

To load a dataset from Kaggle, you'll need the Kaggle API. If you don't have one, follow the instructions [here](#) to get one.

Once you have your API key, you can upload it to your Colab notebook to authenticate with Kaggle.

```
# Install the Kaggle library
!pip install kaggle

Requirement already satisfied: kaggle in /usr/local/lib/python3.12/dist-packages (1.7.4.5)
Requirement already satisfied: bleach in /usr/local/lib/python3.12/dist-packages (from kaggle) (6.3.0)
Requirement already satisfied: certifi>=14.05.14 in /usr/local/lib/python3.12/dist-packages (from kaggle) (2025.11.12)
Requirement already satisfied: charset-normalizer in /usr/local/lib/python3.12/dist-packages (from kaggle) (3.4.4)
Requirement already satisfied: idna in /usr/local/lib/python3.12/dist-packages (from kaggle) (3.11)
Requirement already satisfied: protobuf in /usr/local/lib/python3.12/dist-packages (from kaggle) (5.29.5)
Requirement already satisfied: python-dateutil>=2.5.3 in /usr/local/lib/python3.12/dist-packages (from kaggle) (2.9.0.post0)
Requirement already satisfied: python-slugify in /usr/local/lib/python3.12/dist-packages (from kaggle) (8.0.4)
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-packages (from kaggle) (2.32.4)
Requirement already satisfied: setuptools>=21.0.0 in /usr/local/lib/python3.12/dist-packages (from kaggle) (75.2.0)
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.12/dist-packages (from kaggle) (1.17.0)
Requirement already satisfied: text-unidecode in /usr/local/lib/python3.12/dist-packages (from kaggle) (1.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from kaggle) (4.67.1)
Requirement already satisfied: urllib3>=1.15.1 in /usr/local/lib/python3.12/dist-packages (from kaggle) (2.5.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.12/dist-packages (from kaggle) (0.5.1)
```

Now, upload your `kaggle.json` file to authenticate.

```
from google.colab import files

files.upload()

Choose Files kaggle.json
kaggle.json(application/json) - 63 bytes, last modified: 11/27/2025 - 100% done
Saving kaggle.json to kaggle.json
{'kaggle.json': b'{"username": "hosen42", "key": "379f4f241c066cfe7fdef48fa4544e7b"}'}
```

Next, move the file to the `.kaggle` directory and set the appropriate permissions.

```
!mkdir ~/.kaggle
!mv kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json

mkdir: cannot create directory '/root/.kaggle': File exists
```

```
!kaggle datasets download -d birdy654/cifake-real-and-ai-generated-synthetic-images

Dataset URL: https://www.kaggle.com/datasets/birdy654/cifake-real-and-ai-generated-synthetic-images
License(s): other
Downloading cifake-real-and-ai-generated-synthetic-images.zip to /content
  0% 0.00/105M [00:00<?, ?B/s]
100% 105M/105M [00:00<00:00, 1.73GB/s]
```

```
import zipfile
import os

# Assuming the downloaded file is cifake-real-and-ai-generated-synthetic-images.zip
zip_file_path = 'cifake-real-and-ai-generated-synthetic-images.zip'
extract_path = 'cifake-dataset'

with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall(extract_path)
```

1. Data Preparation

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras.preprocessing import image_dataset_from_directory
from tensorflow.keras import layers, models
import os
import random
import shutil
import tensorflow_hub as hub
```

```
!pip install -q visualkeras
!pip install -q tf-keras-vis matplotlib
import visualkeras
import matplotlib.pyplot as plt
from sklearn import metrics
from tf_keras_vis.gradcam import Gradcam
from tf_keras_vis.utils.scores import BinaryScore
from tensorflow.keras.preprocessing.image import load_img, img_to_array

train_dir_fake = '/content/cifake-dataset/train/FAKE'
train_dir_real = '/content/cifake-dataset/train/REAL'
```

1.0/1.0 MB 18.5 MB/s eta 0:00:00
 52.5/52.5 kB 2.1 MB/s eta 0:00:00

2.1 Prepare Directories

```
random.seed(2024)

# Prepare training sample directory
os.makedirs('/content/train_sample', exist_ok=True)
os.makedirs('/content/train_sample/FAKE', exist_ok=True)
os.makedirs('/content/train_sample/REAL', exist_ok=True)
# Prepare validation set directory
os.makedirs('/content/validation', exist_ok=True)
os.makedirs('//content/validation/FAKE', exist_ok=True)
os.makedirs('/content/validation/REAL', exist_ok=True)

# Select all training images and shuffle them
train_real_files = [f for f in os.listdir(train_dir_real) if os.path.isfile(os.path.join(train_dir_real, f))]
train_fake_files = [f for f in os.listdir(train_dir_fake) if os.path.isfile(os.path.join(train_dir_fake, f))]
random.shuffle(train_real_files)
random.shuffle(train_fake_files)

# Select 45000 training images per class
sample_real_train = train_real_files[:45000]
sample_fake_train = train_fake_files[:45000]

# Select 5000 validation images per class
sample_real_val = train_real_files[45000:50000]
sample_fake_val = train_fake_files[45000:50000]

# Copy selected images into train_sample directory
for file_name in sample_real_train:
    source_file = os.path.join(train_dir_real, file_name)
    destination_file = os.path.join('/content/train_sample/REAL', file_name)
    shutil.copy(source_file, destination_file)
for file_name in sample_fake_train:
    source_file = os.path.join(train_dir_fake, file_name)
    destination_file = os.path.join('/content/train_sample/FAKE', file_name)
    shutil.copy(source_file, destination_file)

# Also copy into validation directory
for file_name in sample_real_val:
    source_file = os.path.join(train_dir_real, file_name)
    destination_file = os.path.join('/content/validation/REAL', file_name)
    shutil.copy(source_file, destination_file)
for file_name in sample_fake_val:
    source_file = os.path.join(train_dir_fake, file_name)
    destination_file = os.path.join('/content/validation/FAKE', file_name)
    shutil.copy(source_file, destination_file)
```

2.2 Prepare Images into Tensorflow

```
# Load the (sample) training dataset (90000 images)
train = image_dataset_from_directory(
    '/content/train_sample',
    labels = 'inferred',
    label_mode = 'binary',
    image_size = (32, 32),
    batch_size = 32,
    shuffle = True)

# Load the validation set (10000 images)
validation = image_dataset_from_directory(
```

```

'/content/validation',
labels = 'inferred',
label_mode = 'binary',
image_size = (32, 32),
batch_size = 32,
shuffle = True)

# Load the test set (20000 images)
test = image_dataset_from_directory(
    '/content/cifake-dataset/test',
    labels = 'inferred',
    label_mode = 'binary',
    image_size = (32, 32),
    batch_size = 32,
    shuffle = False) # no shuffling
# The labels are as follows: FAKE = 0, REAL = 1

Found 90000 files belonging to 2 classes.
Found 10000 files belonging to 2 classes.
Found 20000 files belonging to 2 classes.

```

```

import os
import matplotlib.pyplot as plt
import numpy as np

# Define the paths to the directories
train_sample_dir = '/content/train_sample'
validation_dir = '/content/validation'
test_dir_fake = '/content/cifake-dataset/test/FAKE'
test_dir_real = '/content/cifake-dataset/test/REAL'

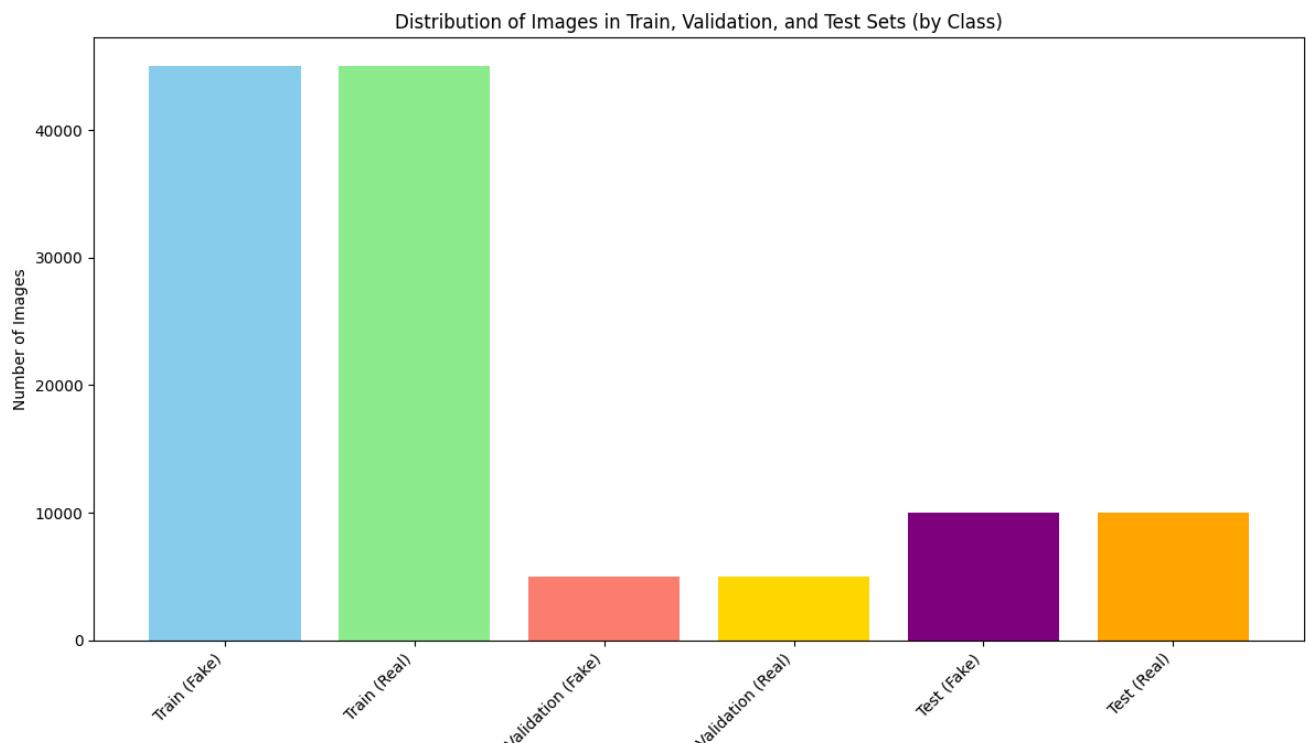
# Count the number of images in each directory
train_fake_count = len(os.listdir(os.path.join(train_sample_dir, 'FAKE')))
train_real_count = len(os.listdir(os.path.join(train_sample_dir, 'REAL')))
val_fake_count = len(os.listdir(os.path.join(validation_dir, 'FAKE')))
val_real_count = len(os.listdir(os.path.join(validation_dir, 'REAL')))
test_fake_count = len(os.listdir(test_dir_fake))
test_real_count = len(os.listdir(test_dir_real))

# Prepare data for visualization
labels = ['Train (Fake)', 'Train (Real)', 'Validation (Fake)', 'Validation (Real)', 'Test (Fake)', 'Test (Real)']
counts = [train_fake_count, train_real_count, val_fake_count, val_real_count, test_fake_count, test_real_count]

# Create a bar plot
plt.figure(figsize=(12, 7))
plt.bar(labels, counts, color=['skyblue', 'lightgreen', 'salmon', 'gold', 'purple', 'orange'])
plt.ylabel('Number of Images')
plt.title('Distribution of Images in Train, Validation, and Test Sets (by Class)')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

print(f"Total training images (Fake): {train_fake_count}")
print(f"Total training images (Real): {train_real_count}")
print(f"Total validation images (Fake): {val_fake_count}")
print(f"Total validation images (Real): {val_real_count}")
print(f"Total test images (Fake): {test_fake_count}")
print(f"Total test images (Real): {test_real_count}")

```

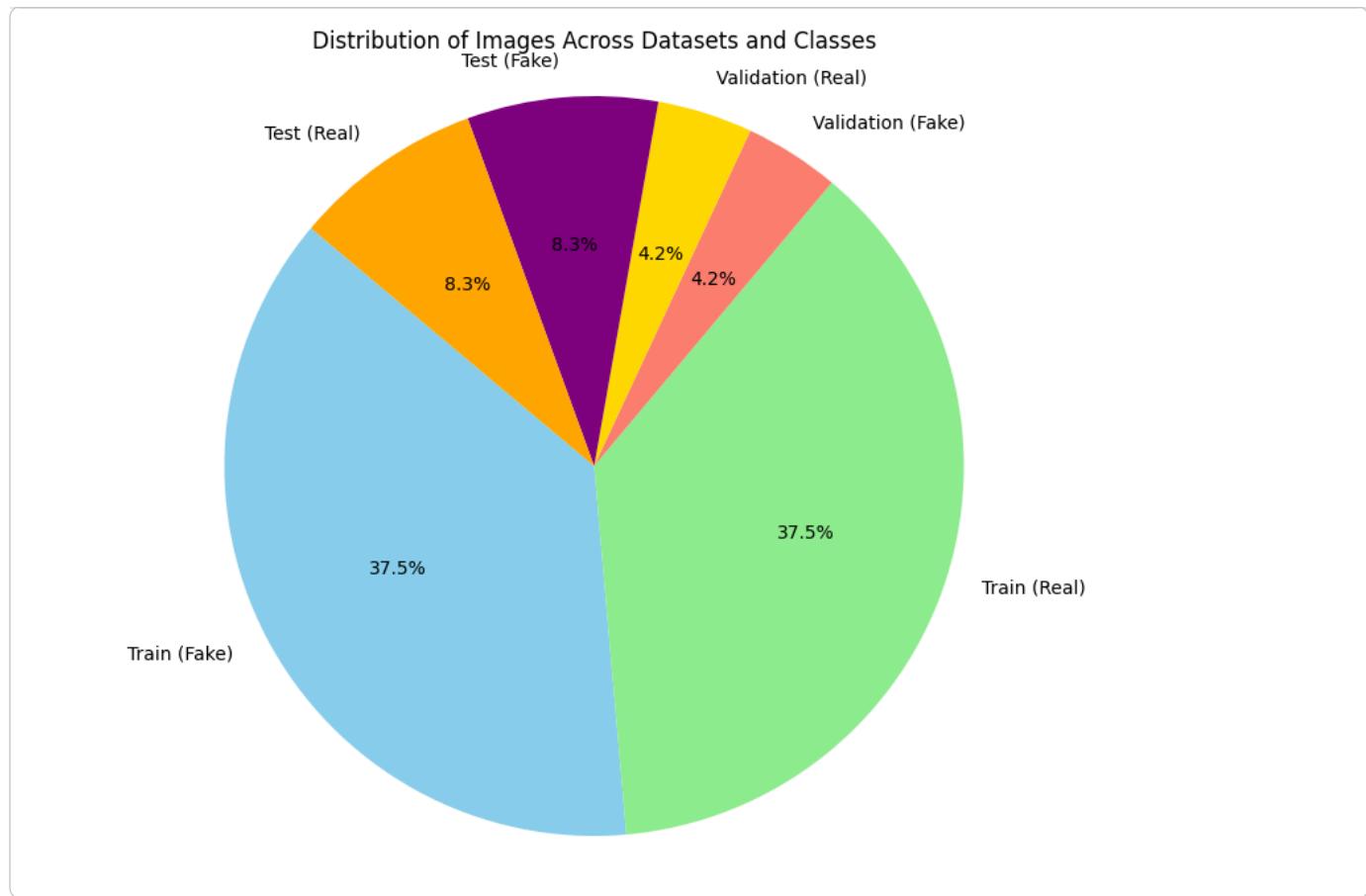


```
Total training images (Fake): 45000  
Total training images (Real): 45000  
Total validation images (Fake): 5000  
Total validation images (Real): 5000  
Total test images (Fake): 10000  
Total test images (Real): 10000
```

```
import matplotlib.pyplot as plt

# Reuse the counts and labels from the previous visualization
labels = ['Train (Fake)', 'Train (Real)', 'Validation (Fake)', 'Validation (Real)', 'Test (Fake)', 'Test (Real)']
counts = [train_fake_count, train_real_count, val_fake_count, val_real_count, test_fake_count, test_real_count]

# Create a pie chart
plt.figure(figsize=(10, 8))
plt.pie(counts, labels=labels, autopct='%1.1f%%', startangle=140, colors=['skyblue', 'lightgreen', 'salmon', 'gold', 'purple', 'darkviolet'])
plt.title('Distribution of Images Across Datasets and Classes')
plt.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()
```



Start coding or [generate](#) with AI.

```

import matplotlib.pyplot as plt
import os
import random
import cv2

# Get a list of all image files in the training set (REAL class)
image_files = [f for f in os.listdir('/content/cifake-dataset/train/REAL') if os.path.isfile(os.path.join('/content/cifake-dataset/train/REAL', f))]

# Select a random image file
random_image_file = random.choice(image_files)
random_image_path = os.path.join('/content/cifake-dataset/train/REAL', random_image_file)

# Read the image
img = cv2.imread(random_image_path)

if img is not None:
    # 1. Green Channel Extraction
    green_channel = img[:, :, 1]

    # Display the original and green channel images
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))

    # Display original image
    axes[0].imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    axes[0].set_title("Original Image")
    axes[0].axis('off')

    # Display green channel image (as grayscale)
    axes[1].imshow(green_channel, cmap='gray')
    axes[1].set_title("Green Channel")
    axes[1].axis('off')

    plt.tight_layout()
    plt.show()
else:
    print(f"Failed to load image: {random_image_path}")

```



The following preprocessing techniques were applied to the images:

- **Green Channel Extraction:** Isolates the green channel of the image, which can be useful for highlighting certain features.
- **CLAHE (Contrast Limited Adaptive Histogram Equalization):** Enhances the local contrast of the image.
- **Gaussian Blur:** Smooths the image to reduce noise.
- **Grayscale Conversion:** Converts the image to grayscale, reducing the color channels to one.
- **Canny Edge Detection:** Identifies edges within the image.
- **Sobel Filter:** Detects edges and gradients in the image.

▼ Task

Apply the following preprocessing steps to the "CIFAKE: Real and AI-Generated Synthetic Images" dataset: Green Channel Extraction, CLAHE, Gaussian Blur, Grayscale Conversion, Canny Edge Detection, and Sobel Filter. Then, load the preprocessed images into TensorFlow datasets for training, validation, and testing.

▼ Install opencv

Subtask:

Install the necessary library for image processing.

Reasoning: The subtask is to install the necessary library for image processing, which is opencv-python. This can be done using pip in a code block.

```
!pip install opencv-python
Requirement already satisfied: opencv-python in /usr/local/lib/python3.12/dist-packages (4.12.0.88)
Requirement already satisfied: numpy<2.3.0,>=2 in /usr/local/lib/python3.12/dist-packages (from opencv-python) (2.0.2)
```

▼ Define preprocessing function

Subtask:

Create a Python function that takes an image path as input and applies the specified preprocessing steps (Green Channel Extraction, CLAHE, Gaussian Blur, Grayscale Conversion, Canny Edge Detection, Sobel Filter).

Reasoning: Define a function to perform the specified preprocessing steps on an input image path.

```

import cv2
import numpy as np

def preprocess_image(image_path):
    """
    Applies various preprocessing steps to an image.

    Args:
        image_path: The path to the input image file.

    Returns:
        A dictionary containing the results of each preprocessing step.
    """

    # Read the image
    img = cv2.imread(image_path)
    if img is None:
        return None # Return None if the image cannot be read

    # 1. Green Channel Extraction
    green_channel = img[:, :, 1]

    # 2. CLAHE (Contrast Limited Adaptive Histogram Equalization) on Green Channel
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    clahe_green = clahe.apply(green_channel)

    # 3. Gaussian Blur on CLAHE-enhanced green channel
    gaussian.blur = cv2.GaussianBlur(clahe_green, (5, 5), 0)

    # 4. Grayscale Conversion (from original image)
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # 5. Canny Edge Detection on grayscale image
    canny.edges = cv2.Canny(gray_img, 100, 200) # Adjust thresholds as needed

    # 6. Sobel Filter on grayscale image
    sobelx = cv2.Sobel(gray_img, cv2.CV_64F, 1, 0, ksize=5)
    sobely = cv2.Sobel(gray_img, cv2.CV_64F, 0, 1, ksize=5)
    sobel.magnitude = np.sqrt(sobelx**2 + sobely**2)
    sobel.magnitude = cv2.normalize(sobel.magnitude, None, 0, 255, cv2.NORM_MINMAX, cv2.CV_8U) # Normalize to 0-255

    return {
        'green_channel': green_channel,
        'clahe_green': clahe_green,
        'gaussian.blur': gaussian.blur,
        'grayscale': gray_img,
        'canny.edges': canny.edges,
        'sobel.magnitude': sobel.magnitude
    }

```

▼ Create new directories

Subtask:

Create new directories to store the preprocessed images, maintaining the same class structure (FAKE/REAL) as the original dataset.

Reasoning: Create the necessary directory structure for the preprocessed dataset.

```

import os

# Define the base directory for preprocessed data
preprocessed_base_dir = 'preprocessed_cifake_dataset'

# Define the subdirectories for train, validation, and test sets
train_dir = os.path.join(preprocessed_base_dir, 'train')
validation_dir = os.path.join(preprocessed_base_dir, 'validation')
test_dir = os.path.join(preprocessed_base_dir, 'test')

# Create the base directory
os.makedirs(preprocessed_base_dir, exist_ok=True)

```

```
# Create the train, validation, and test subdirectories
os.makedirs(train_dir, exist_ok=True)
os.makedirs(validation_dir, exist_ok=True)
os.makedirs(test_dir, exist_ok=True)

# Create the FAKE and REAL subdirectories within each set
os.makedirs(os.path.join(train_dir, 'FAKE'), exist_ok=True)
os.makedirs(os.path.join(train_dir, 'REAL'), exist_ok=True)
os.makedirs(os.path.join(validation_dir, 'FAKE'), exist_ok=True)
os.makedirs(os.path.join(validation_dir, 'REAL'), exist_ok=True)
os.makedirs(os.path.join(test_dir, 'FAKE'), exist_ok=True)
os.makedirs(os.path.join(test_dir, 'REAL'), exist_ok=True)

print("Directory structure created successfully.")
```

Directory structure created successfully.

▼ Apply preprocessing to images

Subtask:

Iterate through the original image files in the train, validation, and test sets, apply the preprocessing function to each image, and save the processed images to the new directories.

Reasoning: Iterate through the original image files in the train, validation, and test sets, apply the preprocessing function to each image, and save the processed images to the new directories.

```
import os
import cv2

# Define the paths to the original and preprocessed image directories.
original_base_dir = '/content/cifake-dataset'
preprocessed_base_dir = 'preprocessed_cifake_dataset'

# Create a list of tuples: (source directory, destination directory, set name)
directory_map = [
    (os.path.join(original_base_dir, 'train'), os.path.join(preprocessed_base_dir, 'train'), 'train'),
    (os.path.join(original_base_dir, 'test'), os.path.join(preprocessed_base_dir, 'test'), 'test'),
    (os.path.join('/content/validation'), os.path.join(preprocessed_base_dir, 'validation'), 'validation') # Use the created validation directory
]

# Iterate through the list of directories.
for original_dir, preprocessed_dir, set_name in directory_map:
    print(f"Processing {set_name} set...")
    # Iterate through the 'FAKE' and 'REAL' subdirectories.
    for class_name in ['FAKE', 'REAL']:
        original_class_dir = os.path.join(original_dir, class_name)
        preprocessed_class_dir = os.path.join(preprocessed_dir, class_name)

        # Ensure the preprocessed class directory exists
        os.makedirs(preprocessed_class_dir, exist_ok=True)

        # Iterate through the image files.
        if os.path.exists(original_class_dir):
            for file_name in os.listdir(original_class_dir):
                # For each image file, construct the full path to the original image.
                original_image_path = os.path.join(original_class_dir, file_name)

                # Call the preprocess_image function with the original image path.
                preprocessed_images = preprocess_image(original_image_path)

                # If the preprocessing is successful
                if preprocessed_images is not None:
                    # Iterate through the returned dictionary of preprocessed images.
                    for step_name, processed_image in preprocessed_images.items():
                        # Construct a new file path in the corresponding destination directory
                        base, ext = os.path.splitext(file_name)
                        new_file_name = f"{base}_{step_name}{ext}"
                        new_file_path = os.path.join(preprocessed_class_dir, new_file_name)

                        # Save the preprocessed image to the new file path
                        cv2.imwrite(new_file_path, processed_image)
                else:

```

```

        print(f"Warning: Failed to process image {original_image_path}")
    else:
        print(f"Warning: Original directory not found: {original_class_dir}")

    print("Preprocessing complete.")

    Processing train set...
    Processing test set...
    Processing validation set...
    Preprocessing complete.

```

▼ Load preprocessed images

Subtask:

Load the preprocessed images from the new directories into TensorFlow datasets using `image_dataset_from_directory`.

Reasoning: Load the preprocessed images from the new directories into TensorFlow datasets using `image_dataset_from_directory`, following the instructions to define directory paths, set parameters, and assign to variables.

```

import os
from tensorflow.keras.preprocessing import image_dataset_from_directory

# Define the path to the base directory containing the preprocessed images.
preprocessed_base_dir = 'preprocessed_cifake_dataset'

# Load the preprocessed training dataset.
train_ds = image_dataset_from_directory(
    os.path.join(preprocessed_base_dir, 'train'),
    labels='inferred',
    label_mode='binary',
    image_size=(32, 32),
    batch_size=32,
    shuffle=True
)

# Load the preprocessed validation dataset.
validation_ds = image_dataset_from_directory(
    os.path.join(preprocessed_base_dir, 'validation'),
    labels='inferred',
    label_mode='binary',
    image_size=(32, 32),
    batch_size=32,
    shuffle=True
)

# Load the preprocessed test dataset.
test_ds = image_dataset_from_directory(
    os.path.join(preprocessed_base_dir, 'test'),
    labels='inferred',
    label_mode='binary',
    image_size=(32, 32),
    batch_size=32,
    shuffle=False # Do not shuffle the test dataset
)

print("TensorFlow datasets created successfully.")

Found 600000 files belonging to 2 classes.
Found 60000 files belonging to 2 classes.
Found 120000 files belonging to 2 classes.
TensorFlow datasets created successfully.

```

▼ Update visualization (optional)

Subtask:

Update the visualization to reflect the counts of the preprocessed images.

Reasoning: Count the preprocessed images in each directory and then visualize the counts using bar and pie charts to show the distribution across the datasets and classes.

```

import os
import matplotlib.pyplot as plt
import numpy as np

# Define the paths to the 'FAKE' and 'REAL' subdirectories within the preprocessed data
preprocessed_train_fake_dir = 'preprocessed_cifake_dataset/train/FAKE'
preprocessed_train_real_dir = 'preprocessed_cifake_dataset/train/REAL'
preprocessed_validation_fake_dir = 'preprocessed_cifake_dataset/validation/FAKE'
preprocessed_validation_real_dir = 'preprocessed_cifake_dataset/validation/REAL'
preprocessed_test_fake_dir = 'preprocessed_cifake_dataset/test/FAKE'
preprocessed_test_real_dir = 'preprocessed_cifake_dataset/test/REAL'

# Count the number of image files in each of these directories
preprocessed_train_fake_count = len(os.listdir(preprocessed_train_fake_dir))
preprocessed_train_real_count = len(os.listdir(preprocessed_train_real_dir))
preprocessed_validation_fake_count = len(os.listdir(preprocessed_validation_fake_dir))
preprocessed_validation_real_count = len(os.listdir(preprocessed_validation_real_dir))
preprocessed_test_fake_count = len(os.listdir(preprocessed_test_fake_dir))
preprocessed_test_real_count = len(os.listdir(preprocessed_test_real_dir))

# Prepare data for the bar plot
labels_bar = ['Preprocessed Train (Fake)', 'Preprocessed Train (Real)',  

              'Preprocessed Validation (Fake)', 'Preprocessed Validation (Real)',  

              'Preprocessed Test (Fake)', 'Preprocessed Test (Real)']
counts_bar = [preprocessed_train_fake_count, preprocessed_train_real_count,  

              preprocessed_validation_fake_count, preprocessed_validation_real_count,  

              preprocessed_test_fake_count, preprocessed_test_real_count]

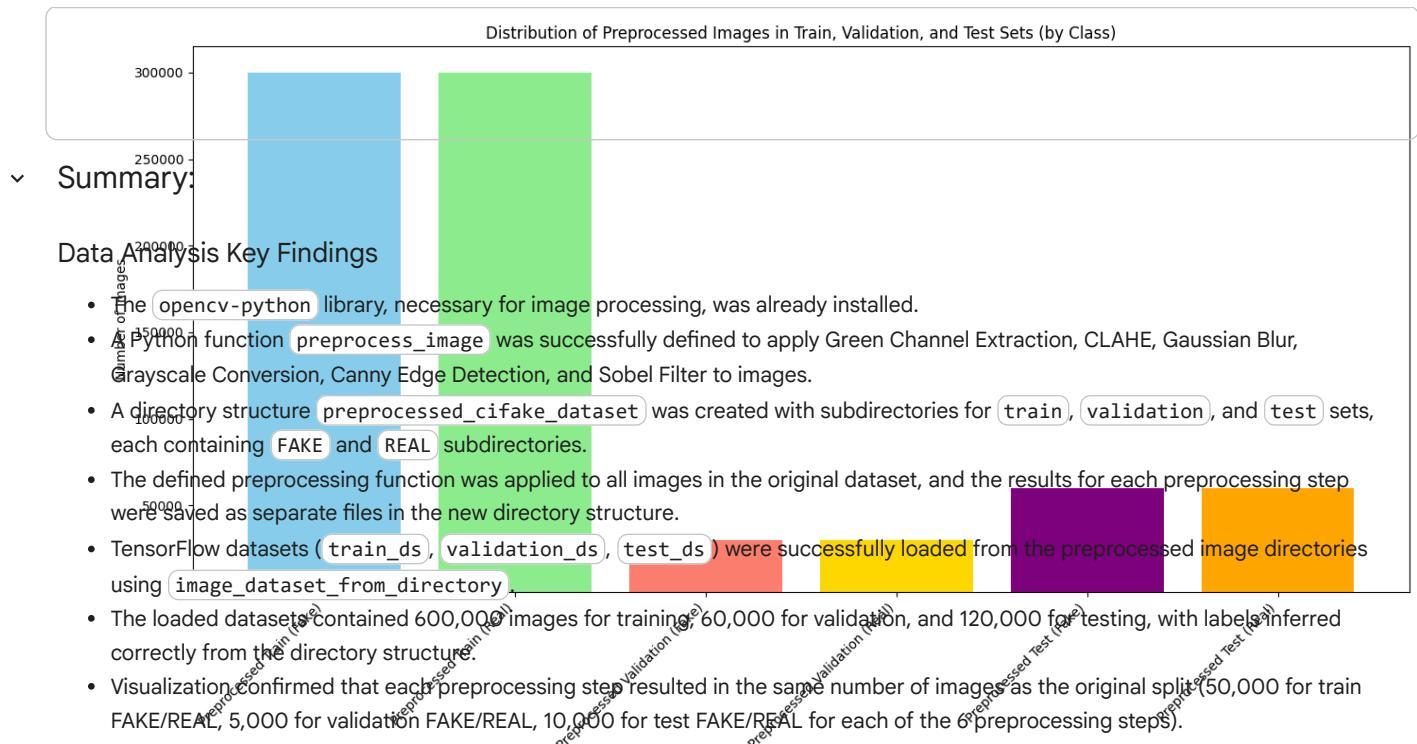
# Create the bar plot
plt.figure(figsize=(14, 8))
plt.bar(labels_bar, counts_bar, color=['skyblue', 'lightgreen', 'salmon', 'gold', 'purple', 'orange'])
plt.ylabel('Number of Images')
plt.title('Distribution of Preprocessed Images in Train, Validation, and Test Sets (by Class)')
plt.xticks(rotation=45, ha='right')
plt.tight_layout()
plt.show()

# Prepare data for the pie chart
labels_pie = labels_bar # Reuse labels
counts_pie = counts_bar # Reuse counts

# Create the pie chart
plt.figure(figsize=(10, 8))
plt.pie(counts_pie, labels=labels_pie, autopct='%1.1f%%', startangle=140, colors=['skyblue', 'lightgreen', 'salmon', 'gold', 'purple', 'orange'])
plt.title('Overall Distribution of Preprocessed Images Across Datasets and Classes')
plt.axis('equal') # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()

# Print the exact counts
print(f"Preprocessed Train (Fake): {preprocessed_train_fake_count}")
print(f"Preprocessed Train (Real): {preprocessed_train_real_count}")
print(f"Preprocessed Validation (Fake): {preprocessed_validation_fake_count}")
print(f"Preprocessed Validation (Real): {preprocessed_validation_real_count}")
print(f"Preprocessed Test (Fake): {preprocessed_test_fake_count}")
print(f"Preprocessed Test (Real): {preprocessed_test_real_count}")

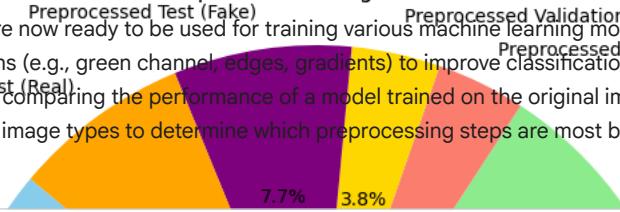
```

Insights or Next Steps

Overall Distribution of Preprocessed Images Across Datasets and Classes

- The preprocessed datasets are now ready to be used for training various machine learning models, potentially leveraging the different image representations (e.g., green channel, edges, gradients) to improve classification performance.
- Further analysis could involve comparing the performance of a model trained on the original images versus models trained on each of the different preprocessed image types to determine which preprocessing steps are most beneficial for the fake image detection task.



```

import matplotlib.pyplot as plt
import os
import random
import cv2

# Get a list of all image files in the training set (REAL class)
image_files = [f for f in os.listdir('/content/cifake-dataset/train/REAL') if os.path.isfile(os.path.join('/content/cifake-dataset/train/REAL', f))]

# Select a random image file
random_image_file = random.choice(image_files)
random_image_path = os.path.join('/content/cifake-dataset/train/REAL', random_image_file)

# Apply preprocessing to the selected image
preprocessed_images = preprocess_image(random_image_path)

if preprocessed_images:
    # Display the original and preprocessed images
    fig, axes = plt.subplots(2, 4, figsize=(20, 10))
    axes = axes.flatten()

    # Display original image
    original_img = cv2.imread(random_image_path)
    axes[0].imshow(cv2.cvtColor(original_img, cv2.COLOR_BGR2RGB))
    axes[0].set_title("Original Image")
    axes[0].axis('off')

    # Display preprocessed images
    for i, (step_name, processed_image) in enumerate(preprocessed_images.items()):
        if len(processed_image.shape) == 2: # Grayscale images
            axes[i + 1].imshow(processed_image, cmap='gray')
        else: # Color images (like green channel, though it will be grayscale visually)
            axes[i + 1].imshow(processed_image, cmap='gray') # Display green channel as grayscale
        axes[i + 1].set_title(step_name.replace('_', ' ').title())
        axes[i + 1].axis('off')

    # Hide any unused subplots
    for j in range(len(preprocessed_images) + 1, len(axes)):
        axes[j].axis('off')

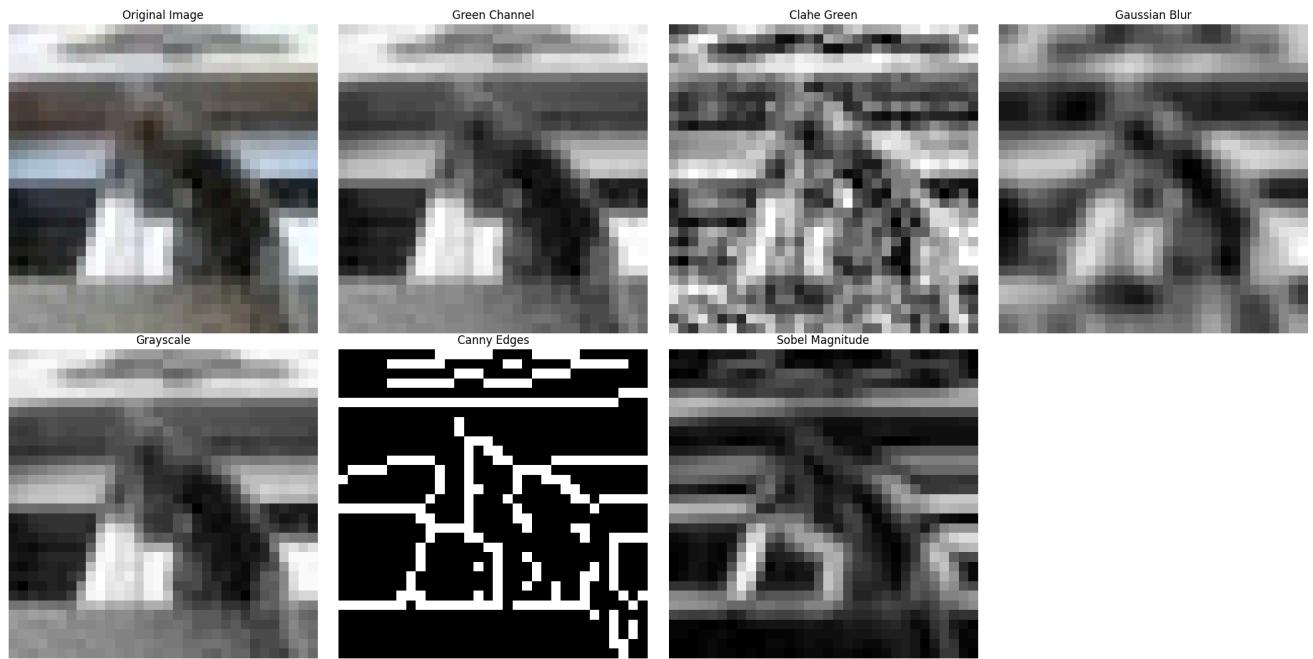
```

```

fig.delaxes(axes[j])

plt.tight_layout()
plt.show()
else:
    print(f"Failed to process image: {random_image_path}")

```



```

import tensorflow as tf

# Define data augmentation layers based on user's requirements
data_augmentation = tf.keras.Sequential([
    tf.keras.layers.RandomRotation(factor=(-0.2, 0.2)), # ±20 degrees (approx 0.2 radians)
    tf.keras.layers.RandomFlip("horizontal_and_vertical"),
    tf.keras.layers.RandomZoom(height_factor=(0.2, 0.2), width_factor=(0.2, 0.2)), # 80% to 120% zoom
    # TensorFlow doesn't have a direct "Image Translation" layer, RandomTranslation can be used
    # Adjust translation parameters as needed
    tf.keras.layers.RandomTranslation(height_factor=0.1, width_factor=0.1),
    tf.keras.layers.RandomContrast(factor=0.2), # Brightness/Contrast Adjustment (RandomContrast handles both)
    # TensorFlow doesn't have a built-in "Random Noise Addition" layer
    # A custom layer or function would be needed for this. We will skip this for now.
])

# Apply data augmentation to the training dataset
train_ds_augmented = train_ds.map(lambda x, y: (data_augmentation(x, training=True), y))

print("Data augmentation layers applied to the training dataset.")

```

Data augmentation layers applied to the training dataset.

```

import matplotlib.pyplot as plt

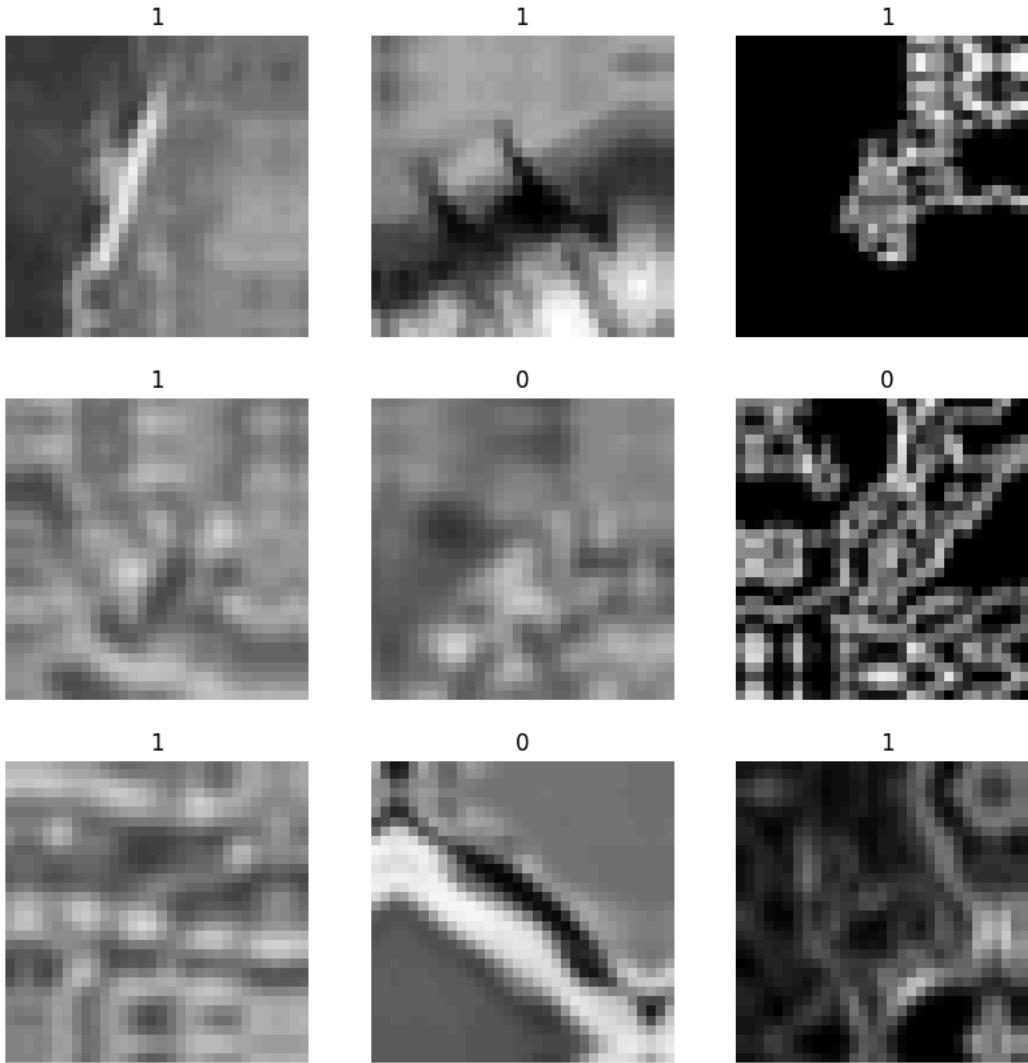
# Take one batch of augmented images from the training dataset
for images, labels in train_ds_augmented.take(1):
    plt.figure(figsize=(10, 10))
    # Display the first 9 images in the batch
    for i in range(min(9, images.shape[0])):

```

```

ax = plt.subplot(3, 3, i + 1)
# tf.keras.layers.Rescaling is not applied here, so convert to uint8 for display
plt.imshow(images[i].numpy().astype("uint8"))
plt.title(int(labels[i]))
plt.axis("off")
plt.show()

```



3. Network Architecture

```

from tensorflow.keras.optimizers import Adam
from tensorflow.keras.metrics import Recall, Precision
import tensorflow as tf
from tensorflow.keras import layers, models

model = tf.keras.Sequential([
    # Block 1
    layers.Conv2D(32, (3, 3), padding='same', activation='relu', input_shape=(32, 32, 3)),
    layers.BatchNormalization(),
    layers.Conv2D(32, (3, 3), padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Dropout(0.2),

    # Block 2
    layers.Conv2D(64, (3, 3), padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.Conv2D(64, (3, 3), padding='same', activation='relu'),
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Dropout(0.3),

    # Block 3
])

```

```

layers.Conv2D(128, (3, 3), padding='same', activation='relu'),
layers.BatchNormalization(),
layers.Conv2D(128, (3, 3), padding='same', activation='relu'),
layers.BatchNormalization(),
layers.MaxPooling2D(pool_size=(2, 2)),
layers.Dropout(0.3),

# Block 4
layers.Conv2D(256, (3, 3), padding='same', activation='relu'),
layers.BatchNormalization(),
layers.Conv2D(256, (3, 3), padding='same', activation='relu'),
layers.BatchNormalization(),
layers.MaxPooling2D(pool_size=(2, 2)),
layers.Dropout(0.4),

# Block 5 - No MaxPooling here
layers.Conv2D(512, (3, 3), padding='same', activation='relu'),
layers.BatchNormalization(),
layers.Conv2D(512, (3, 3), padding='same', activation='relu'),
layers.BatchNormalization(),
# Instead of MaxPooling2D, use GlobalAveragePooling2D
layers.GlobalAveragePooling2D(), # Prevents shape issues

layers.Dropout(0.5),

# Classification Head
layers.Dense(512, activation='relu'),
layers.BatchNormalization(),
layers.Dropout(0.5),
layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(
    optimizer=Adam(learning_rate=1e-5),
    loss='binary_crossentropy',
    metrics=['binary_accuracy', Precision(), Recall()]
)

```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input_shape` argument to `super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```

# Early Stopping
from tensorflow.keras import callbacks
early_stopping = callbacks.EarlyStopping(
    min_delta = 0.001, # minimum amount of change to count as an improvement
    patience = 20, # how many epochs to wait (while there's no improvement) before stopping
    restore_best_weights = True)

```

```

# Visualization of Architecture
model.summary()
# visualkeras.layered_view(model, legend=True) # Commented out to prevent AttributeError. See explanation above for fix.

```


Model: "sequential_1"

4. Training

Layer (type)	Output Shape	Param #
import random		
import numpy as np		
import tensorflow as tf		
# Set random seeds for reproducibility		
random.seed(2024)		
np.random.seed(2024)		
tf.random.set_seed(2024)		
# Train the model		
history = model.fit(train, epochs=200, validation_data=validation, callbacks=[early_stopping], verbose=1 # Show progress)		
2813/2813 normalization_3	15s 5ms/step 16, binary_accuracy: 0.9833 - loss: 0.0435 - precision: 0.9832 - recall: 0.9833 - va	
Epoch 144/200		
2813/2813	15s 5ms/step - binary_accuracy: 0.9839 - loss: 0.0423 - precision: 0.9842 - recall: 0.9837 - va	
Epoch 145/200 (MaxPooling2D)	(None, 8, 8, 64) 0	
2813/2813	15s 5ms/step - binary_accuracy: 0.9839 - loss: 0.0428 - precision: 0.9845 - recall: 0.9833 - va	
Epoch 146/200 (Dropout)	(None, 8, 8, 64) 0	
2813/2813	15s 5ms/step - binary_accuracy: 0.9844 - loss: 0.0412 - precision: 0.9843 - recall: 0.9846 - va	
Epoch 147/200 (Conv2D)	(None, 8, 8, 128) 73,856	
2813/2813	15s 5ms/step - binary_accuracy: 0.9845 - loss: 0.0413 - precision: 0.9842 - recall: 0.9849 - va	
Epoch 148/200 (BatchNormalization_4)	(None, 8, 8, 128) 512	
2813/2813 normalization_4	15s 5ms/step - binary_accuracy: 0.9850 - loss: 0.0406 - precision: 0.9844 - recall: 0.9856 - va	
Epoch 149/200		
2813/2813 (Conv2D)	16s 6ms/step 8, binary_accuracy: 0.98517,586 - loss: 0.0398 - precision: 0.9852 - recall: 0.9851 - va	
Epoch 150/200		
2813/2813 normalization_5	16s 6ms/step 8, binary_accuracy: 0.9852 - loss: 0.0400 - precision: 0.9856 - recall: 0.9848 - va	
Epoch 151/200 (BatchNormalization)		
2813/2813	15s 5ms/step - binary_accuracy: 0.9858 - loss: 0.0408 - precision: 0.9857 - recall: 0.9858 - va	
Epoch 152/200 (MaxPooling2D)	(None, 4, 4, 128) 0	
2813/2813	15s 5ms/step - binary_accuracy: 0.9859 - loss: 0.0377 - precision: 0.9858 - recall: 0.9860 - va	
Epoch 153/200 (Dropout)	(None, 4, 4, 128) 0	
2813/2813	15s 5ms/step - binary_accuracy: 0.9853 - loss: 0.0392 - precision: 0.9852 - recall: 0.9854 - va	
Epoch 154/200 (Conv2D)	(None, 4, 4, 256) 295,168	
2813/2813 normalization_6	15s 5ms/step - binary_accuracy: 0.9859 - loss: 0.0370 - precision: 0.9862 - recall: 0.9855 - va	
Epoch 155/200 (BatchNormalization)	(None, 4, 4, 256) 1,024	
2813/2813	15s 5ms/step - binary_accuracy: 0.9865 - loss: 0.0374 - precision: 0.9866 - recall: 0.9864 - va	
Epoch 156/200		
2813/2813 (Conv2D)	15s 5ms/step 4, binary_accuracy: 0.9862 - loss: 0.0373 - precision: 0.9854 - recall: 0.9870 - va	
Epoch 157/200		
2813/2813 normalization_7	16s 6ms/step 4, binary_accuracy: 0.9854 - loss: 0.0381 - precision: 0.9858 - recall: 0.9849 - va	
Epoch 158/200 (BatchNormalization)		
2813/2813	15s 5ms/step - binary_accuracy: 0.9859 - loss: 0.0381 - precision: 0.9865 - recall: 0.9854 - va	
Epoch 159/200 (MaxPooling2D)	(None, 2, 2, 256) 0	
2813/2813	15s 5ms/step - binary_accuracy: 0.9857 - loss: 0.0379 - precision: 0.9859 - recall: 0.9855 - va	
Epoch 160/200 (Dropout)	(None, 2, 2, 256) 0	
2813/2813 (Conv2D)	15s 5ms/step - binary_accuracy: 0.9857 - loss: 0.0368 - precision: 0.9851 - recall: 0.9863 - va	
Epoch 161/200		
2813/2813 normalization_8	15s 5ms/step 2, binary_accuracy: 0.9859 - loss: 0.0372 - precision: 0.9866 - recall: 0.9852 - va	
Epoch 162/200 (BatchNormalization)	(None, 2, 2, 312) 2,048	
2813/2813	15s 5ms/step - binary_accuracy: 0.9872 - loss: 0.0342 - precision: 0.9876 - recall: 0.9868 - va	
Epoch 163/200 (Conv2D)	(None, 2, 2, 512) 2,359,808	
2813/2813	15s 5ms/step - binary_accuracy: 0.9870 - loss: 0.0352 - precision: 0.9869 - recall: 0.9870 - va	
Epoch 164/200 (MaxPooling2D)	(None, 2, 2, 512) 2,048	
2813/2813 normalization_9	15s 5ms/step - binary_accuracy: 0.9871 - loss: 0.0351 - precision: 0.9868 - recall: 0.9874 - va	
Epoch 165/200 (BatchNormalization)		
2813/2813	15s 5ms/step - binary_accuracy: 0.9874 - loss: 0.0331 - precision: 0.9876 - recall: 0.9871 - va	
Epoch 166/200 (Conv2D)	(None, 2, 2, 512) 2,359,808	
2813/2813	15s 5ms/step - binary_accuracy: 0.9870 - loss: 0.0345 - precision: 0.9868 - recall: 0.9872 - va	
Epoch 167/200 (Dropout)	(None, 512) 0	
2813/2813	15s 5ms/step - binary_accuracy: 0.9874 - loss: 0.0340 - precision: 0.9876 - recall: 0.9871 - va	
Epoch 168/200 (Dense)	(None, 512) 262,656	
2813/2813	15s 5ms/step - binary_accuracy: 0.9883 - loss: 0.0319 - precision: 0.9885 - recall: 0.9881 - va	
Epoch 169/200 (BatchNormalization)	(None, 512) 2,048	
2813/2813	16s 6ms/step - binary_accuracy: 0.9882 - loss: 0.0312 - precision: 0.9886 - recall: 0.9879 - va	
Epoch 170/200		
2813/2813 (Dropout)	15s 5ms/step 512 binary_accuracy: 0.9871 - loss: 0.0349 - precision: 0.9873 - recall: 0.9869 - va	
Epoch 171/200		
2813/2813 (Dense)	16s 6ms/step 1 binary_accuracy: 0.9879 - loss: 0.0329 - precision: 0.9879 - recall: 0.9879 - va	

Total params: 4,985,377 (19.02 MB)

import matplotlib.pyplot as plt

```
import matplotlib.pyplot as plt
import pandas as pd

history_frame = pd.DataFrame(history.history)

fig, axes = plt.subplots(2, 2, figsize=(14, 10))

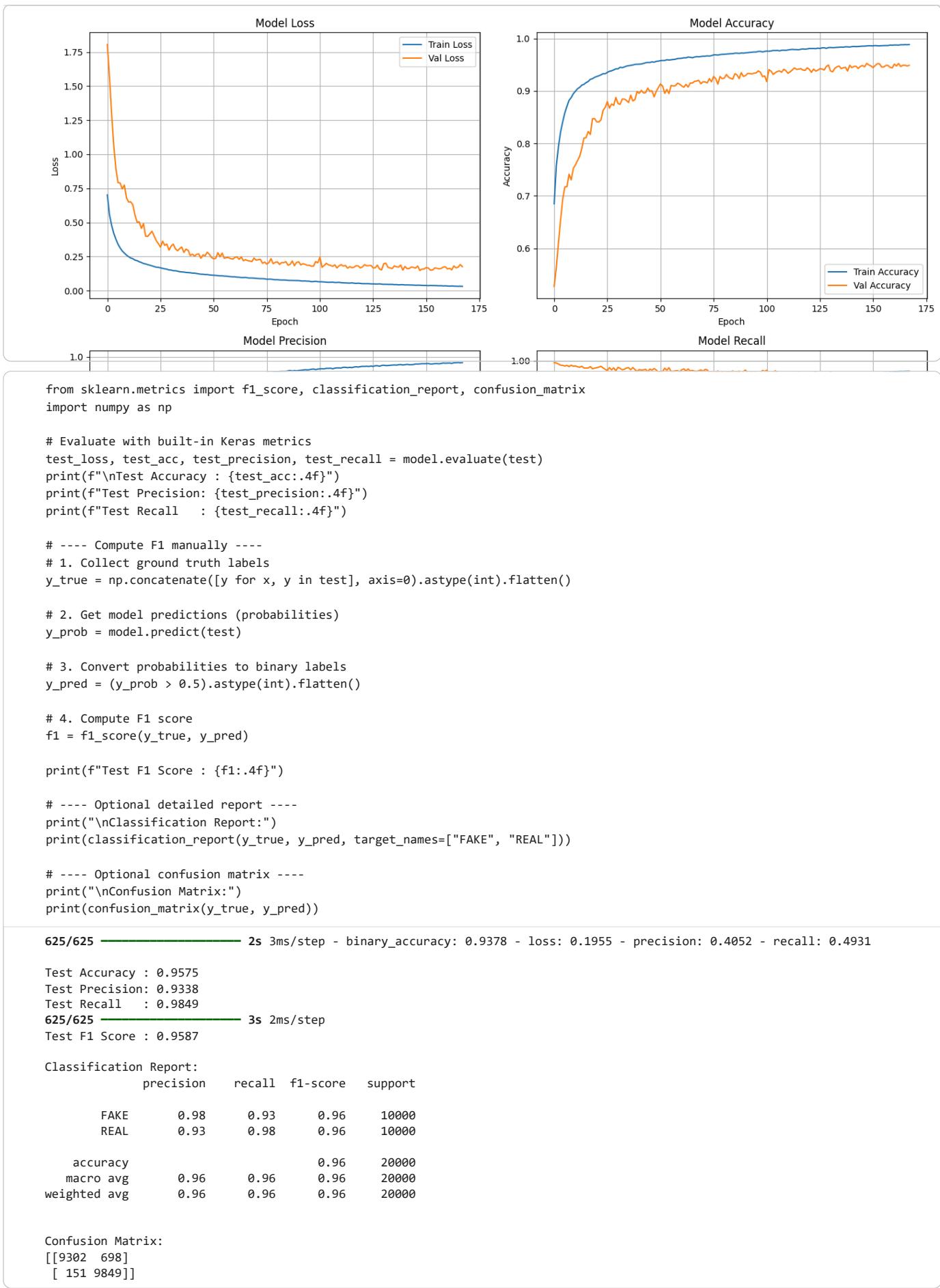
# --- Loss ---
axes[0, 0].plot(history_frame['loss'], label='Train Loss')
axes[0, 0].plot(history_frame['val_loss'], label='Val Loss')
axes[0, 0].set_title('Model Loss')
axes[0, 0].set_xlabel('Epoch')
axes[0, 0].set_ylabel('Loss')
axes[0, 0].legend()
axes[0, 0].grid(True)

# --- Accuracy ---
axes[0, 1].plot(history_frame['binary_accuracy'], label='Train Accuracy')
axes[0, 1].plot(history_frame['val_binary_accuracy'], label='Val Accuracy')
axes[0, 1].set_title('Model Accuracy')
axes[0, 1].set_xlabel('Epoch')
axes[0, 1].set_ylabel('Accuracy')
axes[0, 1].legend()
axes[0, 1].grid(True)

# --- Precision (use precision) ---
axes[1, 0].plot(history_frame['precision'], label='Train Precision')
axes[1, 0].plot(history_frame['val_precision'], label='Val Precision')
axes[1, 0].set_title('Model Precision')
axes[1, 0].set_xlabel('Epoch')
axes[1, 0].set_ylabel('Precision')
axes[1, 0].legend()
axes[1, 0].grid(True)

# --- Recall (use recall) ---
axes[1, 1].plot(history_frame['recall'], label='Train Recall')
axes[1, 1].plot(history_frame['val_recall'], label='Val Recall')
axes[1, 1].set_title('Model Recall')
axes[1, 1].set_xlabel('Epoch')
axes[1, 1].set_ylabel('Recall')
axes[1, 1].legend()
axes[1, 1].grid(True)

plt.tight_layout()
plt.show()
```



4.2. Training Confusion Matrix

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report, f1_score, precision_score, recall_
import numpy as np
import matplotlib.pyplot as plt

# Collect ground-truth and predictions from the validation set
y_true, y_pred = [], []
for images, labels in validation:
    probs = model.predict(images, verbose=0)
    y_pred.extend((probs > 0.5).astype(int).ravel())           # predicted labels {0,1}
    y_true.extend(labels.numpy().astype(int).ravel())           # true labels {0,1}

y_true = np.array(y_true)
y_pred = np.array(y_pred)

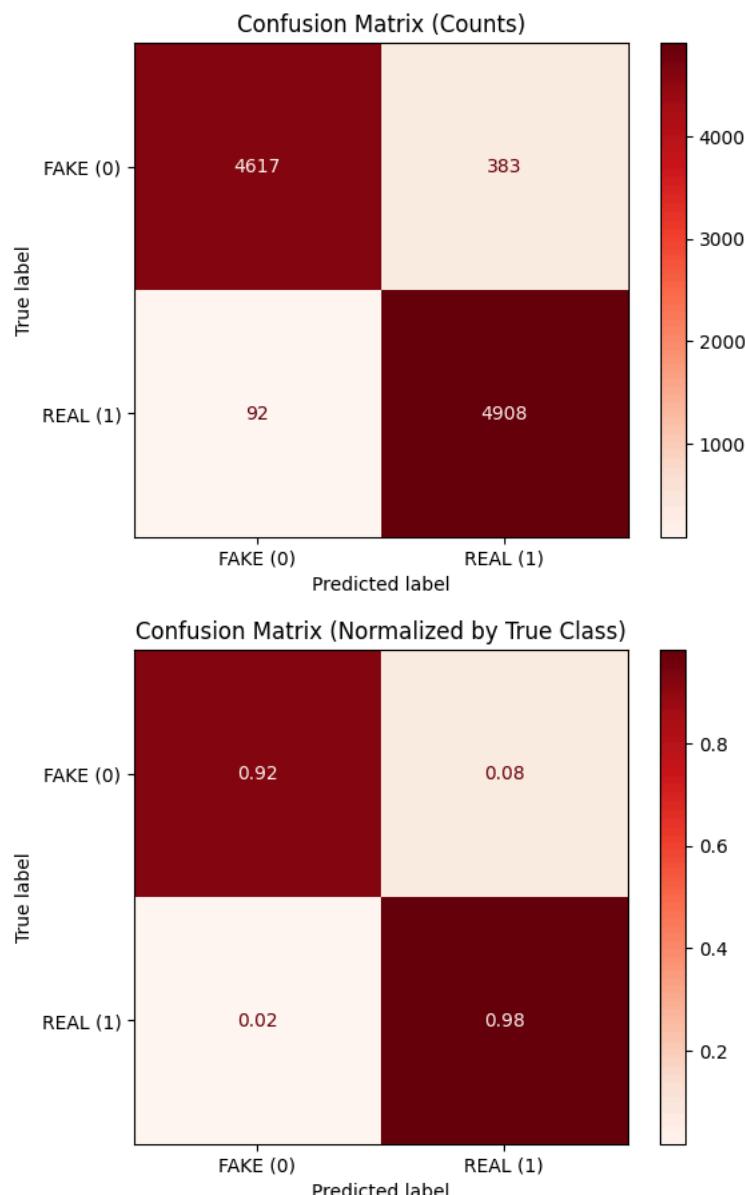
# Confusion matrix (rows = true, cols = predicted)
cm = confusion_matrix(y_true, y_pred, labels=[0, 1])

# Plot raw counts
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["FAKE (0)", "REAL (1)"])
disp.plot(cmap=plt.cm.Reds, values_format='d')
plt.title("Confusion Matrix (Counts)")
plt.xlabel("Predicted label")
plt.ylabel("True label")
plt.grid(False)
plt.show()

# Plot normalized (per true class)
cm_norm = confusion_matrix(y_true, y_pred, labels=[0, 1], normalize='true')
disp_norm = ConfusionMatrixDisplay(confusion_matrix=cm_norm, display_labels=["FAKE (0)", "REAL (1)"])
disp_norm.plot(cmap=plt.cm.Reds, values_format='.2f')
plt.title("Confusion Matrix (Normalized by True Class)")
plt.xlabel("Predicted label")
plt.ylabel("True label")
plt.grid(False)
plt.show()

# Metrics summary
prec = precision_score(y_true, y_pred)
rec = recall_score(y_true, y_pred)
f1 = f1_score(y_true, y_pred)

print(f"Precision: {prec:.4f}")
print(f"Recall : {rec:.4f}")
print(f"F1 Score : {f1:.4f}\n")
print(classification_report(y_true, y_pred, target_names=["FAKE", "REAL"]))
```



Precision: 0.9276
 Recall : 0.9816
 F1 Score : 0.9538

	precision	recall	f1-score	support
FAKE	0.98	0.92	0.95	5000
REAL	0.93	0.98	0.95	5000
accuracy			0.95	10000
macro avg	0.95	0.95	0.95	10000
weighted avg	0.95	0.95	0.95	10000

These results are very promising. You can see from the graphs that the model is neither overfitting (due to the similarity between training and validation), or underfitting (due to the satisfactorily high metrics). All metrics are high, above .90, indicating the model is quite good at distinguishing between classes. The improvements made over epochs are (mostly) smooth, unlike the volatile changes in previous models.

In practical terms:

Binary Accuracy: This means that the model correctly classified images in 94.75% of the validation cases. This surpasses the bound of 92.98% found by Bird & Lotfi, but the performance here is not on the test set. We will look at the test-set performance later.

Recall (Sensitivity): With a recall of 92.86% on the validation set (for the artificial cases), the model is relatively good at identifying images as artificial, given that they truly are artificial. Being lower than the precision metric, this shows that the classifier has a (slightly) higher

tendency for False Negatives (rather than False Positives), which may be detrimental in practice (as we argue that the most important metric here is recall).

Precision: Precision (96.50%) unlike recall, reflects the proportion of true fake images from the total number of images that were identified as fake. Thus, the classifier is less likely to incorrectly identify an image as artificial, than it is to miss correct identification of artificial images.

5. Test-Set Performance

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, ConfusionMatrixDisplay, classification_report
)

# --- Predict on test set ---
random.seed(2024)
probs = model.predict(test, verbose=0)
y_pred = (probs > 0.5).astype(int).ravel()

# True labels
y_true = np.concatenate([y.numpy() for _, y in test], axis=0).astype(int).ravel()

# --- Scalars ---
acc = accuracy_score(y_true, y_pred)                                # positive class = 1 (REAL) by default
prec = precision_score(y_true, y_pred)                               # recall for class 1
rec1 = recall_score(y_true, y_pred)                                 # recall for class 0 (FAKE)
rec0 = recall_score(y_true, y_pred, pos_label=0)                   # recall for class 0 (FAKE)
f1 = f1_score(y_true, y_pred)

print(f"Accuracy : {acc:.4f}")
print(f"Precision (class=1) : {prec:.4f}")
print(f"Recall (REAL=1) : {rec1:.4f}")
print(f"Recall (FAKE=0) : {rec0:.4f}" # <- the one you prioritize
print(f"F1 Score : {f1:.4f}\n")

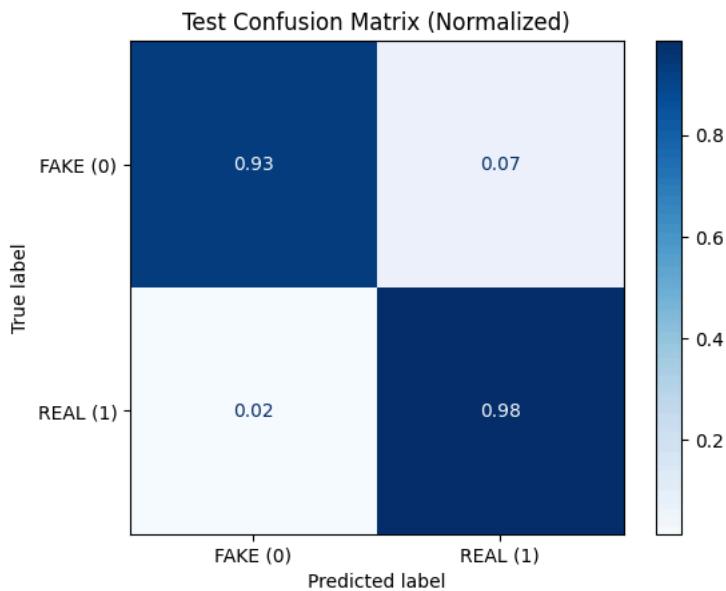
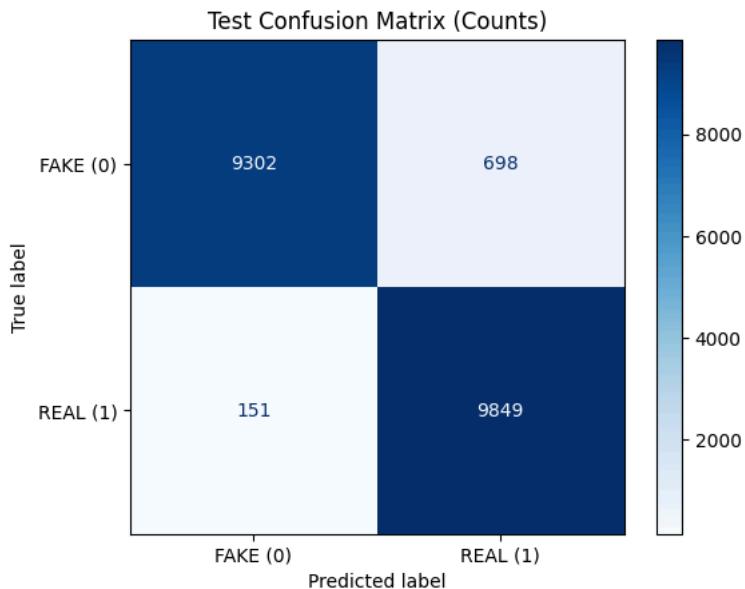
# --- Confusion matrix (counts) ---
cm = confusion_matrix(y_true, y_pred, labels=[0, 1])
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["FAKE (0)", "REAL (1)"])
disp.plot(cmap=plt.cm.Blues, values_format='d')
plt.title("Test Confusion Matrix (Counts)")
plt.xlabel("Predicted label")
plt.ylabel("True label")
plt.grid(False)
plt.show()

# --- Confusion matrix (normalized by true class) ---
cm_norm = confusion_matrix(y_true, y_pred, labels=[0, 1], normalize='true')
disp_norm = ConfusionMatrixDisplay(confusion_matrix=cm_norm, display_labels=["FAKE (0)", "REAL (1)"])
disp_norm.plot(cmap=plt.cm.Blues, values_format='.2f')
plt.title("Test Confusion Matrix (Normalized)")
plt.xlabel("Predicted label")
plt.ylabel("True label")
plt.grid(False)
plt.show()

# --- Detailed report (per-class precision/recall/F1) ---
print(classification_report(y_true, y_pred, target_names=["FAKE", "REAL"]))

```

```
Accuracy : 0.9576
Precision (class=1) : 0.9338
Recall (REAL=1) : 0.9849
Recall (FAKE=0) : 0.9302
F1 Score : 0.9587
```



	precision	recall	f1-score	support
FAKE	0.98	0.93	0.96	10000
REAL	0.93	0.98	0.96	10000
accuracy			0.96	20000
macro avg	0.96	0.96	0.96	20000
weighted avg	0.96	0.96	0.96	20000

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc
import numpy as np

# Get true labels and predicted probabilities from the test set
y_true = np.concatenate([y.numpy() for _, y in test], axis=0).astype(int).ravel()
y_prob = model.predict(test).ravel() # Get probabilities for the positive class (REAL)

# Calculate ROC curve and AUC
fpr, tpr, thresholds = roc_curve(y_true, y_prob)
roc_auc = auc(fpr, tpr)

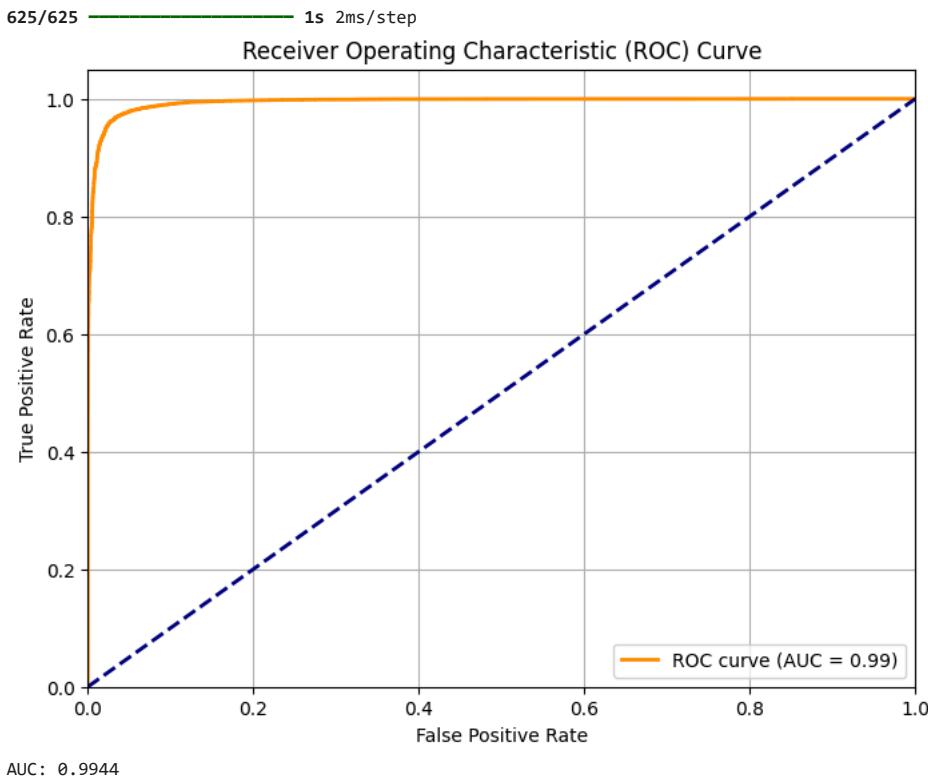
# Plot the ROC curve
```

```

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.grid(True)
plt.show()

print(f"AUC: {roc_auc:.4f}")

```



Results are satisfactory:

Binary Accuracy - 94.295%. Surpassing the performance of the dataset authors' model (92.98%, Bird & Lotfi, 2024), our classifier proves highly effective at differentiating between real and AI-generated images, if only for images similar to those used for training.

Recall (Sensitivity) - 92.98%. With similar recall performance to that of the dataset's authors, the classifier is reasonably good at identifying artificial images. With about 7% of artificial images passing as real, the network is by no means perfect, but provides great improvements over the human eye.

6. Feature Importance: Grad-CAM

Interpretability is important to this project, as human detection of AI-generated images can be considered similar to the convolutional process, there are parts one might focus on to make a decision on an image's origin. Here, we use gradient-weighted class activation mapping (Grad-CAM) (Selvaraju et al., 2016) to visualise which parts of an image contribute most to the model's classification.

Grad-CAM works by using class specific gradients flowing into the last convolutional layer, computing a weighted combination of this layer's feature maps and their weights and then using them to produce a heatmap highlighting influential regions.

Code for this part was originally based on Christian Versloot's article, adapted for compatibility with our project and then corrected with ChatGPT.

```

gradcam = Gradcam(model, clone=True)

def binary_score_function(output):
    return output[..., 0]

for images, labels in validation.take(1): #you can rerun cell to get more, individuals not so important as in aggregate

```

```
cam_images = images[:10]
cam_labs = labels[:10]

for i in range(len(cam_images)):
    input_image = cam_images[i].numpy()
    input_lab = cam_labs[i].numpy()

    pred_lab = model.predict(tf.expand_dims(input_image, axis=0))[0][0] #model predicted labels, rounded
    pred_class = 'REAL' if pred_lab > 0.5 else 'FAKE'
    true_class = 'REAL' if input_lab == 1 else 'FAKE'

    cam = gradcam(binary_score_function, tf.expand_dims(input_image, axis=0))[0] #actually making the cam

    heatmap = np.uint8(255 * cam)
    overlay = tf.keras.utils.array_to_img(input_image).convert("RGBA")
    heatmap_img = plt.cm.jet(heatmap)[..., :3] * 255
    heatmap_overlay = tf.keras.utils.array_to_img(np.uint8(heatmap_img)).convert("RGBA")
    blended = tf.keras.utils.img_to_array(overlay).astype("uint8")
    blended[..., :3] = (blended[..., :3] * 0.5 + heatmap_img * 0.5).astype("uint8")

    fig, axes = plt.subplots(1, 3, figsize=(12, 4))
    fig.suptitle(f'True Label = {true_class}, Predicted Label = {pred_class} (Confidence: {pred_lab:.2f})')

    axes[0].imshow(input_image.astype('uint8')) #original
    axes[0].set_title('Input Image')
    axes[0].axis('off')

    axes[1].imshow(cam, cmap='jet') #cam
    axes[1].set_title('Grad-CAM Heatmap')
    axes[1].axis('off')

    axes[2].imshow(blended) #cam overlaid on input
    axes[2].set_title('Overlay')
    axes[2].axis('off')

plt.show()
```


1/1 ━━━━━━ 1s 1s/step

True Label = FAKE, Predicted Label = FAKE (Confidence: 0.00)

Input Image



Grad-CAM Heatmap



Overlay



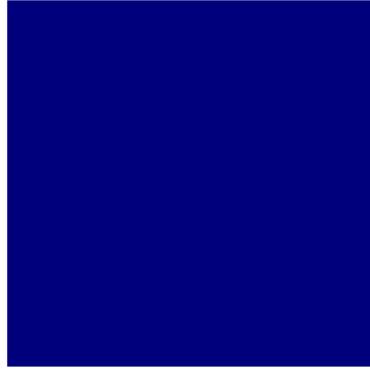
1/1 ━━━━━━ 0s 34ms/step

True Label = FAKE, Predicted Label = FAKE (Confidence: 0.48)

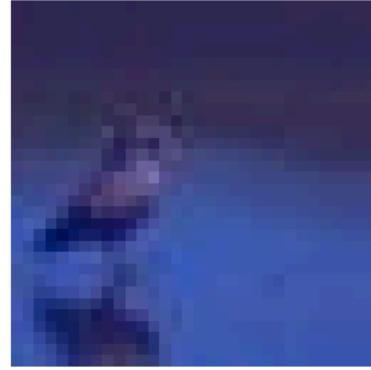
Input Image



Grad-CAM Heatmap



Overlay



1/1 ━━━━━━ 0s 35ms/step

True Label = FAKE, Predicted Label = FAKE (Confidence: 0.00)

Input Image



Grad-CAM Heatmap



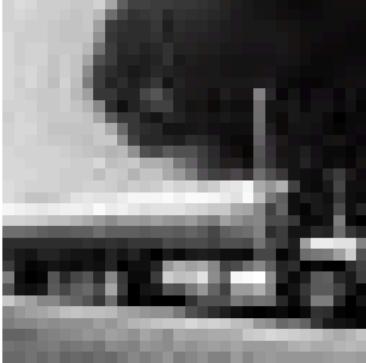
Overlay



1/1 ━━━━━━ 0s 34ms/step

True Label = FAKE, Predicted Label = FAKE (Confidence: 0.00)

Input Image



Grad-CAM Heatmap

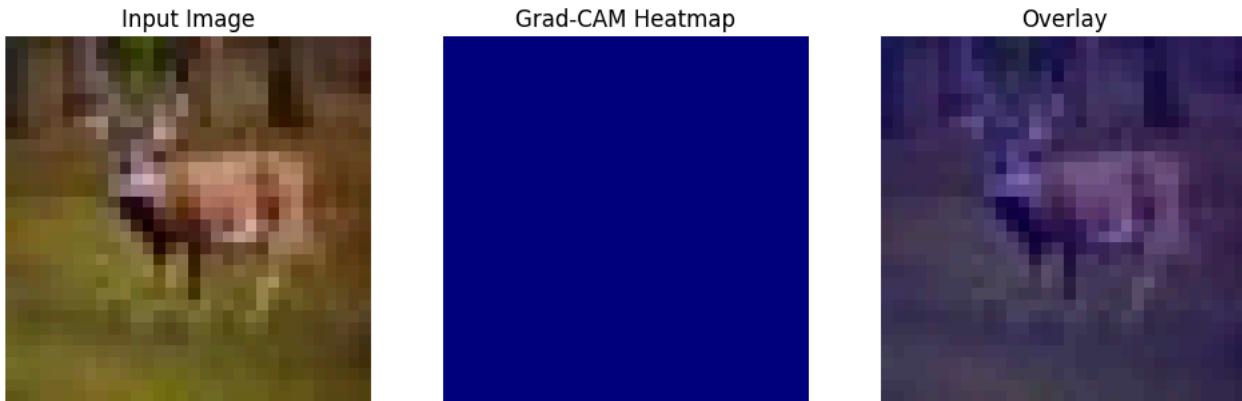


Overlay



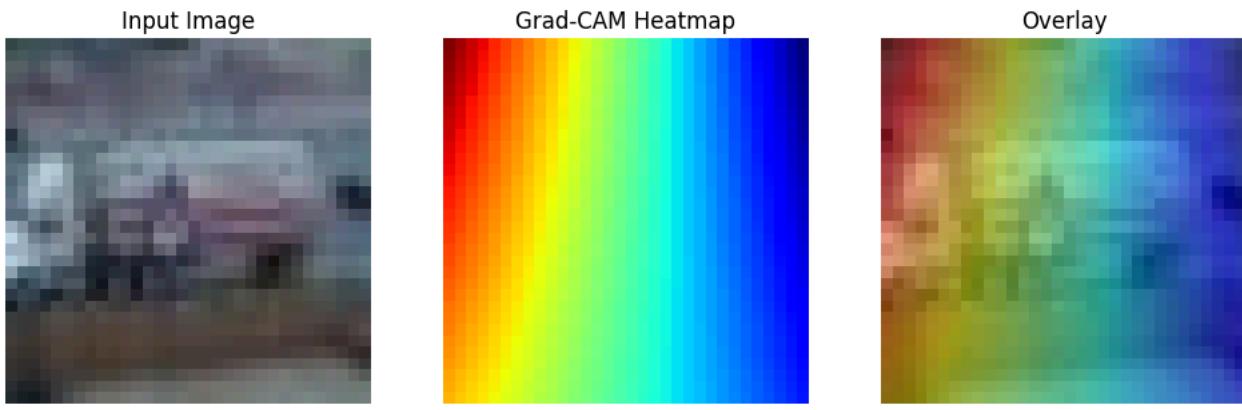
1/1 ————— 0s 34ms/step

True Label = FAKE, Predicted Label = FAKE (Confidence: 0.00)



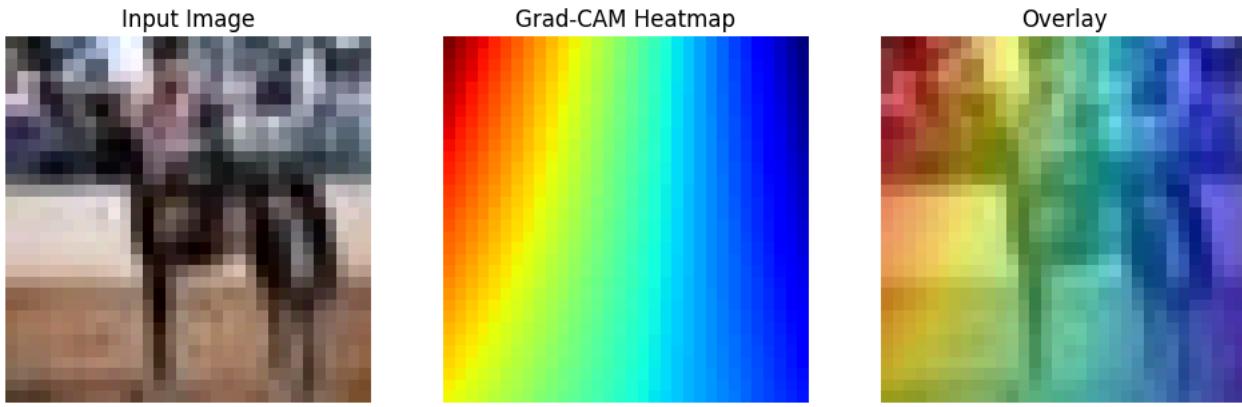
1/1 ————— 0s 34ms/step

True Label = REAL, Predicted Label = REAL (Confidence: 1.00)



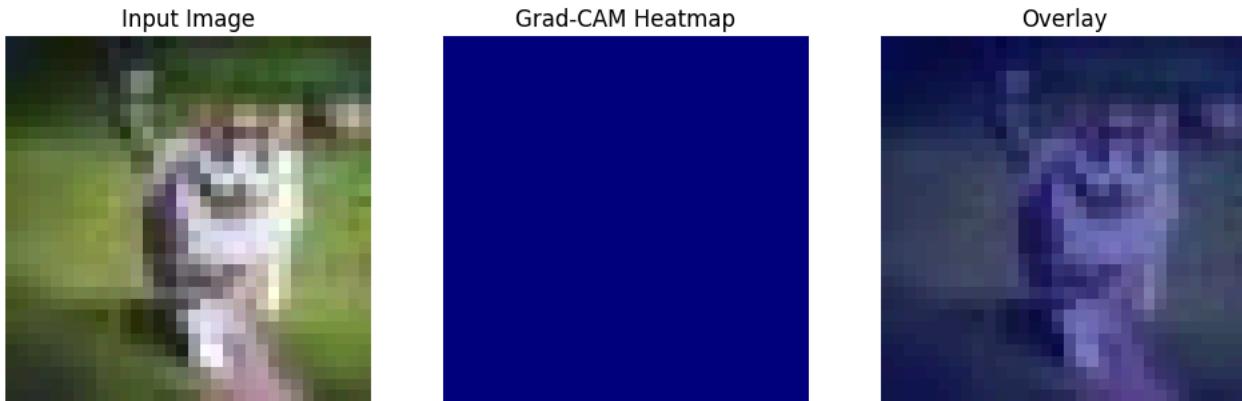
1/1 ————— 0s 35ms/step

True Label = REAL, Predicted Label = REAL (Confidence: 1.00)



1/1 ————— 0s 34ms/step

True Label = FAKE, Predicted Label = FAKE (Confidence: 0.00)



7. Feature Importance: LIME 0s 39ms/step

True Label = REAL, Predicted Label = REAL (Confidence: 1.00)

```

!pip install lime
import lime
import lime.lime_image
import numpy as np
from tensorflow.keras.preprocessing.image import img_to_array
from skimage.segmentation import mark_boundaries

# Assuming you have a trained model called 'model' and test data 'test_ds'
# We need a function that takes a batch of images and returns the model's prediction probabilities

def predict_fn(images):
    # LIME expects a numpy array of images
    # Ensure images are in the correct format and scale for your model
    # If your model was trained with rescaled images, rescale here too.
    # For this model, images are already rescaled [0, 1] by the dataset pipeline.
    return model.predict(images)

# Initialize LIME Image Explainer
# kernel_width: width of the kernel for the exponential similarity kernel
# feature_selection: method for selecting features (e.g., 'auto', 'forward_selection', 'lasso_path', 'none')
# random_state: seed for random number generator
explainer = lime.lime_image.LimeImageExplainer(random_state=42)

print("LIME explainer initialized.")

```

```

Collecting lime
  Downloading lime-0.2.0.1.tar.gz (275 kB)
    275.7/275.7 kB 6.6 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: matplotlib in /usr/local/lib/python3.12/dist-packages (from lime) (3.10.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.12/dist-packages (from lime) (2.0.2)
Requirement already satisfied: scipy in /usr/local/lib/python3.12/dist-packages (from lime) (1.16.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from lime) (4.67.1)
Requirement already satisfied: scikit-learn>=0.18 in /usr/local/lib/python3.12/dist-packages (from lime) (1.1.1)
Requirement already satisfied: scikit-image>0.12 in /usr/local/lib/python3.12/dist-packages (from lime) (0.19.2)
Requirement already satisfied: networkx>=3.0 in /usr/local/lib/python3.12/dist-packages (from scikit-image>0.12->lime) (3.6)
Requirement already satisfied: pillow>=10.1 in /usr/local/lib/python3.12/dist-packages (from scikit-image>0.12->lime) (11.3.0)
Requirement already satisfied: imageio!=2.35.0,>=2.33 in /usr/local/lib/python3.12/dist-packages (from scikit-image>0.12->lime)
Requirement already satisfied: tifffile>=2022.8.12 in /usr/local/lib/python3.12/dist-packages (from scikit-image>0.12->lime) (2)
Requirement already satisfied: packaging>=21 in /usr/local/lib/python3.12/dist-packages (from scikit-image>0.12->lime) (25.0)
Requirement already satisfied: lazy-loader>=0.4 in /usr/local/lib/python3.12/dist-packages (from scikit-image>0.12->lime) (0.4)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn>=0.18->lime) (1.5.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn>=0.18->lime) (1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>lime) (1.3.3)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.12/dist-packages (from matplotlib>lime) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.12/dist-packages (from matplotlib>lime) (4.60.1)
Requirement already satisfied: kiwisolver>1.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>lime) (1.4.9)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.12/dist-packages (from matplotlib>lime) (3.2.5)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.12/dist-packages (from matplotlib>lime) (2.9.0.po)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.7->matplotlib>lime)
Building wheels for collected packages: lime
  Building wheel for lime (setup.py) ... done
  Created wheel for lime: filename=lime-0.2.0.1-py3-none-any.whl size=283834 sha256=4c1c875b96e7c804fa6866e438a205409743c113b4aa
  Stored in directory: /root/.cache/pip/wheels/e7/5d/0e/4b4fff9a47468fed5633211fb3b76d1db43fe806a17fb7486a
Successfully built lime
Installing collected packages: lime
Successfully installed lime-0.2.0.1
LIME explainer initialized.

```

```

import matplotlib.pyplot as plt
import numpy as np

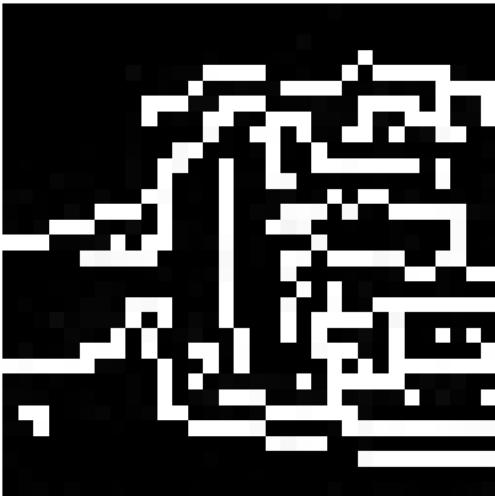
# Take one batch from the test dataset
for images, labels in test_ds.take(1):
    # Select the first image and its label from the batch
    sample_image = images[0].numpy()
    sample_label = labels[0].numpy()

    # Display the image
    plt.imshow(sample_image.astype("uint8"))
    plt.title(f"Sample Test Image (Label: {int(sample_label)}))")
    plt.axis("off")
    plt.show()
    break # Only take one image

```

```
/tmp/ipython-input-694662771.py:12: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will
plt.title(f"Sample Test Image (Label: {int(sample_label)})")
```

Sample Test Image (Label: 0)



```
# If needed:
# !pip install -q lime scikit-image

import numpy as np
import matplotlib.pyplot as plt
from lime import lime_image
from skimage.segmentation import mark_boundaries, slic
import tensorflow as tf

# -----
# 1) Choose which class to explain
# CIFAKE: FAKE=0, REAL=1
TARGET_CLASS = 0    # set to 1 to explain REAL

# -----
# 2) Prediction wrapper for LIME
def predict_proba_for_lime(imgs_uint8):
    """
    LIME passes uint8 images (N,H,W,C) in [0..255].
    Convert to model's input and return (N,2) probs [P(0), P(1)].
    """
    x = imgs_uint8.astype(np.float32) / 255.0    # adjust if you used different preprocessing
    p1 = model.predict(x, verbose=0).reshape(-1)    # sigmoid -> P(class=1)
    p0 = 1.0 - p1
    return np.stack([p0, p1], axis=1)

# -----
# 3) Segmentation function tuned for 32x32
def segmentation_fn(img):
    # Fewer superpixels work better on tiny images; tweak if needed
    return slic(img, n_segments=25, compactness=10, sigma=1, start_label=0)

# -----
# 4) Build explainer
explainer = lime_image.LimeImageExplainer()

# -----
# 5) Grab a small batch from validation
for images, labels in validation.take(1):
    imgs = images.numpy()                      # assume already [0,1]
    labs = labels.numpy().astype(int)
    imgs_u8 = (imgs * 255).astype(np.uint8)     # LIME expects uint8
    break

# -----
# 6) Explain first 10 validation images
N = min(10, len(imgs_u8))
for i in range(N):
    img_u8 = imgs_u8[i]
    true_lbl = 'REAL' if labs[i] == 1 else 'FAKE'
```

```
p1 = float(model.predict(imgs[i:i+1], verbose=0).ravel()[0])
pred_lbl = 'REAL' if p1 > 0.5 else 'FAKE'

exp = explainer.explain_instance(
    image=img_u8,
    classifier_fn=predict_proba_for_lime,
    top_labels=2,
    hide_color=None,           # keep original pixels outside selected superpixels
    num_samples=1000,          # ↑ for smoother explanations (slower)
    batch_size=64,
    segmentation_fn=segmentation_fn
)

# Positive evidence for TARGET_CLASS
# Returns (overlay_image, mask); we'll use the mask to draw boundaries on original img
_, mask_pos = exp.get_image_and_mask(
    label=TARGET_CLASS,
    positive_only=True,
    num_features=10,
    hide_rest=False
)

# Positive + Negative evidence
_, mask_all = exp.get_image_and_mask(
    label=TARGET_CLASS,
    positive_only=False,
    num_features=10,
    hide_rest=False
)

# -----
# 7) Visualize
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
fig.suptitle(f"True: {true_lbl} | Pred: {pred_lbl} (p1={p1:.2f}) | LIME → class {TARGET_CLASS}")

# Original
axes[0].imshow(img_u8)
axes[0].set_title("Input")
axes[0].axis("off")

# LIME (+ evidence) – boundaries on original image
axes[1].imshow(mark_boundaries(img_u8/255.0, mask_pos))
axes[1].set_title("LIME (+ evidence)")
axes[1].axis("off")

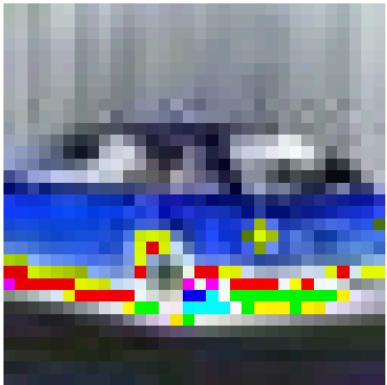
# LIME (+/- evidence) – boundaries on original image
axes[2].imshow(mark_boundaries(img_u8/255.0, mask_all))
axes[2].set_title("LIME (+/- evidence)")
axes[2].axis("off")

plt.tight_layout()
plt.show()
```

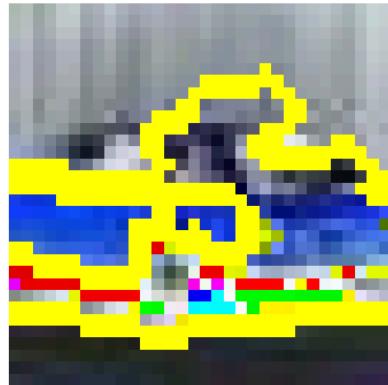

100%

1000/1000 [00:01<00:00, 853.72it/s]

Input

True: FAKE | Pred: FAKE (p1=0.00) | LIME → class 0
LIME (+ evidence)

LIME (+/- evidence)



100%

1000/1000 [00:01<00:00, 743.66it/s]

Input

True: REAL | Pred: REAL (p1=1.00) | LIME → class 0
LIME (+ evidence)

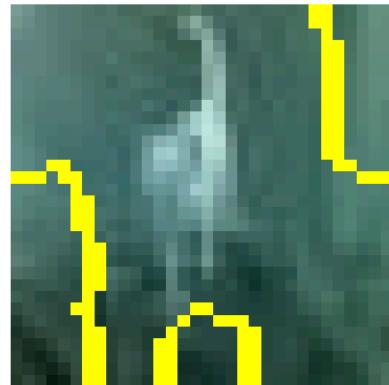
LIME (+/- evidence)



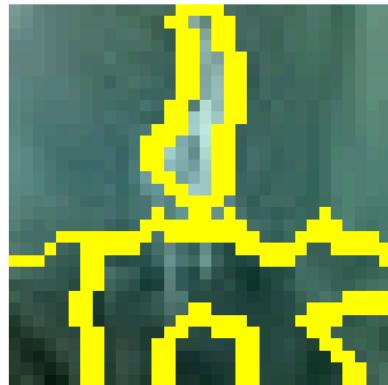
100%

1000/1000 [00:01<00:00, 717.51it/s]

Input

True: REAL | Pred: REAL (p1=1.00) | LIME → class 0
LIME (+ evidence)

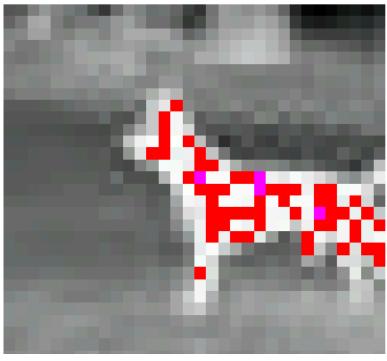
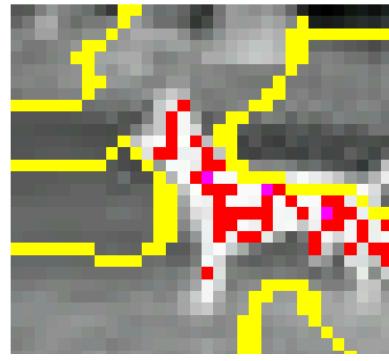
LIME (+/- evidence)



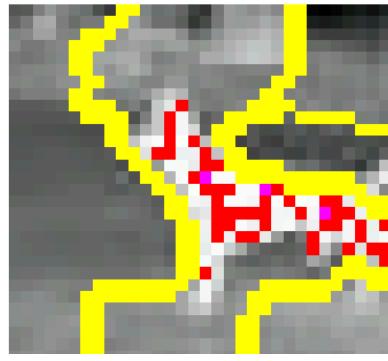
100%

1000/1000 [00:01<00:00, 813.67it/s]

Input

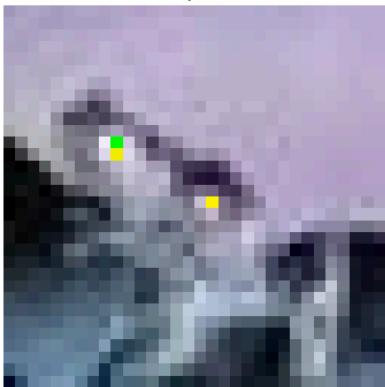
True: FAKE | Pred: FAKE (p1=0.00) | LIME → class 0
LIME (+ evidence)

LIME (+/- evidence)



100%

Input



1000/1000 [00:01<00:00, 826.91it/s]

True: REAL | Pred: REAL ($p1=1.00$) | LIME → class 0
LIME (+ evidence)

LIME (+/- evidence)



100%

Input

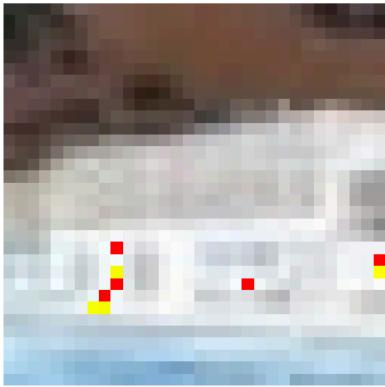
True: FAKE | Pred: FAKE ($p1=0.00$) | LIME → class 0
LIME (+ evidence)

LIME (+/- evidence)



100%

Input

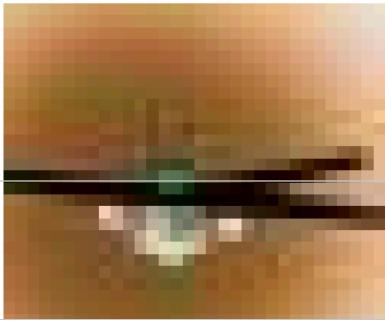
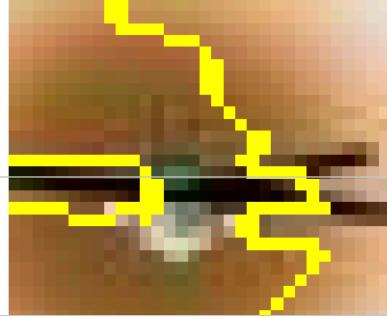
True: FAKE | Pred: REAL ($p1=0.88$) | LIME → class 0
LIME (+ evidence)

LIME (+/- evidence)

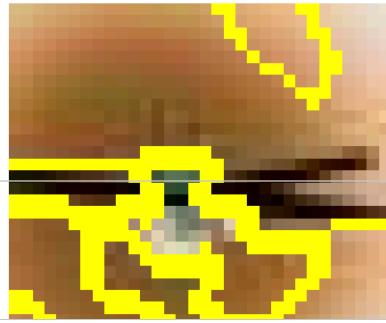


100%

Input

True: FAKE | Pred: FAKE ($p1=0.00$) | LIME → class 0
LIME (+ evidence)

LIME (+/- evidence)



```
from lime import lime_image
from skimage.segmentation import mark_boundaries
import matplotlib.pyplot as plt

# Reuse prediction wrapper & explainer from before
explainer = lime_image.LimeImageExplainer()

# Example: take first validation image
for images, labels in validation.take(1):
    img = (images[0].numpy() * 255).astype(np.uint8)    # ensure uint8
    true_label = int(labels[0].numpy())
    break

# Explain instance
explanation = explainer.explain_instance(
    image=img,
    classifier_fn=predict_proba_for_lime,
    top_labels=2,
    hide_color=None,
    num_samples=1000
)

# ---- Positive only (class = 0 or 1) ----
lime_img, mask = explanation.get_image_and_mask(
    label=0,                                # target class
    positive_only=True,
    num_features=5,
    hide_rest=False
)

# ---- Positive & negative evidence ----
lime_img_all, mask_all = explanation.get_image_and_mask(
    label=0,
    positive_only=False,
    num_features=5,
    hide_rest=False
)

# ---- Visualization ----
fig, axes = plt.subplots(1, 3, figsize=(12, 4))

# Original
axes[0].imshow(img)
axes[0].set_title("Input")
axes[0].axis("off")

# LIME (+ evidence)
axes[1].imshow(mark_boundaries(img/255.0, mask))
axes[1].set_title("LIME (+ evidence)")
axes[1].axis("off")

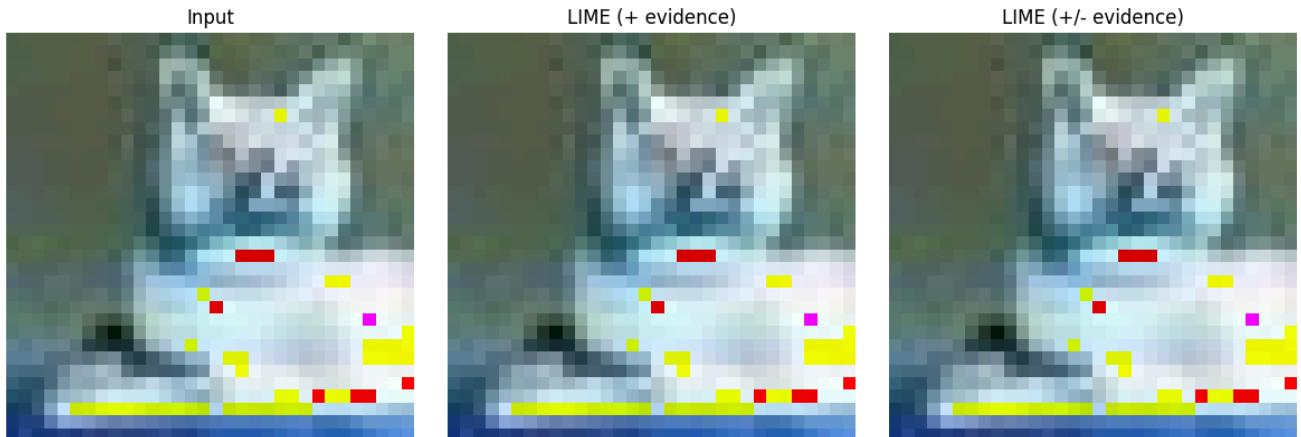
# LIME (+/- evidence)
axes[2].imshow(mark_boundaries(img/255.0, mask_all))
axes[2].set_title("LIME (+/- evidence)")
axes[2].axis("off")

plt.tight_layout()
plt.show()
```

```
/tmp/ipython-input-2870261098.py:11: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will
true_label = int(labels[0].numpy())
```

100%

1000/1000 [00:10<00:00, 134.49it/s]



```
# !pip install -q lime scikit-image

import numpy as np
import matplotlib.pyplot as plt
from lime import lime_image
from skimage.segmentation import slic
from skimage.color import gray2rgb

# --- choose class to explain: predicted (default) or fixed (e.g., FAKE=0 / REAL=1) ---
EXPLAIN_PREDICTED_CLASS = True
TARGET_CLASS = 0 # used only if EXPLAIN_PREDICTED_CLASS=False

# --- model wrapper (binary sigmoid -> 2-class probs) ---
def predict_proba_for_lime(imgs_uint8):
    x = imgs_uint8.astype(np.float32) / 255.0 # adjust if you used different preprocessing
    p1 = model.predict(x, verbose=0).reshape(-1) # P(class=1)
    p0 = 1.0 - p1
    return np.stack([p0, p1], axis=1)

# --- segmentation tuned for tiny images like 32x32 ---
def segmentation_fn(img):
    # n_segments ~ 20-60 works well for 32x32; tweak if needed
    return slic(img, n_segments=40, compactness=10, sigma=1, start_label=0)

explainer = lime_image.LimeImageExplainer()

# ----- get one validation image (uint8 for LIME) -----
for images, labels in validation.take(1):
    img_f = images[0].numpy() # [0,1] float
    if img_f.shape[-1] == 1: # safety: make RGB if grayscale
        img_f = gray2rgb(img_f[:, :, 0])
    img_u8 = (img_f * 255).astype(np.uint8)
    true_label = int(labels[0].numpy())
    break

# ----- decide label to explain -----
p1 = float(model.predict(img_f[None, ...], verbose=0).ravel()[0])
pred_label = 1 if p1 > 0.5 else 0
label_to_explain = pred_label if EXPLAIN_PREDICTED_CLASS else TARGET_CLASS

# ----- run LIME -----
exp = explainer.explain_instance(
    image=img_u8,
    classifier_fn=predict_proba_for_lime,
    top_labels=2,
    hide_color=None, # keep original pixels
    num_samples=2000, # ↑ for smoother explanations (slower)
    batch_size=64,
    segmentation_fn=segmentation_fn
```

```

)
# ----- build colored overlay from superpixel weights -----
# local_exp[label] gives list of (superpixel_id, weight)
segments = exp.segments
sp_weights = dict(exp.local_exp[label_to_explain])

# normalize weights for alpha
w_vals = np.array(list(sp_weights.values()))
if w_vals.size == 0:
    w_min, w_max = 0.0, 1.0
else:
    w_min, w_max = np.percentile(w_vals, [5, 95])
    if w_max <= w_min: w_max = w_min + 1e-6

overlay = img_f.copy()
alpha_map = np.zeros_like(segments, dtype=np.float32)

# color positive = green, negative = red
pos_color = np.array([0.0, 1.0, 0.0]) # RGB in 0..1
neg_color = np.array([1.0, 0.0, 0.0])

for sp_id, w in sp_weights.items():
    mask = (segments == sp_id)
    # map weight -> alpha (0..0.6)
    a = np.clip((abs(w) - w_min) / (w_max - w_min + 1e-6), 0, 1) * 0.6
    alpha_map[mask] = np.maximum(alpha_map[mask], a)
    color = pos_color if w > 0 else neg_color
    overlay[mask] = (1 - a) * overlay[mask] + a * color

# ----- plots: original, positive-only, full overlay with legend -----
from skimage.segmentation import mark_boundaries

pos_img, pos_mask = exp.get_image_and_mask(
    label=label_to_explain, positive_only=True, num_features=10, hide_rest=False
)
pos_boundaries = mark_boundaries(img_f, pos_mask)

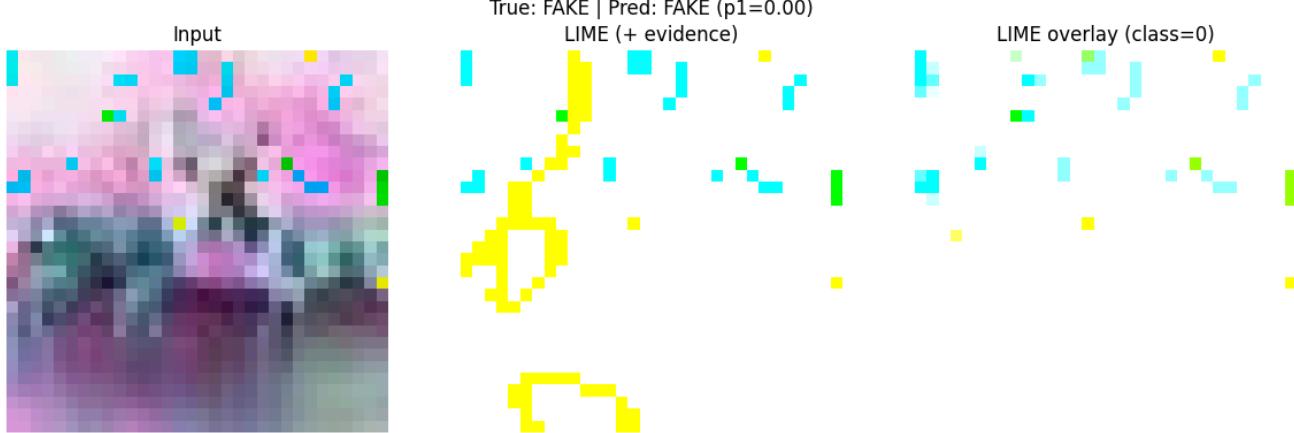
fig, axes = plt.subplots(1, 3, figsize=(12, 4))
axes[0].imshow(img_u8); axes[0].set_title("Input"); axes[0].axis("off")
axes[1].imshow(pos_boundaries); axes[1].set_title("LIME (+ evidence)"); axes[1].axis("off")
axes[2].imshow(overlay); axes[2].set_title(f"LIME overlay (class={label_to_explain})"); axes[2].axis("off")
plt.suptitle(f"True: {'FAKE', 'REAL'}[true_label] | Pred: {'FAKE', 'REAL'}[pred_label] (p1={p1:.2f})")
plt.tight_layout(); plt.show()

```

```

/tmp/ipython-input-1973020902.py:33: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will
true_label = int(labels[0].numpy())
100%                                         2000/2000 [00:02<00:00, 811.77it/s]
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)

```



```
# !pip install -q shap
import shap
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
```

```
# grab ~100 background examples (keep small for stability/speed)
bg_list = []
for batch_x, _ in train.take(4): # 4 * 32(batch) ≈ 128 images
    bg_list.append(batch_x.numpy())
background = np.concatenate(bg_list, axis=0)[:100] # (100, 32, 32, 3)

# grab a set of images to explain (e.g., 20 from validation)
test_list = []
test_labels = []
for batch_x, batch_y in validation.take(1):
    test_list.append(batch_x.numpy())
    test_labels.append(batch_y.numpy().astype(int))
X_explain = np.concatenate(test_list, axis=0)[:20] # (20, 32, 32, 3)
y_explain = np.concatenate(test_labels, axis=0)[:20].ravel()
```

```
def predict_p1(x):
    """Return probability of class 1 (REAL). Shape: (N,1)."""
    return model.predict(x, verbose=0)
```

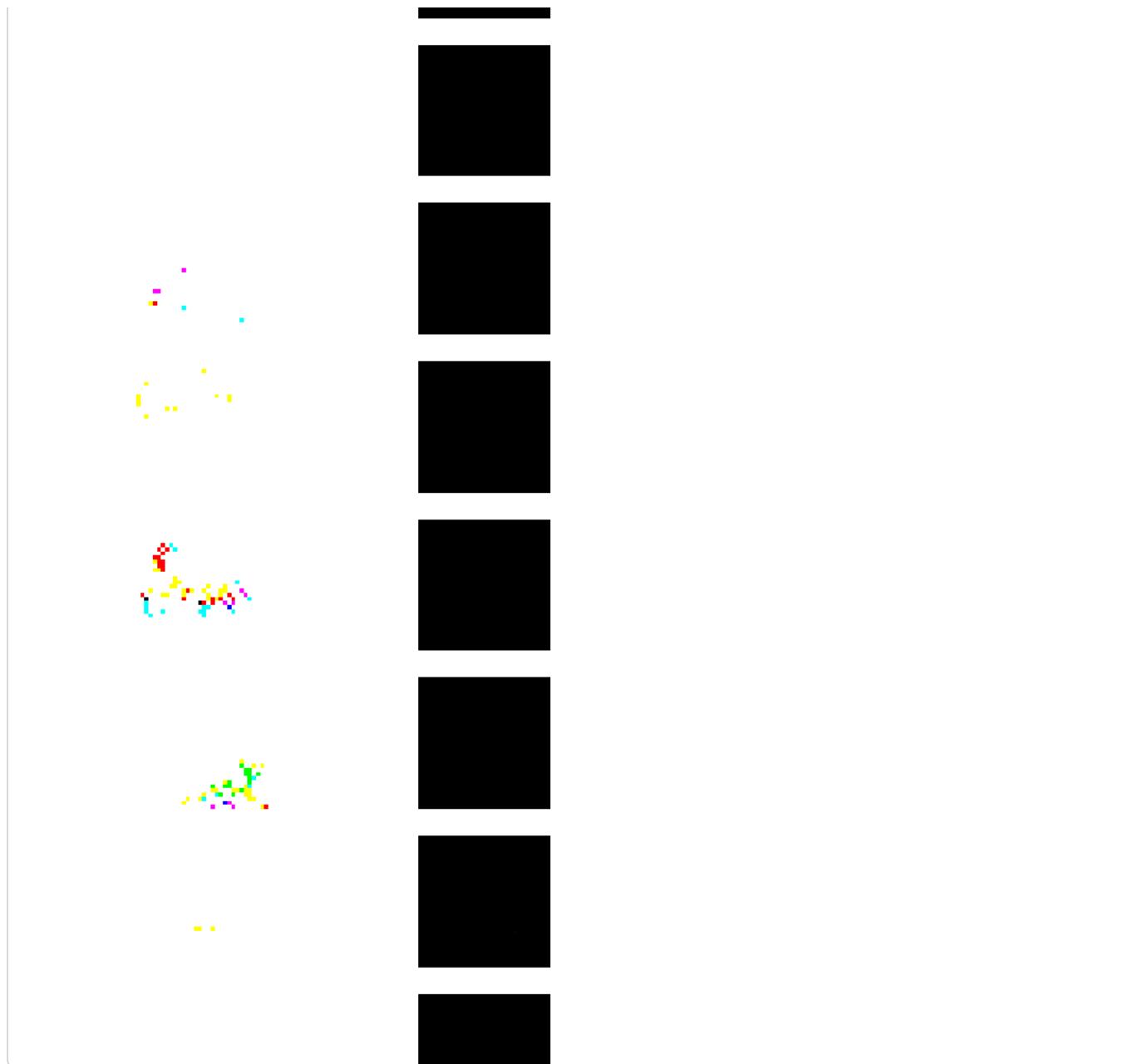
```
try:
    explainer = shap.GradientExplainer(model, background) # TF2/Keras supported
    # SHAP values for class 1 (REAL); shape: (N, H, W, C)
    shap_values_p1 = explainer.shap_values(X_explain) # returns array for single-output model
    # ensure it's a numpy array
    if isinstance(shap_values_p1, list):
        shap_values_p1 = shap_values_p1[0]
    print("GradientExplainer OK:", shap_values_p1.shape)
except Exception as e:
    print("GradientExplainer failed, falling back to KernelExplainer:", e)
    # KernelExplainer fallback (slower)
    f = lambda z: predict_p1(z).reshape(-1) # (N,) for KernelExplainer
    # downsample background further for speed
    background_small = background[:50]
    explainer = shap.KernelExplainer(f, background_small)
    # run on a few samples to keep it quick
    shap_values_p1 = explainer.shap_values(X_explain[:10], nsamples=200) # (10, H*W*C)
    # reshape back to image
    shap_values_p1 = np.array(shap_values_p1).reshape(-1, *X_explain.shape[1:])
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/models/functional.py:241: UserWarning: The structure of `inputs` doesn't match
Expected: keras_tensor_6
Received: inputs=['Tensor(shape=(20, 32, 32, 3))']
    warnings.warn(msg)
/usr/local/lib/python3.12/dist-packages/keras/src/models/functional.py:241: UserWarning: The structure of `inputs` doesn't match
Expected: keras_tensor_6
Received: inputs=['Tensor(shape=(50, 32, 32, 3))']
    warnings.warn(msg)
GradientExplainer OK: (20, 32, 32, 3, 1)
```

```
# SHAP expects images in [0,1]; ensure that's true for X_explain
X_disp = np.clip(X_explain, 0.0, 1.0)

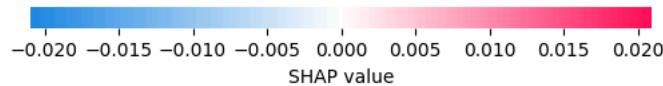
# Positive values (red) push toward REAL (class 1), negative (blue) push toward FAKE (class 0)
shap.image_plot([shap_values_p1], X_disp, show=True)
```



```
# Option A: negate to view contributions toward class 0
shap_values_p0 = -shap_values_p1
shap.image_plot([shap_values_p0], X_disp, show=True)

# Option B (alt): build a wrapper for class 0 and recompute
# def predict_p0(x): return 1.0 - predict_p1(x)
# explainer_0 = shap.GradientExplainer(lambda t: predict_p0(t), background)
# shap_values_p0 = explainer_0.shap_values(X_explain)
```



```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
WARNTNG:matplotlib.image:Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
```