image authenticity, with values closer to 1 indicating genuine images and values closer to 0 indicating GAN-generated images.

During inference, a threshold (typically 0.5) is applied to determine the final binary classification:

- Probability ≥ 0.5: Classified as real
- Probability < 0.5: Classified as fake (GAN-generated)

## 8. Integrated Data Flow

The complete data flow through the GANFingerprint model can be summarized as follows:

1. Input image → Preprocessing and normalization → Normalized tensor
2. Normalized tensor → Multi-path feature extraction → Low/mid/high-level features
3. Normalized tensor → Frequency encoder → Frequency domain features
4. All features → Feature adaptation → Standardized feature maps
5. Standardized features → Feature fusion → Combined representation
6. Combined representation → Self-attention → Attention-weighted features
7. Attention-weighted features → Fingerprint block → Enhanced fingerprint features
8. Enhanced features → Global average pooling → Pooled feature vector
9. Pooled vector → Embedding layer → Fingerprint embedding
10. Fingerprint embedding → Enhanced classifier → Authenticity prediction

## Training Process with Weighted Metrics

The GANFingerprint model employs a sophisticated training process designed to optimize performance specifically for deepfake detection challenges.

### 1. Experiment Tracking and Logging

The training process incorporates comprehensive experiment tracking through the ExperimentTracker class:

```
tracker = ExperimentTracker(config.EXPERIMENT_NAME,
base_dir=config.EXPERIMENT_LOGS)
```

This system provides:

1. **Structured Directory Organization**: Creates timestamped experiment directories for each training run
2. **Configuration Storage**: Saves all hyperparameters and settings in JSON format
3. **Chronological Logging**: Records all training events with timestamps
4. **Metric Tracking**: Saves per-epoch metrics and final performance summaries
5. **Reproducibility Information**: Records random states and seed configurations

This systematic approach to experiment management allows for better comparison between different model iterations and more reliable reproduction of results.

### 2. Weighted Metrics Approach

A key innovation in the training process is the weighted metrics approach for model selection:

```
metric_weights = {
    'accuracy': 0.10,
    'precision': 0.05,
    'recall': 0.70,      # Higher weight for recall
    'f1': 0.10,
    'auc': 0.05
}

# Calculate the weighted combination score
combined_score = sum(metric_weights[metric] * val_metrics[metric]
                     for metric in metric_weights.keys())
```

This approach:

1. **Prioritizes Recall**: Assigns a 70% weight to recall (true positive rate), which is critical for reliable deepfake detection where missing a fake image is costlier than incorrectly flagging a real one
2. **Balances Multiple Aspects**: Incorporates accuracy, precision, F1-score, and AUC to ensure overall performance quality
3. **Optimizes for Real-World Application**: Aligns model selection with the practical requirements of deepfake detection systems

The weighted metric optimization provides a more nuanced approach to model selection than simply using accuracy or AUC, resulting in models that better balance the various performance tradeoffs in deepfake detection.

## 3. Enhanced Learning Rate Scheduling

The training process implements an advanced learning rate scheduling strategy:

```
def get_lr_scheduler(optimizer, warmup_epochs, total_epochs):
    def lr_lambda(epoch):
        if epoch < warmup_epochs:
            return float(epoch) / float(max(1, warmup_epochs))
        return 0.5 * (1.0 + np.cos(np.pi * (epoch - warmup_epochs) /
(total_epochs - warmup_epochs)))

    return optim.lr_scheduler.LambdaLR(optimizer, lr_lambda)
```

This scheduler combines:

1. **Linear Warmup**: Gradually increases the learning rate during initial epochs to stabilize early training dynamics
2. **Cosine Annealing**: Smoothly decreases the learning rate following a cosine curve to facilitate fine convergence

This scheduling approach helps the model avoid poor local minima and converge more effectively to optimal weights.

## 4. Checkpoint Management

The training process implements comprehensive checkpoint management:

```
# Save the best model
checkpoint_path = os.path.join(config.CHECKPOINT_DIR,
f"{config.EXPERIMENT_NAME}_best.pth")
torch.save({
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'scheduler_state_dict': scheduler.state_dict() if scheduler else None,
    'combined_score': best_combined_score,
    'val_metrics': val_metrics,
    'metric_weights': metric_weights,  # Save the weights used
    'random_state': get_random_state(),
    'config': config_dict
}, checkpoint_path)
```

Key features include:

1. **Comprehensive State Preservation**: Saves model, optimizer, and scheduler states
2. **Metric Information**: Records validation metrics and scoring methodology
3. **Reproducibility Support**: Stores random states for exact reproduction of results
4. **Configuration Tracking**: Includes model configuration information

This thorough checkpoint system enables reliable model resumption and evaluation, facilitating incremental development and experimentation.

## Inference and Visualization

The GANFingerprint model provides sophisticated inference capabilities with robust visualization tools for model explainability and comprehensive batch processing.

### 1. Grad-CAM Visualization

The model integrates Gradient-weighted Class Activation Mapping (Grad-CAM) to enable visual explanation of model decisions:

```
def generate_gradcam(model, image_tensor, orig_image):
    # Get the target layer for Grad-CAM
    target_layer = get_gradcam_layer(model)

    # Initialize Grad-CAM
    grad_cam = GradCAM(model, target_layer)

    # Generate Grad-CAM
    raw_logit, heatmap = grad_cam(image_tensor)

    # Clean up hooks
    grad_cam.remove_hooks()

    # Create visualization
    # Convert PIL Image to numpy array
    img_np = np.array(orig_image)

    # Create color map
    heatmap_colored = cm.jet(heatmap)[:, :, :3]  # Remove alpha channel

    # Overlay heatmap on original image
    superimposed = (0.7 * heatmap_colored + 0.3 * img_np / 255.0)

    # Ensure values are in valid range
    superimposed = np.clip(superimposed, 0, 1)

    return raw_logit, heatmap, superimposed
```

This visualization technique:

1. **Highlights Discriminative Regions**: Shows which parts of the image most influenced the classification decision
2. **Provides Model Transparency**: Reveals whether the model is focusing on meaningful features or irrelevant patterns

3. **Facilitates Error Analysis**: Helps identify why the model made incorrect predictions
4. **Enhances Trust**: Provides evidence that the model is detecting genuine visual cues rather than statistical artifacts

The Grad-CAM implementation in the GANFingerprint model targets the final convolutional layer, where feature maps retain spatial information while encoding high-level semantic content related to deepfake artifacts.

## 2. Calibrated Probability System

The inference system implements probability calibration for more intuitive and interpretable results, since raw logits for fake predictions have consistently been highly negative, but the original output indicates false low confidence:

```
# Original probability
orig_prob = torch.sigmoid(output).item()


# Apply calibration for fake images
if orig_prob < 0.5:  # Predicted as fake
    # Map [0, 0.5] range to [1.0, 0] - this inverts the scale for fake images
    calibrated_prob = 1.0 - (2.0 * orig_prob)
else:  # Predicted as real
    calibrated_prob = orig_prob
```

This calibration:

1. **Standardizes Interpretation**: For both real and fake predictions, higher probability values indicate higher confidence
2. **Inverts Scale for Fake Images**: Remaps the [0, 0.5] range to [1.0, 0], making it more intuitive that larger values mean higher confidence
3. **Preserves Real Image Probabilities**: Maintains the original scale for images predicted as real

This approach results in a more intuitive confidence scoring system where a value of 0.95 consistently indicates high confidence (whether in a "real" or "fake" prediction), facilitating easier interpretation by users.

## 3. Batch Processing Capabilities

The inference system supports comprehensive batch processing of image directories:

```
def run_inference(checkpoint_path, input_path, output_dir=None,
batch_mode=False, use_gradcam=False):
    # Batch mode (directory of images)
    if batch_mode:
        # Process each image
        results = []
        for img_path in tqdm(image_files, desc=f"Processing images from
{input_folder_name}"):
            true_class = extract_true_label(img_path)
            # Processing and prediction logic...
            results.append((img_path, prob, pred_class, true_class))

        # Calculate metrics if true labels are available
        if true_labels and len(true_labels) == len(pred_labels):
            metrics = calculate_metrics(true_labels, pred_labels)

        # Save results to CSV
        # Print summary
        # Print metrics if available
```

Key features include:

1. **Automated Directory Processing**: Processes all images in a specified directory
2. **Structured Output Organization**: Creates organized result directories with logical naming
3. **Comprehensive Result Logging**: Saves detailed CSV files with predictions and confidence scores
4. **Performance Tracking**: Calculates metrics when ground truth labels are available
5. **Visualization Generation**: Creates visualizations for each processed image

This batch processing capability significantly enhances the model's utility for large-scale deepfake analysis tasks.

## 4. Performance Metrics Analysis

When processing batches of images with known labels, the inference system calculates comprehensive performance metrics:

```
def calculate_metrics(true_labels, pred_labels):
    # Count total predictions
    total = len(true_labels)

    # Count correct predictions
    correct = sum(1 for true, pred in zip(true_labels, pred_labels) if true
== pred)

    # For "Real" class metrics
    true_positives = sum(1 for true, pred in zip(true_labels, pred_labels)
                         if true == "Real" and pred == "Real")
    false_positives = sum(1 for true, pred in zip(true_labels, pred_labels)
                         if true == "Fake" and pred == "Real")
    false_negatives = sum(1 for true, pred in zip(true_labels, pred_labels)
                         if true == "Real" and pred == "Fake")
    true_negatives = sum(1 for true, pred in zip(true_labels, pred_labels)
                         if true == "Fake" and pred == "Fake")

    # Calculate metrics
    accuracy = correct / total if total > 0 else 0
    precision = true_positives / (true_positives + false_positives) if
(true_positives + false_positives) > 0 else 0
    recall = true_positives / (true_positives + false_negatives) if
(true_positives + false_negatives) > 0 else 0
    f1 = 2 * (precision * recall) / (precision + recall) if (precision +
recall) > 0 else 0

    return {
        "accuracy": accuracy,
        "precision": precision,
        "recall": recall,
        "f1": f1,
        "true_positives": true_positives,
        "false_positives": false_positives,
```

The metrics analysis:

1. **Calculates Standard Metrics**: Accuracy, precision, recall, and F1-score
2. **Counts Prediction Categories**: True positives, false positives, true negatives, and false negatives
3. **Generates Confusion Matrix**: Visual representation of prediction performance
4. **Provides Comprehensive Reports**: Saves results to CSV files and prints summary statistics

This functionality enables systematic evaluation of model performance across different datasets and operational contexts.

## Conclusion

The GANFingerprint model implements a sophisticated approach to deepfake detection through multi-domain analysis of images. By examining both spatial and frequency domains at multiple scales, the model identifies subtle artifacts characteristic of GAN-generated images. The integration of attention mechanisms and specialized fingerprint enhancement components further improves the model's ability to focus on discriminative patterns.

The model employs a weighted metrics approach during training that prioritizes recall, which is critical for the deepfake detection task where missing a fake image can have serious consequences. The Grad-CAM visualization capabilities provide transparency and explainability, showing which regions of an image influenced the model's decision.

The comprehensive experiment tracking system ensures reproducibility and facilitates systematic improvement of the model. The calibrated probability system and batch processing capabilities enhance the model's utility in real-world applications, providing intuitive confidence scores and efficient processing of large image collections.

Together, these features make the GANFingerprint model not only effective at detecting sophisticated deepfakes but also transparent and trustworthy in its operation—essential qualities for deployment in real-world scenarios where understanding model behaviour is as important as raw detection performance.