

BAB 4

HASIL DAN DISKUSI

4.1 Implementasi

Tahap implementasi merupakan realisasi dari rancangan metodologi yang telah dijelaskan pada bab sebelumnya. Seluruh proses, mulai dari pra-pemrosesan data hingga pelatihan model, diimplementasikan menggunakan bahasa pemrograman Python dengan bantuan beberapa pustaka utama.

4.2 Skenario Pengujian

Pengujian dilakukan untuk mengevaluasi kemampuan model yang telah dilatih dalam membedakan antara audio asli dan audio *deepfake* pada data yang belum pernah dilihat sebelumnya (data tes). Skenario pengujian adalah sebagai berikut:

1. Model dengan bobot terbaik yang disimpan selama pelatihan dimuat kembali.
2. Seluruh data tes (20% dari total dataset) dimasukkan ke dalam model untuk mendapatkan prediksi.
3. Prediksi dari model dibandingkan dengan label asli dari data tes.
4. Berdasarkan perbandingan ini, serangkaian metrik evaluasi dihitung untuk mengukur performa model secara kuantitatif. Metrik evaluasi yang digunakan adalah Akurasi, Loss, AUC, Precision, Recall, F1-Score, EER, dan Confusion Matrix.

4.2.1 Lingkungan Implementasi

Implementasi dilakukan dalam lingkungan Google Colaboratory untuk memanfaatkan sumber daya komputasi berbasis cloud, terutama GPU, guna mempercepat proses pelatihan model. Rincian perangkat keras dan lunak yang digunakan dapat dilihat pada tabel 4.1:

Tabel 4.1. Spesifikasi Lingkungan Implementasi

Komponen	Spesifikasi
Perangkat Keras (Hardware)	
GPU	<i>T4 GPU</i>
RAM	<i>12 GB</i>
Perangkat Lunak (Software)	
Sistem Operasi	Linux (via Google Colab)
Bahasa	Python 3
Pustaka Utama	TensorFlow, Keras, Librosa, Scikit-learn NumPy, Matplotlib, OpenCV

4.2.2 Constraints

Berikut merupakan imports dan constraints yang dipakai dalam proses pembangunan model DCGANs untuk deteksi audio deeppoax.

```

1 # Constraints
2 SAMPLE_RATE = 22050
3 MAX_DURATION = 5
4 IMAGE_SIZE = 128
5 N_FFT = 2048
6 HOP_LENGTH = 512
7 N_MELS = 128
8 SEED = 42
9 INPUT_SHAPE = (IMAGE_SIZE, IMAGE_SIZE, 1)
10 BUFFER_SIZE = 5000
11 BATCH_SIZE = 64
12 EPOCHS = 50
13 LEARNING_RATE = 0.00005
14 BETA_1 = 0.5

```

Kode 4.1: Imports dan Constraints

1. **SAMPLE_RATE**: Menetapkan laju sampling audio ke 22050 Hz. Ini adalah standar umum dalam pemrosesan audio untuk *machine learning* yang menangkap rentang frekuensi ucapan manusia secara memadai.
2. **MAX_DURATION**: Menentukan bahwa setiap segmen audio akan diproses dalam durasi seragam, yaitu 5 detik, baik dengan cara memotong audio yang lebih panjang atau menambahkan keheningan pada audio yang lebih pendek.

3. `IMAGE_SIZE`: Mengatur resolusi gambar spektrogram menjadi 128x128 piksel. Ukuran ini memastikan semua input ke model memiliki dimensi yang konsisten.
4. `N_FFT`: Jumlah titik untuk *Fast Fourier Transform* (FFT). Parameter ini memengaruhi resolusi frekuensi dari spektrogram.
5. `HOP_LENGTH`: Jumlah sampel antara jendela FFT yang berurutan. Parameter ini memengaruhi resolusi waktu dari spektrogram.
6. `N_MELS`: Jumlah pita frekuensi Mel yang akan dibuat, yang secara efektif menentukan tinggi gambar spektrogram (128 piksel).
7. `SEED`: Nilai acak tetap (42) yang digunakan untuk memastikan bahwa setiap proses yang melibatkan keacakan (seperti pembagian data dan inisialisasi bobot) dapat direproduksi secara konsisten.
8. `INPUT_SHAPE`: Mendefinisikan bentuk tensor input untuk model, yaitu gambar 128x128 piksel dengan 1 kanal warna (skala abu-abu).
9. `BUFFER_SIZE`: Ukuran buffer yang digunakan untuk mengacak dataset, memastikan bahwa data disajikan ke model dalam urutan yang tidak terduga di setiap epoch.
10. `BATCH_SIZE`: Menentukan jumlah sampel (64) yang akan diproses oleh model dalam satu iterasi selama pelatihan.
11. `EPOCHS`: Jumlah maksimum siklus pelatihan penuh pada seluruh dataset. Pelatihan dapat berhenti lebih awal berkat *callback* `EarlyStopping`.
12. `LEARNING_RATE`: Laju pembelajaran awal untuk optimizer, yang mengontrol seberapa besar bobot model diperbarui pada setiap iterasi.
13. `BETA_1`: Hyperparameter untuk optimizer Adam, yang mengontrol laju peluruhan eksponensial untuk estimasi momen pertama.

4.2.3 Implementasi *Preprocess* Data

Pra-pemrosesan data adalah langkah kunci untuk mengubah data audio mentah menjadi format yang dapat diolah oleh model CNN [10]. Fungsi `load_and_preprocess_data_for_dcgan` dan `generate_spectrograms` dibuat

untuk mengotomatiskan proses ini, yang mencakup pemuatan, segmentasi, pembuatan spektrogram, normalisasi, dan augmentasi data.

A Loading data

Berikut merupakan implementasi loading data di bahasa Python:

```
1 def load_and_preprocess_data_for_dcgan(...):
2     # Construct the full paths for the real and fake audio
    directories.
3     real_dir = os.path.join(data_directory, 'real')
4     fake_dir = os.path.join(data_directory, 'fake')
5
6     # Check if the directories exist to prevent errors.
7     if not os.path.exists(real_dir):
8         print(f"Error: Directory not found: {real_dir}")
9         return
10    if not os.path.exists(fake_dir):
11        print(f"Error: Directory not found: {fake_dir}")
12        return
13
14    # Create a list of full file paths for all .wav and .mp3 files
    in the real directory.
15    real_files = [os.path.join(real_dir, f) for f in os.listdir(
    real_dir) if f.endswith(('.wav', '.mp3'))]
16
17    # Do the same for the fake directory.
18    fake_files = [os.path.join(fake_dir, f) for f in os.listdir(
    fake_dir) if f.endswith(('.wav', '.mp3'))]
19
20    # Check if any audio files were actually found.
21    if not real_files and not fake_files:
22        print("Error: No audio files found.")
23        return None, None
```

Kode 4.2: Loading Data Audio

Berdasarkan kode 4.2, tahap pertama yang dilakukan ialah membangun jalur atau direktori file audio yang akan digunakan dalam pembangunan model, lalu dilakukannya validasi mengenai ada atau tidaknya direktori tersebut. Setelah dibangun jalur direktorinya, akan dibangunnya jalur untuk memuat semua file audio di direktori audio asli dan hoax, dan diberlakukannya validasi keberadaan file agar mencegah terjadinya error.

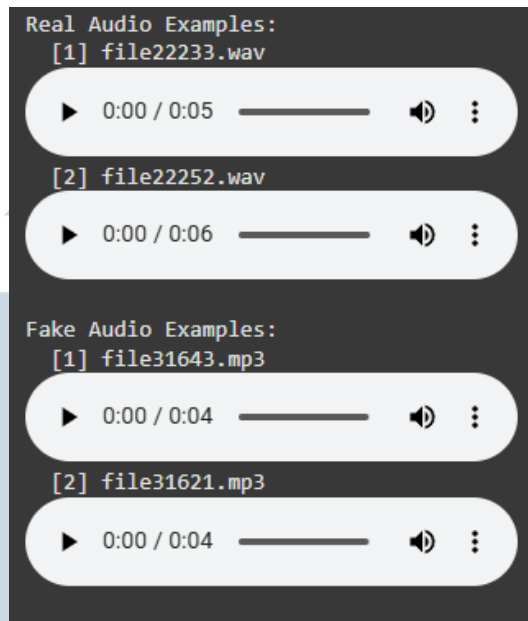
B *Splitting Data*

Setelah loading data berhasil dilakukan, maka akan diterapkan *splitting* data pada tahap *preprocess* dengan implementasi sebagai berikut:

```
1 # This code is inside the load_and_preprocess_data_for_dcgan
  function.
2
3 # Combine the lists of real and fake files into a single list.
4 all_files = real_files + fake_files
5
6 # Create a corresponding list of labels: 0 for real, 1 for fake.
7 labels = [0] * len(real_files) + [1] * len(fake_files)
8
9 # Use scikit-learn's function to split the data into training and
  testing sets.
10 train_files, test_files, train_labels, test_labels =
    train_test_split(
11     all_files,          # The list of all file paths to be split.
12     labels,             # The corresponding labels for each file.
13     test_size=test_size, # The proportion for the test set (
e.g., 0.2 for 20%).
14     random_state=random_state, # Seed for reproducibility.
15     stratify=labels        # Ensures train/test sets have the same
    class ratio.
16 )
```

Kode 4.3: Splitting Data

Berdasarkan kode 4.3, tahap pertama yang dilakukan ialah mengkombinasi list dari audio asli dan hoaks menjadi satu list gabungan yang dimana diberikan label (0) untuk audio asli dan (1) untuk audio hoaks. Setelah itu, digunakannya fungsi *splitting* data dari skicit-learn untuk memisahkan data tes dan data train, serta label tes dan label train.



Gambar 4.1. Hasil *splitting* data

Dapat dilihat pada gambar 4.1 bahwa audio berhasil dilabelkan menjadi *real* (0) dan *fake* (1).

C *Cleaning & Segmenting Data*

Setelah *splitting* data berhasil dilakukan, maka akan diterapkan *cleaning & segmenting* data pada tahap *preprocess* dengan tujuan untuk menyeragamkan durasi setiap data audio. Dalam tahap ini, ada beberapa langkah yang dilakukan:

1. Memeriksa durasi setiap file audio.
2. Jika durasi audio lebih pendek dari 5 detik, audio akan diberi tambahan keheningan di akhir (*padding*) hingga mencapai durasi 5 detik.
3. Jika durasi audio lebih panjang dari 5 detik, audio akan dipotong menjadi beberapa segmen berdurasi 5 detik yang saling tumpang tindih (*overlapping*).

Langkah-langkah di atas kemudian diimplementasikan dalam Notebook Python. *Code snippet* di bawah ini merupakan implementasi dari tahap *audio cleaning & segmenting*.

```
1 # This code is inside the generate_spectrograms function.
2
3 # Get the duration of the loaded audio file.
```

```

4 audio_duration = librosa.get_duration(y=y, sr=sr)
5
6 # Check if the audio's duration is within the 5-second limit.
7 if audio_duration <= max_duration:
8     # If the audio is shorter than the max duration...
9     if audio_duration < max_duration:
10         # ...pad the end of the audio with silence to make it
11         # exactly 5 seconds long.
12         y = librosa.util.fix_length(y, size=int(max_duration * sr)
13         )
14 # If the audio is longer than the max duration...
15 else:
16     # Check if the instruction is to split the audio.
17     if split_audio:
18         # Define a hop size to create overlapping segments.
19         # Here it's a 50% overlap of the max duration.
20         segment_length_samples = int(max_duration * sr)
21         hop_samples = int(segment_length_samples * 0.5)
22
23         # Loop through the audio signal, creating 5-second
24         segments.
25         for i in range(0, len(y) - segment_length_samples + 1,
26         hop_samples):
27             segment = y[i:i + int(max_duration * sr)]
28             # ... spectrogram processing happens for each segment
29             here ...

```

Kode 4.4: Audio Cleaning dan Segmenting

Berdasarkan kode 4.4, fungsi pertama-tama memeriksa durasi dari file audio yang telah dimuat. Jika durasinya kurang dari atau sama dengan 5 detik, audio yang lebih pendek akan diperpanjang dengan keheningan (*padding*). Namun, jika audio lebih panjang, skrip akan memotongnya menjadi beberapa segmen 5 detik yang saling tumpang tindih (*overlapping*). Setelah audio distandarisasi, langkah selanjutnya adalah generasi spektrogram.

D Generasi Spektrogram

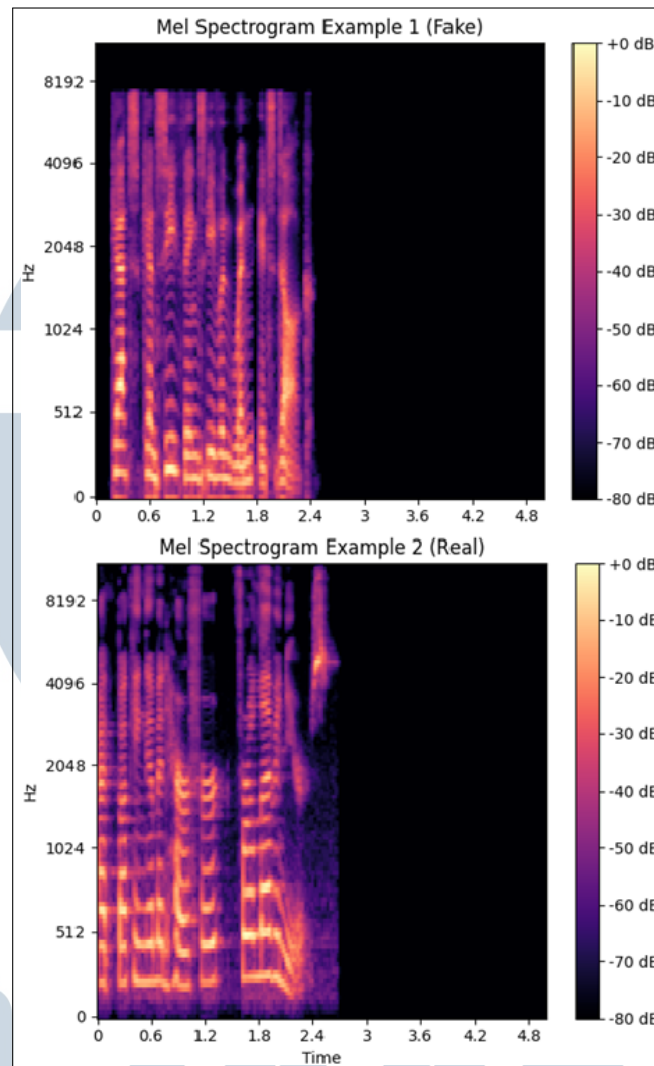
Setelah audio distandarisasi, langkah selanjutnya adalah mengubah data audio dari domain waktu menjadi domain frekuensi-waktu dalam bentuk gambar. Proses ini mencakup:

1. Mengubah setiap segmen audio 5 detik menjadi Mel-spektrogram menggunakan library Librosa.
2. Proses ini menggunakan parameter `n_fft=2048`, `hop_length=512`, dan `n_mels=128`.
3. Mengonversi nilai daya (power) dari spektrogram ke skala desibel (dB) untuk menyesuaikan dengan persepsi pendengaran manusia.

```
1 # This code is inside the generate_spectrograms function for each
  segment.
2
3 # Create a Mel-spectrogram from the audio segment.
4 spectrogram = librosa.feature.melspectrogram(
5     y=y,
6     sr=sr,
7     n_fft=N_FFT,
8     hop_length=HOP_LENGTH,
9     n_mels=N_MELS,
10    fmin=20,
11    fmax=8000
12 )
13
14 # Convert the power spectrogram to a decibel (dB) scale.
15 spectrogram_db = librosa.power_to_db(
16     spectrogram,
17     ref=np.max
18 )
```

Kode 4.5: Generasi Spektrogram

Berdasarkan kode 4.5, setiap segmen audio yang telah distandarisasi durasinya diubah menjadi Mel-spektrogram. Proses ini menggunakan fungsi `librosa.feature.melspectrogram` dengan parameter kunci seperti `n_fft` untuk ukuran jendela FFT dan `n_mels` untuk jumlah pita frekuensi Mel. Selanjutnya, `librosa.power_to_db` mengonversi skala daya spektrogram menjadi skala desibel (dB), yang lebih sesuai dengan cara manusia mempersepsikan kenyaringan suara. Gambar 4.2 merupakan hasil dari fungsi generasi spektrogram yang memiliki label asli (0) dan hoaks (1).



Gambar 4.2. Hasil dari Fungsi Generasi Spectrogram

Setelah spektrogram berhasil dibuat, akan dilakukannya normalisasi data terhadap spektrogram audio.

E Normalisasi Data

Spektrogram yang dihasilkan kemudian diproses lebih lanjut agar menjadi input yang ideal untuk model. Langkah-langkahnya adalah:

1. Setiap spektrogram diubah ukurannya menjadi gambar persegi 128x128 piksel.
2. Nilai piksel gambar dinormalisasi dan diskalakan ke rentang nilai $[-1, 1]$.

3. Dimensi "channel" ditambahkan pada setiap gambar, mengubah bentuknya menjadi (128, 128, 1) agar sesuai dengan input lapisan konvolusi.

```
1 # This code is inside the generate_spectrograms function.
2
3 # Resize the spectrogram to a standard image size (128x128).
4 img = cv2.resize(spectrogram_db, (image_size, image_size),
5                  interpolation=cv2.INTER_AREA)
6
7 # Standardize the image (z-score).
8 img = (img - np.mean(img)) / np.std(img)
9
10 # Shift and scale the values. This aims to bring the typical
11 # dB range (e.g., -80 to 0) into an approximate [0, 1] range.
12 img = (img + 80) / 80
13
14 # Scale the values to the final [-1, 1] range for the model.
15 img = (img * 2) - 1
```

Kode 4.6: Normalisasi dalam Fungsi generate_spectrograms

Berdasarkan kode 4.6, spektrogram dalam skala dB yang dihasilkan kemudian diproses seperti gambar. Pertama, ukurannya diubah menjadi 128x128 piksel. Setelah itu, dilakukan normalisasi multi-tahap untuk menstabilkan input yang diterima oleh model. Terakhir, setelah semua spektrogram terkumpul, dimensi "channel" akan ditambahkan.

```
1 # This code is located in the main script, after collecting all
  spectrograms.
2
3 # Add a channel dimension for the CNN input.
4 # The shape changes from (num_samples, 128, 128) to (num_samples,
  128, 128, 1).
5 train_data = train_data[..., tf.newaxis]
6 test_data = test_data[..., tf.newaxis]
```

Kode 4.7: Penambahan Channel Dimension

Berdasarkan kode 4.7, setelah semua spektrogram dari semua file audio dikumpulkan menjadi sebuah array NumPy, sebuah dimensi 'channel' ditambahkan di akhir. Proses ini mengubah bentuk array agar sesuai dengan format input yang dibutuhkan oleh lapisan konvolusi 2D.

4.2.4 Implementasi Konfigurasi Model DCGANs

Tahap ini mencakup perancangan arsitektur, kompilasi, dan proses pelatihan model.

A Pembangunan Arsitektur Model

Arsitektur yang digunakan adalah model Diskriminator dari DCGANs yang dimodifikasi untuk tugas klasifikasi audio dalam format gambar spektogram.

1. Model dibangun menggunakan `tf.keras.Sequential`.
2. Terdiri dari beberapa blok lapisan `Conv2D` untuk ekstraksi fitur, dengan `LeakyReLU` sebagai fungsi aktivasi.
3. Lapisan `BatchNormalization` digunakan untuk menstabilkan pelatihan.
4. Lapisan `Dropout` dan regularisasi `L2` ditambahkan untuk mencegah *overfitting*.
5. Lapisan `Flatten` dan `Dense` dengan aktivasi `sigmoid` di akhir berfungsi sebagai output klasifikasi biner.

```
1 def def make_discriminator(input_shape_tuple):
2     model = tf.keras.Sequential(name="
discriminator_classifier_l2_specaug")
3     model.add(Input(shape=input_shape_tuple))
4
5     # === CONVOLUTIONAL LAYER 1 ===
6     # Takes the 128x128x1 input and downsamples it to 64x64x32
7     model.add(layers.Conv2D(32, (4, 4), strides=(2, 2), padding='
same', kernel_regularizer=regularizers.l2(L2_LAMBDA)))
8     model.add(layers.LeakyReLU(negative_slope=0.2))
9     model.add(layers.Dropout(0.4))
10
11    # === CONVOLUTIONAL LAYER 2 ===
12    # Takes the 64x64x32 input and downsamples it to 32x32x64
13    model.add(layers.Conv2D(64, (4, 4), strides=(2, 2), padding='
same', kernel_regularizer=regularizers.l2(L2_LAMBDA)))
14    model.add(layers.BatchNormalization())
15    model.add(layers.LeakyReLU(negative_slope=0.2))
16    model.add(layers.Dropout(0.4))
```

```

17
18     # === CONVOLUTIONAL LAYER 3 ===
19     # Takes the 32x32x64 input and downsamples it to 16x16x128
20     model.add(layers.Conv2D(128, (4, 4), strides=(2, 2), padding='
same', kernel_regularizer=regularizers.l2(L2_LAMBDA)))
21     model.add(layers.BatchNormalization())
22     model.add(layers.LeakyReLU(negative_slope=0.2))
23     model.add(layers.Dropout(0.4))
24
25     # === CONVOLUTIONAL LAYER 4 ===
26     # Takes the 16x16x128 input and downsamples it to 8x8x256
27     model.add(layers.Conv2D(256, (4, 4), strides=(2, 2), padding='
same', kernel_regularizer=regularizers.l2(L2_LAMBDA)))
28     model.add(layers.BatchNormalization())
29     model.add(layers.LeakyReLU(negative_slope=0.2))
30     model.add(layers.Dropout(0.4))
31
32     # Flatten the final feature map to a 1D vector before the
final classification.
33     model.add(layers.Flatten())
34
35     # Final dense layer for classification output.
36     model.add(layers.Dense(1, activation='sigmoid',
kernel_regularizer=regularizers.l2(L2_LAMBDA)))
37
38     return model

```

Kode 4.8: Arsitektur Model Klasifikasi

Berdasarkan kode 4.8, arsitektur model klasifikasi dibangun menggunakan Keras Sequential API. Model ini diawali dengan lapisan Input yang mendefinisikan bentuk data masukan. Arsitekturnya terdiri dari empat blok konvolusi yang secara bertahap mengurangi dimensi spasial gambar sambil menambah kedalaman fitur. Setiap blok menggunakan lapisan Conv2D untuk ekstraksi fitur, LeakyReLU sebagai fungsi aktivasi, Dropout untuk regularisasi, dan BatchNormalization untuk stabilitas. Di akhir, fitur yang telah diekstraksi diratakan (*flatten*) dan dimasukkan ke lapisan Dense dengan satu neuron dan aktivasi sigmoid untuk menghasilkan probabilitas klasifikasi biner.

B Kompilasi Model

Sebelum dilatih, model perlu dikompilasi dengan mendefinisikan optimizer, fungsi loss, dan metrik.

1. **Optimizer:** AdamW digunakan, yang merupakan varian dari Adam dengan *weight decay*.
2. **Loss Function:** BinaryCrossentropy dipilih karena ini adalah masalah klasifikasi biner.
3. **Metrics:** BinaryAccuracy dan AUC digunakan untuk memantau performa model.

```
1 # Define the optimizer with a learning rate and weight decay.
2 optimizer = tf.keras.optimizers.AdamW(
3     learning_rate=LEARNING_RATE,
4     weight_decay=0.004
5 )
6
7 # Compile the model, linking the optimizer, loss, and metrics.
8 classifier_model.compile(
9     optimizer=optimizer,
10    loss='binary_crossentropy',
11    metrics=['bin_accuracy', 'auc']
12 )
```

Kode 4.9: Kompilasi Model

Berdasarkan kode 4.9, model dikonfigurasi untuk proses pelatihan. AdamW dipilih sebagai *optimizer* karena kemampuannya dalam menerapkan *weight decay* secara efektif untuk regularisasi. Fungsi *loss* yang digunakan adalah BinaryCrossentropy, yang cocok untuk masalah klasifikasi dengan dua kelas. Selama pelatihan, dua metrik utama dipantau: `bin-accuracy` untuk akurasi dan `auc` untuk mengukur kemampuan diskriminatif model.

C Callback Setup

Beberapa *callback* digunakan untuk mengoptimalkan proses pelatihan.

1. **ModelCheckpoint:** Menyimpan model dengan performa terbaik.
2. **EarlyStopping:** Menghentikan pelatihan jika performa tidak lagi meningkat.

3. ReduceLROnPlateau: Mengurangi *learning rate* secara dinamis saat performa stagnan.

```
1 # Saves the best version of the model based on validation loss.
2 checkpoint_callback = ModelCheckpoint(
3     monitor='val_loss', mode='min', save_best_only=True, ...)
4
5 # Stops training if validation loss does not improve for 15 epochs
6
7 early_stopping_callback = EarlyStopping(
8     monitor='val_loss', patience=15, restore_best_weights=True,
9     ...)
10
11 # Reduces learning rate if validation loss plateaus.
12 reduce_lr_callback = ReduceLROnPlateau(
13     monitor='val_loss', factor=0.2, patience=5, ...)
```

Kode 4.10: Callback Setup

Berdasarkan kode 4.10, tiga *callback* utama disiapkan. `ModelCheckpoint` bertugas menyimpan versi model terbaik setiap kali nilai `val_loss` mencapai rekor terendah. `EarlyStopping` memantau `val_loss` dan akan menghentikan pelatihan jika tidak ada perbaikan selama 15 epoch, sekaligus mengembalikan bobot model ke versi terbaiknya. Terakhir, `ReduceLROnPlateau` akan secara otomatis menurunkan *learning rate* jika `val_loss` tidak kunjung membaik.



```

Epoch 14/50
446/446 ————— 0s 21ms/step - auc: 0.9334 - bin_accuracy: 0.8752 - loss: 0.3436
Epoch 14: val_loss did not improve from 0.35589

Epoch 14: ReduceLROnPlateau reducing learning rate to 1.9999999494757505e-06.
446/446 ————— 11s 25ms/step - auc: 0.9334 - bin_accuracy: 0.8752 - loss: 0.3436
Epoch 15/50
446/446 ————— 0s 21ms/step - auc: 0.9325 - bin_accuracy: 0.8770 - loss: 0.3460
Epoch 15: val_loss did not improve from 0.35589
446/446 ————— 10s 23ms/step - auc: 0.9325 - bin_accuracy: 0.8770 - loss: 0.3460
Epoch 16/50
446/446 ————— 0s 21ms/step - auc: 0.9339 - bin_accuracy: 0.8747 - loss: 0.3432
Epoch 16: val_loss did not improve from 0.35589
446/446 ————— 21s 25ms/step - auc: 0.9339 - bin_accuracy: 0.8747 - loss: 0.3432
Epoch 17/50
446/446 ————— 0s 21ms/step - auc: 0.9339 - bin_accuracy: 0.8735 - loss: 0.3428
Epoch 17: val_loss did not improve from 0.35589
446/446 ————— 10s 23ms/step - auc: 0.9339 - bin_accuracy: 0.8735 - loss: 0.3428
Epoch 18/50
444/446 ————— 0s 21ms/step - auc: 0.9338 - bin_accuracy: 0.8742 - loss: 0.3434
Epoch 18: val_loss did not improve from 0.35589
446/446 ————— 10s 23ms/step - auc: 0.9338 - bin_accuracy: 0.8742 - loss: 0.3434
Epoch 19/50
445/446 ————— 0s 22ms/step - auc: 0.9337 - bin_accuracy: 0.8746 - loss: 0.3440
Epoch 19: val_loss did not improve from 0.35589

Epoch 19: ReduceLROnPlateau reducing learning rate to 3.999999989900971e-07.
446/446 ————— 21s 25ms/step - auc: 0.9337 - bin_accuracy: 0.8746 - loss: 0.3440
Epoch 19: early stopping
Restoring model weights from the end of the best epoch: 4.
Training finished.

```

Gambar 4.3. *Output callback setup*

Penerapan langsung dari *callback setup* dapat dilihat pada gambar 4.3 yang menampilkan bahwa walaupun max epoch nya 50, dikarenakan `val_loss` tidak menurun dari epoch ke 4 sampai epoch ke 19 maka proses training akan stop dan akan menyimpan dan menggunakan hasil model dengan `val_loss` terendah dan juga dikarenakan `val_loss` tidak menurun, `ReduceLROnPlateau` akan berjalan untuk menurunkan *learning rate*.

D Pelatihan Model

Model dilatih menggunakan metode `fit` pada *train* data yang telah disiapkan.

1. Proses pelatihan berjalan untuk jumlah `EPOCHS` yang telah ditentukan sebagai batas maksimal.
2. Data validasi (set *test* data) digunakan di setiap akhir epoch untuk mengevaluasi generalisasi model.
3. Semua *callback* diaktifkan selama proses pelatihan.


```

1 # Start the training process.
2 history = classifier_model.fit(
3     train_dataset,                # The training data generator.
4     epochs=EPOCHS,               # The maximum number of epochs.
5     validation_data=test_dataset, # Data for evaluation.
6     callbacks=[checkpoint_callback,
7                 early_stopping_callback,
8                 reduce_lr_callback] # List of callbacks.
9 )

```

Kode 4.11: Pelatihan Model

Berdasarkan kode 4.11, proses pelatihan model dimulai dengan memanggil fungsi `fit`. Fungsi ini menggunakan `train_dataset` sebagai *train* data dan `test_dataset` sebagai data validasi. Jumlah maksimal epoch diatur oleh variabel `EPOCHS`, namun prosesnya akan berhenti lebih awal jika dipicu oleh *callback* `EarlyStopping`. Semua *callback* yang telah didefinisikan sebelumnya diteruskan untuk mengelola proses pelatihan secara otomatis.

4.3 Testing dan Evaluasi

Setelah model selesai dilatih, performanya diuji secara menyeluruh pada set *test* data.

A Pengujian Model

Model dengan bobot terbaik dievaluasi pada *test* data untuk mendapatkan metrik performa kuantitatif.

1. Metode `evaluate` digunakan untuk menghitung nilai *loss*, akurasi, dan AUC secara keseluruhan pada *test* data.
2. Metode `predict` digunakan untuk mendapatkan probabilitas prediksi dari model untuk setiap sampel *test* data.

```

1 # Evaluate the model on the test dataset to get loss, accuracy,
  and AUC.
2 eval_results = classifier_model.evaluate(test_dataset)
3
4 # Get the raw prediction probabilities for each sample in the test
  set.

```



```
s y_pred_proba = classifier_model.predict(test_dataset)
```

Kode 4.12: Pengujian Model

Berdasarkan kode 4.12, pengujian model dilakukan dalam dua langkah utama. Pertama, metode `evaluate` dipanggil pada `test_dataset` untuk mendapatkan nilai-nilai metrik agregat yang telah dikompilasi. Kedua, metode `predict` digunakan pada dataset yang sama untuk menghasilkan larik probabilitas mentah untuk setiap sampel uji. Hasil probabilitas ini kemudian akan digunakan untuk analisis yang lebih mendalam.

B Analisis Hasil

Hasil prediksi kemudian dianalisis lebih dalam untuk mendapatkan metrik evaluasi yang komprehensif.

1. Probabilitas prediksi diubah menjadi label kelas (0 atau 1) menggunakan ambang batas (threshold) 0.5.
2. Classification Report: Dihasilkan untuk melihat nilai *Precision*, *Recall*, dan *F1-Score* untuk setiap kelas.
3. Equal Error Rate (EER): Dihitung dengan mencari titik di mana *False Positive Rate* (FPR) dan *False Negative Rate* (FNR) memiliki nilai yang sama.

```
1 # Import scikit-learn metrics functions.
2 from sklearn.metrics import classification_report, roc_curve
3
4 # Convert probabilities to binary class predictions (0 or 1).
5 y_pred_class = (y_pred_proba > 0.5).astype(int)
6
7 # 1. Generate and print a detailed report.
8 print(classification_report(y_true, y_pred_class,
9                             target_names=['Real (0)', 'Fake (1)'])
10 )
11
12 # 2. Calculate the values needed for the ROC curve.
13 fpr, tpr, thresholds = roc_curve(y_true, y_pred_proba)
14 fnr = 1 - tpr
15
16 # Find the point where FPR and FNR are closest (EER).
17 eer_index = np.nanargmin(np.abs(fnr - fpr))
```

```
17 eer = fpr[eer_index]
```

Kode 4.13: Analisis Hasil Metrik

Berdasarkan kode 4.13, analisis hasil dimulai dengan mengubah output probabilitas dari model menjadi prediksi kelas biner (0 atau 1). Kemudian, `classification_report` dari Scikit-learn digunakan untuk menghasilkan laporan terperinci yang mencakup metrik *precision*, *recall*, dan *f1-score*. Selain itu, Tingkat Kesalahan Setara (EER) dihitung dengan terlebih dahulu mendapatkan kurva ROC, lalu mencari titik di mana tingkat kesalahan positif palsu (FPR) dan negatif palsu (FNR) paling mendekati satu sama lain.

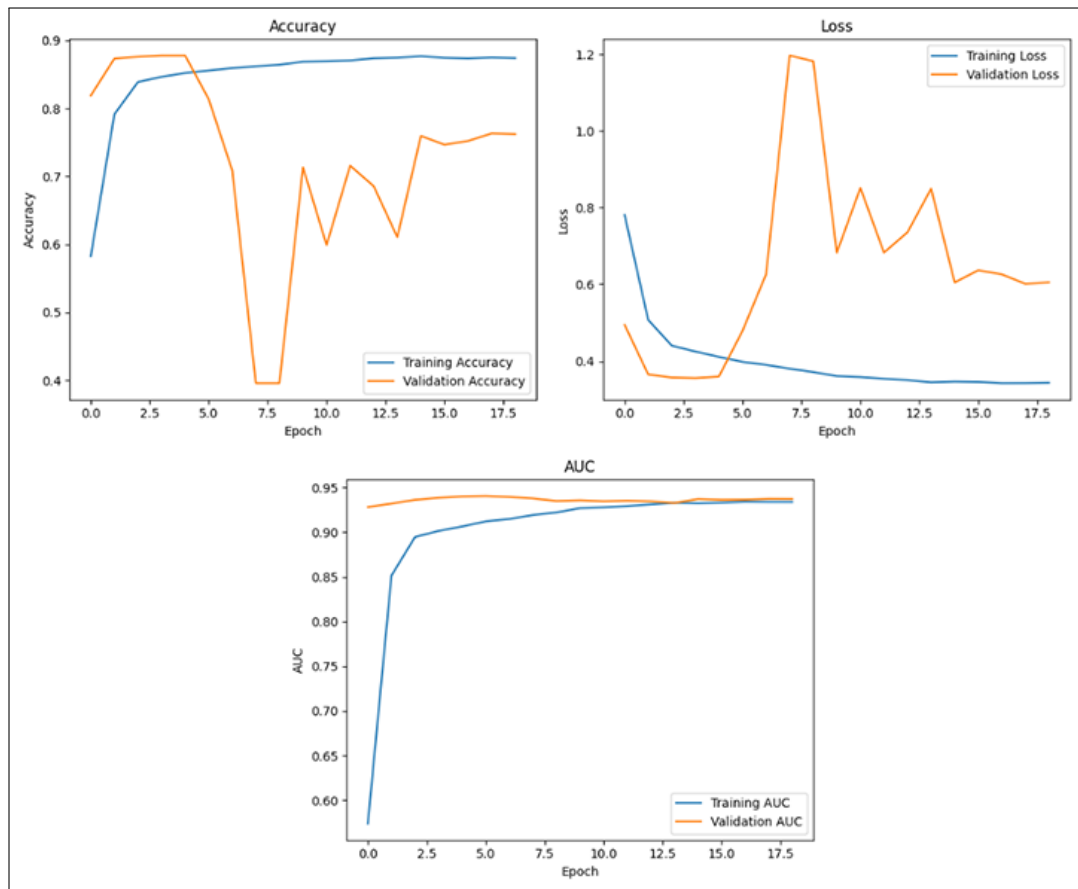
4.4 Hasil

Bagian ini menyajikan dan menganalisis hasil dari proses pelatihan dan pengujian model.

A Hasil Pelatihan Model

Selama proses pelatihan, metrik performa pada *train* data dan data validasi dipantau pada setiap epoch. Grafik berikut menunjukkan histori pelatihan model.





Gambar 4.4. Grafik Akurasi, Loss, dan AUC Selama Pelatihan

Berdasarkan Gambar 4.4, dapat dianalisis dinamika kompleks selama proses pelatihan. Pada sekitar 5 hingga 7 epoch pertama, model menunjukkan proses pembelajaran yang sehat, di mana kurva akurasi dan AUC pada data validasi (oranye) meningkat sejalan dengan data pelatihan (biru), sementara kurva loss menurun. Namun, setelah epoch ke-7, terjadi divergensi yang jelas: metrik pada data pelatihan terus membaik, tetapi performa pada data validasi memburuk secara signifikan—terlihat dari lonjakan tajam pada kurva *validation loss* dan penurunan drastis pada *validation accuracy*.

Fenomena ini adalah indikator kuat dari *overfitting*, di mana model mulai ”menghafal” data pelatihan alih-alih mempelajari fitur yang dapat digeneralisasi. Di sinilah peran krusial dari *callback* `ModelCheckpoint` dan `EarlyStopping` menjadi sangat penting. Meskipun pelatihan dilanjutkan, model terbaik yang disimpan dan digunakan untuk evaluasi akhir adalah model dari epoch sebelum terjadinya *overfitting* parah, sehingga memastikan versi model yang paling optimal dan general yang digunakan.

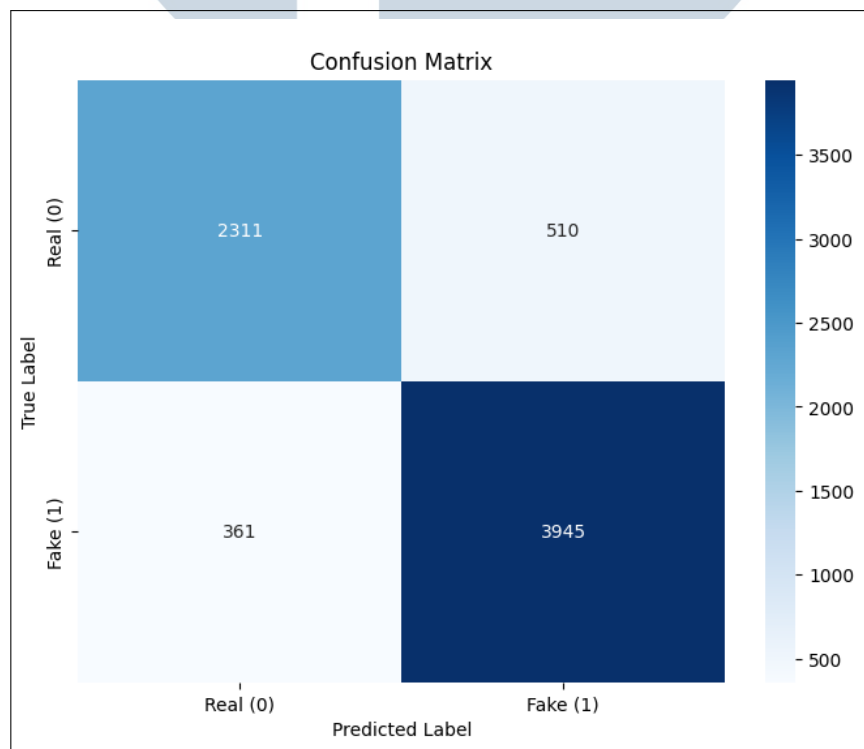
B Hasil Pengujian Model

Setelah pelatihan selesai, model dengan performa terbaik dievaluasi pada data tes untuk mendapatkan metrik performa akhir.

Tabel 4.2. Hasil Metrik Evaluasi pada Data Tes

Metrik	Nilai
Test Loss	0.3559
Test Accuracy	0.8778 (87.8%)
Test AUC	0.9386
Equal Error Rate (EER)	0.1329 (13.3%)

Meskipun metrik pada tabel 4.2 memberikan gambaran umum, penting untuk memahami secara spesifik bagaimana model menangani setiap kelas. Oleh karena itu, *confusion matrix* (Gambar 4.5) dihasilkan untuk menganalisis distribusi prediksi yang benar dan yang salah.



Gambar 4.5. Visualisasi Confusion Matrix pada Data Uji

Berdasarkan Gambar 4.5, disajikan visualisasi performa model klasifikasi pada data uji. Matriks ini membandingkan antara label asli (*True Label*) dengan

label yang diprediksi oleh model (*Predicted Label*) dan membaginya ke dalam empat kuadran:

1. True Negative (TN): Terdapat 2311 sampel audio asli yang berhasil diprediksi dengan benar sebagai *real*.
2. False Positive (FP): Terdapat 510 sampel audio asli yang salah diklasifikasikan sebagai *fake*.
3. False Negative (FN): Terdapat 361 sampel audio *fake* yang gagal dideteksi dan salah diklasifikasikan sebagai *real*.
4. True Positive (TP): Terdapat 3945 sampel audio *fake* yang berhasil diidentifikasi dengan benar sebagai *fake*.

Untuk memverifikasi metrik yang dilaporkan, perhitungan manual dapat dilakukan berdasarkan nilai dari *confusion matrix*:

1. Akurasi: Mengukur total prediksi yang benar dari keseluruhan data.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{3945 + 2311}{3945 + 2311 + 510 + 361} \approx 0.87 \quad (4.1)$$

2. Precision (untuk kelas *fake*): Dari semua yang diprediksi *fake*, mengukur berapa persen yang benar-benar *fake*.

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{3945}{3945 + 510} = \frac{3945}{4455} \approx 0.8855 \quad (4.2)$$

3. Recall (untuk kelas *fake*): Dari semua audio *fake* yang ada, mengukur berapa persen yang berhasil dideteksi.

$$\text{Recall} = \frac{TP}{TP + FN} = \frac{3945}{3945 + 361} = \frac{3945}{4306} \approx 0.9161 \quad (4.3)$$

Perhitungan ini mengonfirmasi hasil yang dilaporkan dalam *classification report*, menunjukkan bahwa model memiliki performa yang kuat. Secara total, model membuat 6,256 prediksi yang benar (TP+TN) dan 871 prediksi yang salah (FP+FN), yang menunjukkan kemampuan generalisasi yang baik pada data uji.

4.5 Diskusi

Bagian ini menyajikan pembahasan mendalam mengenai hasil yang diperoleh dari proses implementasi dan pengujian model. Diskusi dibagi menjadi dua bagian utama: analisis dampak dari proses *fine-tuning* hyperparameter terhadap performa model, dan analisis performa klasifikasi dari model akhir yang telah dioptimalkan.

4.5.1 Dampak Konfigurasi dan *Tuning* Hyperparameter

Proses untuk mencapai model dengan performa optimal melibatkan serangkaian iterasi dan *fine-tuning* hyperparameter. Model awal yang dibangun, meskipun menggunakan arsitektur DCGAN yang solid, menunjukkan beberapa tantangan utama, yaitu *overfitting* yang parah dan ketidakstabilan selama pelatihan, dengan akurasi awal yang rendah. Grafik pelatihan awal menunjukkan divergensi yang jelas antara *training loss* yang terus menurun dan *validation loss* yang meningkat secara drastis, sebuah indikator klasik dari *overfitting*.

Langkah pertama untuk mengatasi ini adalah dengan memperbesar jumlah dataset secara signifikan dari sekitar 200 audio menjadi 1000 audio lalu terakhir menjadi 33445 audio. Meskipun penambahan data ini memberikan fondasi yang lebih baik, masalah *overfitting* masih tetap ada. Oleh karena itu, beberapa strategi *tuning* diterapkan secara bertahap:

1. Regularisasi: Teknik regularisasi yang kuat menjadi fokus utama dalam *hyperparameter tuning* untuk mengatasi *overfitting*. Salah satu teknik yang paling berpengaruh adalah implementasi lapisan Dropout. Dropout bekerja dengan cara menonaktifkan sebagian neuron secara acak pada setiap iterasi pelatihan. Mekanisme ini memaksa jaringan untuk tidak terlalu bergantung pada neuron tunggal atau jalur fitur tertentu, sehingga mendorongnya untuk belajar fitur-fitur yang lebih general dan esensial [56]. Pada iterasi awal, model dibangun tanpa lapisan Dropout, yang menjadi salah satu penyebab utama terjadinya *overfitting* yang signifikan. Sebagai langkah perbaikan, lapisan Dropout kemudian ditambahkan dengan laju 0.3. Namun, karena *overfitting* masih terdeteksi setelah penambahan data, laju dropout kemudian ditingkatkan menjadi 0.4. Peningkatan ini memberikan tingkat regularisasi yang lebih kuat dan terbukti efektif dalam menstabilkan *validation loss* serta

meningkatkan kemampuan generalisasi model secara signifikan, seperti yang terlihat pada hasil pelatihan akhir.

2. Optimizer dan Learning Rate: Eksperimen dilakukan dengan *optimizer* dan *learning rate*. Model diuji menggunakan optimizer AdamW yang menggabungkan *weight decay* untuk regularisasi tambahan. *Learning rate* awal yang lebih rendah (0.00005) dicoba, namun langkah paling signifikan adalah implementasi *callback* ReduceLROnPlateau. Callback ini memungkinkan penggunaan *learning rate* awal yang cukup, lalu menurunkannya secara otomatis ketika *validation loss* tidak lagi membaik. Hal ini terbukti sangat efektif dalam menstabilkan proses konvergensi di tahap akhir pelatihan.

Kombinasi dari penambahan data, regularisasi berlapis (Dropout dan L2), dan strategi *learning rate* yang adaptif terbukti berhasil mengubah model dari yang tidak stabil dan overfit menjadi model yang robust dengan performa yang sangat baik.

A Kesimpulan Cara Mengatasi *Overfitting*

Berdasarkan serangkaian iterasi yang telah dilakukan, dapat disimpulkan bahwa tidak ada satu strategi tunggal yang dapat mengatasi masalah *overfitting* secara efektif. Solusi optimal dicapai melalui kombinasi dari tiga pendekatan utama: memperbesar volume dan variasi data, menerapkan teknik regularisasi yang kuat seperti Dropout dan L2 *weight decay* dari optimizer AdamW, serta mengelola laju pembelajaran (*learning rate*) secara dinamis menggunakan *callback*. Tabel ?? merangkum proses *tuning* yang dilakukan beserta dampaknya.

1. Penambahan Data: Menghadapi masalah *overfitting* parah pada dataset awal yang kecil, strategi pertama adalah meningkatkan volume data secara signifikan hingga mencapai 33445 sampel audio. Langkah ini berhasil memberikan fondasi yang lebih baik untuk pelatihan, meskipun belum sepenuhnya menghilangkan masalah *overfitting*.
2. Implementasi Regularisasi Dropout: Meskipun data telah ditambah, *overfitting* masih terdeteksi. Untuk mengatasinya, teknik regularisasi Dropout diterapkan dengan laju yang ditingkatkan dari 0.3 menjadi 0.4. Hal ini terbukti menjadi langkah yang paling berpengaruh, yang secara signifikan berhasil menstabilkan *validation loss* dan meningkatkan kemampuan generalisasi model.

3. Optimisasi Pelatihan: Untuk mengatasi konvergensi yang lambat dan tidak stabil pada tahap akhir, *optimizer* AdamW digunakan untuk regularisasi *weight decay* yang lebih baik. Selain itu, *callback* ReduceLROnPlateau diimplementasikan untuk menurunkan *learning rate* secara dinamis. Kombinasi ini efektif dalam menstabilkan proses pelatihan dan membantu model mencapai konvergensi yang optimal.

4.5.2 Analisis Performa Diskriminator

Hasil pembangunan model DCGANs untuk membedakan audio asli dan hoaks menunjukkan kemampuan generalisasi yang baik. Hal ini dapat dilihat dari hasil akhir *evaluation metrics* pada tabel 4.3:

Tabel 4.3. Tabel Hasil Metrik Klasifikasi

Kelas	Precision	Recall	F1-Score	Support
Real (0)	0.86	0.82	0.84	2821
Fake (1)	0.89	0.92	0.90	4306

Model akhir yang dihasilkan dari serangkaian fine-tuning menunjukkan kemampuan deteksi yang kuat dan seimbang. Berdasarkan grafik histori pelatihan, terlihat bahwa model mengalami *overfitting* pada epoch-epoch selanjutnya. Namun, fenomena ini berhasil diatasi secara efektif melalui penerapan *callback* EarlyStopping. Proses pelatihan berhenti secara otomatis pada epoch ke-19, dan mekanisme *restore_best_weights* memastikan bahwa bobot model yang digunakan untuk evaluasi akhir adalah dari epoch ke-4, yaitu epoch dengan performa terbaik pada data validasi sebelum *overfitting* terjadi.

Pada saat diuji pada data yang belum pernah dilihat sebelumnya, model ini mencapai akurasi akhir sebesar 87.8% dan AUC sebesar 0.9386. Nilai AUC yang tinggi ini menunjukkan bahwa model memiliki kemampuan diskriminatif yang sangat baik dalam membedakan antara kelas audio asli dan palsu.

Analisis mendalam terhadap metrik klasifikasi mengungkapkan performa model yang tidak hanya tinggi tetapi juga seimbang. Model menunjukkan Recall sebesar 92% untuk kelas *fake*, yang berarti ia sangat efektif dalam mengidentifikasi hampir semua audio *deepfake* di data uji. Pentingnya, performa ini tidak datang dengan mengorbankan kelas *real*, yang juga memiliki Recall kuat sebesar 82%. Hal ini menunjukkan bahwa model tidak salah mengklasifikasikan audio asli secara

berlebihan. Nilai *precision* yang tinggi untuk kedua kelas (*fake*: 89%, *real*: 86%) juga menegaskan bahwa prediksi yang dibuat oleh model dapat diandalkan.

Distribusi kesalahan yang relatif seimbang ini menunjukkan bahwa model tidak memiliki bias yang signifikan terhadap salah satu kelas. Nilai Equal Error Rate (EER) yang rendah sebesar 13.29% mengonfirmasi hal ini, menunjukkan bahwa model telah mencapai titik ekuilibrium yang sangat baik antara kesalahan *false positive* dan *false negative*. Hasil analisis ini secara keseluruhan menunjukkan bahwa proses *fine-tuning* dan strategi pelatihan otomatis berhasil menghasilkan model deteksi yang tidak hanya akurat tetapi juga andal dan seimbang.

