Project ARI3205 Interpretable AI for Deep Learning Models *(Part 4.0)*

Name: Andrea Filiberto Lucas
ID No: 0279704L

Importing Necessary Libraries

```
import json
import os
import subprocess
import warnings
import logging
import absl.logging
#type: ignore

# Constants for colored output
COLORS = {
    "green": "\033[92m",  # Green text
    "red": "\033[91m",    # Red text
    "reset": "\033[0m"    # Reset to default color
}

# Path to the JSON file
lib_file_path = os.path.join("..", "Libraries", "Part4_Lib.json")

# Read the libraries from the JSON file
try:
    with open(lib_file_path, 'r') as file:
        libraries = json.load(file)
except FileNotFoundError:
    print(f"{COLORS['red']}Error: Library file not found at
{lib_file_path}{COLORS['reset']}")
    exit(1)
except json.JSONDecodeError:
    print(f"{COLORS['red']}Error: Failed to decode JSON from the
library file.{COLORS['reset']}")
    exit(1)

# Function to check and install libraries
def check_and_install_libraries(libraries):
    for lib, import_name in libraries.items():
        try:
            # Attempt to import the library
            __import__(import_name)
            print(f"[{COLORS['green']}✔{COLORS['reset']}] Library
```

```
'{lib}' is already installed.")
        except ImportError:
            # If import fails, try to install the library
            print(f"[{COLORS['red']}✖{COLORS['reset']}] Library
'{lib}' is not installed. Installing...")
            try:
                subprocess.check_call(["pip", "install", lib])
                print(f"[{COLORS['green']}✔{COLORS['reset']}]
Successfully installed '{lib}'.")
            except subprocess.CalledProcessError:
                print(f"[{COLORS['red']}✖{COLORS['reset']}] Failed to
install '{lib}'. Please install it manually.")

# Execute the function to check and install libraries
check_and_install_libraries(libraries)

# Suppress specific warnings
warnings.filterwarnings("ignore")

# Import necessary libraries for data analysis and modeling
import tensorflow as tf
#type: ignore
import numpy as np
#type: ignore
import random
#type: ignore
import matplotlib.pyplot as plt
#type: ignore
from tensorflow.keras.layers import (
#type: ignore
    Input, Conv2D, Dense, Flatten, Dropout, GlobalMaxPooling2D,
MaxPooling2D, BatchNormalization
)
from tensorflow.keras.preprocessing.image import ImageDataGenerator
#type: ignore
from tensorflow.keras.models import Model,  load_model
#type: ignore
from tf_explain.core.grad_cam import GradCAM  # Grad-CAM explainer
#type: ignore
from alibi.explainers import IntegratedGradients  # Integrated
Gradients explainer
#type: ignore
from alibi.utils.visualization import visualize_image_attr  #
Visualization function
#type: ignore

# Display TensorFlow version
print(f"TensorFlow Version: {tf.__version__}")
```

```
# Suppress specific warnings
warnings.filterwarnings("ignore")
absl.logging.set_verbosity(absl.logging.ERROR)
```

```
[✔] Library 'tensorflow' is already installed.
[✔] Library 'tensorflow_datasets' is already installed.
[✔] Library 'tf_explain' is already installed.
[✔] Library 'numpy' is already installed.
[✔] Library 'matplotlib' is already installed.
[✔] Library 'alibi' is already installed.
TensorFlow Version: 2.18.0
```

Loading and Preprocessing the CIFAR-10 Dataset

This script demonstrates how to load the CIFAR-10 dataset, split it into training and testing sets, normalize pixel values to the range [0, 1], and flatten label arrays for further use. It also prints dataset summaries at each step for clarity.

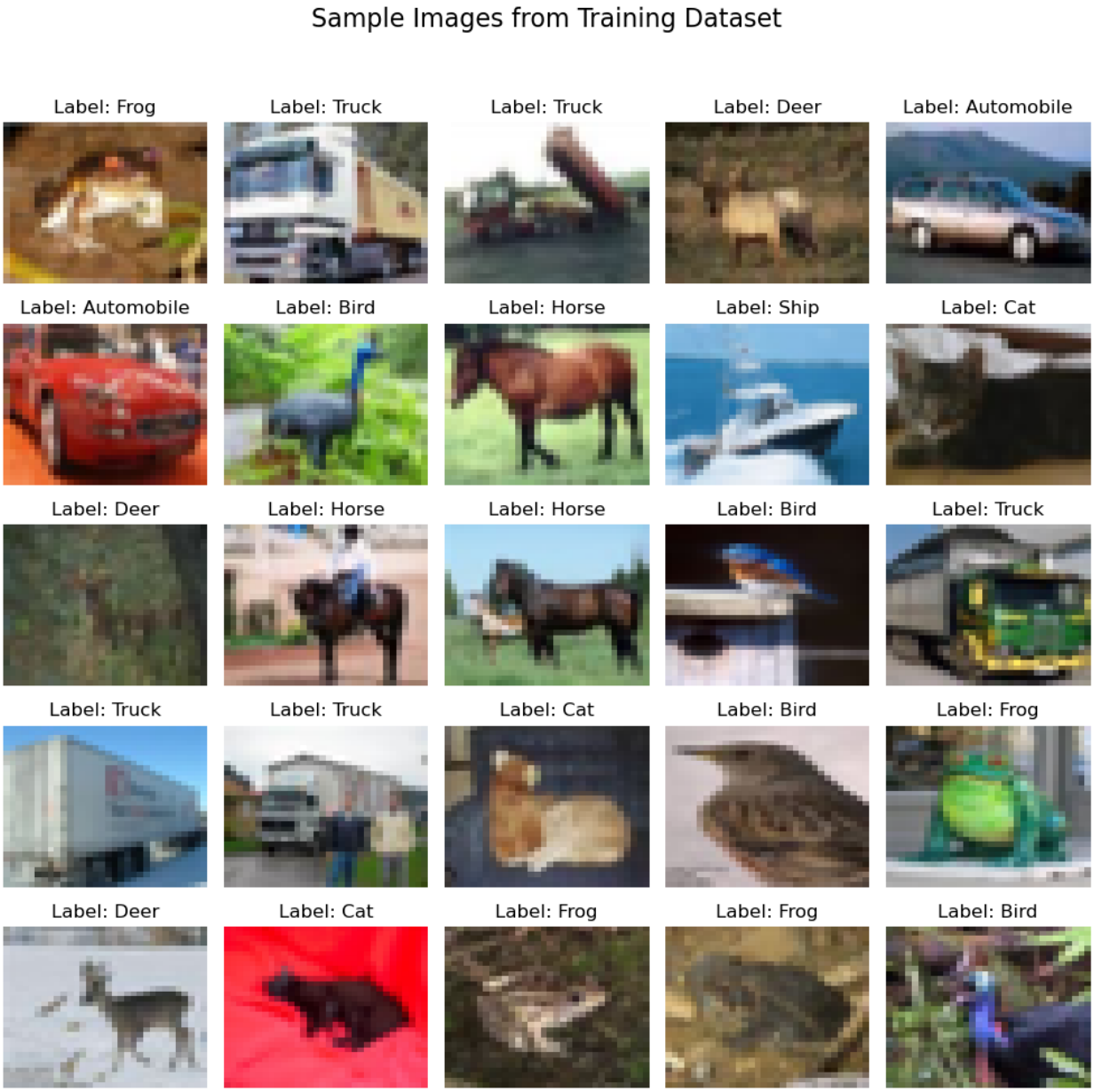
```
# Load the CIFAR-10 dataset
cifar10 = tf.keras.datasets.cifar10

# Split the dataset into training and testing sets
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
print(f"Training Data Shape: {x_train.shape}, Labels Shape:
{y_train.shape}")
print(f"Testing Data Shape: {x_test.shape}, Labels Shape:
{y_test.shape}")

# Normalize pixel values to the range [0, 1]
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Flatten the label arrays to 1D
y_train = y_train.flatten()
y_test = y_test.flatten()

# Print dataset summary after preprocessing
print(f"Normalized Training Data: {x_train.shape}, Labels:
{y_train.shape}")
print(f"Normalized Testing Data: {x_test.shape}, Labels:
{y_test.shape}")
```

```
Training Data Shape: (50000, 32, 32, 3), Labels Shape: (50000, 1)
Testing Data Shape: (10000, 32, 32, 3), Labels Shape: (10000, 1)
Normalized Training Data: (50000, 32, 32, 3), Labels: (50000,)
Normalized Testing Data: (10000, 32, 32, 3), Labels: (10000,)
```

Visualizing CIFAR-10 Dataset with Class Names

This script defines the CIFAR-10 class names and provides a visualization function to display sample images from the training dataset along with their corresponding class labels. It uses a grid layout for better clarity.

```
# Define CIFAR-10 class names
class_names = [
    "Airplane", "Automobile", "Bird", "Cat", "Deer",
    "Dog", "Frog", "Horse", "Ship", "Truck"
]

# Visualize sample images from the training dataset
def visualize_images(images, labels, num_rows=5, num_cols=5,
class_names=None):
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(10, 10))
    fig.suptitle("Sample Images from Training Dataset", fontsize=16)
    k = 0

    for i in range(num_rows):
        for j in range(num_cols):
            axes[i, j].imshow(images[k], aspect="auto")
            # Display the class name instead of numeric label
            label_name = class_names[labels[k]] if class_names else
labels[k]
            axes[i, j].set_title(f"Label: {label_name}")
            axes[i, j].axis("off")  # Turn off axes for clarity
            k += 1

    plt.tight_layout(rect=[0, 0, 1, 0.95])  # Adjust layout to fit the
title
    plt.show()

# Call the function to visualize images with class names
visualize_images(x_train, y_train, class_names=class_names)
```



Building a CNN Model for CIFAR-10 Classification

This script constructs a Convolutional Neural Network (CNN) using TensorFlow's Functional API to classify images in the CIFAR-10 dataset. The model includes three convolutional blocks, batch normalization, max-pooling layers, dropout regularization, and a fully connected output layer for class prediction. The number of classes is dynamically determined from the dataset.

```
# Number of classes based on the unique values in y_train
K = len(set(y_train))
print(f"Number of classes: {K}")

# Build the CNN model using the Functional API
```

```
# Input layer
input_layer = Input(shape=x_train[0].shape, name="Input_Layer")

# First Convolutional Block
x = Conv2D(32, (3, 3), activation='relu', padding='same',
name="Conv2D_Block1_Layer1")(input_layer)
x = BatchNormalization(name="BatchNorm_Block1_Layer1")(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same',
name="Conv2D_Block1_Layer2")(x)
x = BatchNormalization(name="BatchNorm_Block1_Layer2")(x)
x = MaxPooling2D((2, 2), name="MaxPool_Block1")(x)

# Second Convolutional Block
x = Conv2D(64, (3, 3), activation='relu', padding='same',
name="Conv2D_Block2_Layer1")(x)
x = BatchNormalization(name="BatchNorm_Block2_Layer1")(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same',
name="Conv2D_Block2_Layer2")(x)
x = BatchNormalization(name="BatchNorm_Block2_Layer2")(x)
x = MaxPooling2D((2, 2), name="MaxPool_Block2")(x)

# Third Convolutional Block
x = Conv2D(128, (3, 3), activation='relu', padding='same',
name="Conv2D_Block3_Layer1")(x)
x = BatchNormalization(name="BatchNorm_Block3_Layer1")(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same',
name="Conv2D_Block3_Layer2")(x)
x = BatchNormalization(name="BatchNorm_Block3_Layer2")(x)
x = MaxPooling2D((2, 2), name="MaxPool_Block3")(x)

# Flatten the output and add Dropout
x = Flatten(name="Flatten")(x)
x = Dropout(0.2, name="Dropout_Flatten")(x)

# Fully Connected Hidden Layer
x = Dense(1024, activation='relu', name="Dense_Hidden")(x)
x = Dropout(0.2, name="Dropout_Hidden")(x)

# Output Layer
output_layer = Dense(K, activation='softmax', name="Output_Layer")(x)

# Create the model
model = Model(inputs=input_layer, outputs=output_layer,
name="CIFAR10_CNN_Model")

# Print the model summary
model.summary()

Number of classes: 10
```

```
Model: "CIFAR10_CNN_Model"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                          ┃ Output Shape                  ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ Input_Layer (InputLayer)              │ (None, 32, 32, 3)             │               0 │
├───────────────────────────────────────┼───────────────────────────────┼─────────────────┤
│ Conv2D_Block1_Layer1 (Conv2D)         │ (None, 32, 32, 32)            │             896 │
├───────────────────────────────────────┼───────────────────────────────┼─────────────────┤
│ BatchNorm_Block1_Layer1               │ (None, 32, 32, 32)            │             128 │
│ (BatchNormalization)                  │                               │                 │
├───────────────────────────────────────┼───────────────────────────────┼─────────────────┤
│ Conv2D_Block1_Layer2 (Conv2D)         │ (None, 32, 32, 32)            │           9,248 │
├───────────────────────────────────────┼───────────────────────────────┼─────────────────┤
│ BatchNorm_Block1_Layer2               │ (None, 32, 32, 32)            │             128 │
│ (BatchNormalization)                  │                               │                 │
├───────────────────────────────────────┼───────────────────────────────┼─────────────────┤
│ MaxPool_Block1 (MaxPooling2D)         │ (None, 16, 16, 32)            │               0 │
├───────────────────────────────────────┼───────────────────────────────┼─────────────────┤
│ Conv2D_Block2_Layer1 (Conv2D)         │ (None, 16, 16, 64)            │          18,496 │
├───────────────────────────────────────┼───────────────────────────────┼─────────────────┤
│ BatchNorm_Block2_Layer1               │ (None, 16, 16, 64)            │             256 │
│ (BatchNormalization)                  │                               │                 │
├───────────────────────────────────────┼───────────────────────────────┼─────────────────┤
│ Conv2D_Block2_Layer2 (Conv2D)         │ (None, 16, 16, 64)            │          36,928 │
├───────────────────────────────────────┼───────────────────────────────┼─────────────────┤
```

```
| BatchNorm_Block2_Layer2          | (None, 16, 16, 64)      |
256 |
| (BatchNormalization)             |                         |
|
├──────────────────────────────────┼─────────────────────────┼──────
──────┤
| MaxPool_Block2 (MaxPooling2D)    | (None, 8, 8, 64)        |
0 |
├──────────────────────────────────┼─────────────────────────┼──────
──────┤
| Conv2D_Block3_Layer1 (Conv2D)    | (None, 8, 8, 128)       |
73,856 |
├──────────────────────────────────┼─────────────────────────┼──────
──────┤
| BatchNorm_Block3_Layer1          | (None, 8, 8, 128)       |
512 |
| (BatchNormalization)             |                         |
|
├──────────────────────────────────┼─────────────────────────┼──────
──────┤
| Conv2D_Block3_Layer2 (Conv2D)    | (None, 8, 8, 128)       |
147,584 |
├──────────────────────────────────┼─────────────────────────┼──────
──────┤
| BatchNorm_Block3_Layer2          | (None, 8, 8, 128)       |
512 |
| (BatchNormalization)             |                         |
|
├──────────────────────────────────┼─────────────────────────┼──────
──────┤
| MaxPool_Block3 (MaxPooling2D)    | (None, 4, 4, 128)       |
0 |
├──────────────────────────────────┼─────────────────────────┼──────
──────┤
| Flatten (Flatten)                | (None, 2048)            |
0 |
├──────────────────────────────────┼─────────────────────────┼──────
──────┤
| Dropout_Flatten (Dropout)        | (None, 2048)            |
0 |
├──────────────────────────────────┼─────────────────────────┼──────
──────┤
| Dense_Hidden (Dense)             | (None, 1024)            |
2,098,176 |
├──────────────────────────────────┼─────────────────────────┼──────
──────┤
| Dropout_Hidden (Dropout)         | (None, 1024)            |
0 |
├──────────────────────────────────┼─────────────────────────┼──────
──────┤
```

```
| Output_Layer (Dense)              | (None, 10)              |
10,250 |
└──────────────────────────────────┴─────────────────────────┴──────────
──────┘

 Total params: 2,397,226 (9.14 MB)

 Trainable params: 2,396,330 (9.14 MB)

 Non-trainable params: 896 (3.50 KB)
```

Model Selection and Training with Data Augmentation

This subsection outlines the process for selecting and training a model:

1. **Checking for Pre-Trained Models:**
   - The script searches the specified directory (`../Models`) for available `.h5` files.
   - If models are found, they are listed, and the user can choose to load one or train a new model.
2. **Training a New Model:**
   - If no models are available or the user opts to train a new model, they can specify the number of epochs.
   - The model is compiled with the Adam optimizer and sparse categorical cross-entropy loss.
   - Data augmentation is applied to the training dataset using `ImageDataGenerator` with random shifts and flips.
3. **Visualization and Model Saving:**
   - Training and validation accuracy and loss are plotted over epochs.
   - The trained model is saved in the `../Models` directory with the number of epochs in the filename.

This approach ensures flexibility in leveraging pre-trained models while enabling efficient training of new models with augmented data.

```python
# Define the directory to check for models
model_dir = os.path.join("..", "Models")

# List available models
available_models = [f for f in os.listdir(model_dir) if
f.endswith('.h5')]

if available_models:
    print("Available models:")
    print("0: Train a new model")
    for idx, model_name in enumerate(available_models, start=1):
        print(f"{idx}: {model_name}")

    # Prompt user to select a model
    selected_idx = int(input("Enter the number of the model to load"))
```

```
    if selected_idx > 0 and selected_idx <= len(available_models):
        # Load the selected model
        selected_model_path = os.path.join(model_dir,
available_models[selected_idx - 1])
        model = load_model(selected_model_path)
        print(f"'{selected_model_path}' loaded successfully.")
    else:
        print("Training a new model...")
        # Ask user for number of epochs
        epochs = int(input("Enter the number of epochs for training:
"))

        # Compile the model
        model.compile(
            optimizer='adam',
            loss='sparse_categorical_crossentropy',
            metrics=['accuracy']
        )

        # Define batch size
        batch_size = 32

        # Data Augmentation for Training Data
        data_generator = ImageDataGenerator(
            width_shift_range=0.1,  # Random horizontal shift
            height_shift_range=0.1,  # Random vertical shift
            horizontal_flip=True     # Random horizontal flip
        )

        # Create a data generator for the training dataset
        train_generator = data_generator.flow(x_train, y_train,
batch_size=batch_size)

        # Steps per epoch
        steps_per_epoch = x_train.shape[0] // batch_size

        # Train the model using augmented data
        history = model.fit(
            train_generator,
            validation_data=(x_test, y_test),
            steps_per_epoch=steps_per_epoch,
            epochs=epochs,
            verbose=1
        )

        # Plot training and validation accuracy
        plt.figure(figsize=(10, 5))
        plt.plot(history.history['accuracy'], label='Training
Accuracy')
```

```
        plt.plot(history.history['val_accuracy'], label='Validation
Accuracy')
        plt.title('Accuracy Over Epochs')
        plt.xlabel('Epochs')
        plt.ylabel('Accuracy')
        plt.legend()
        plt.show()

        # Plot training and validation loss
        plt.figure(figsize=(10, 5))
        plt.plot(history.history['loss'], label='Training Loss')
        plt.plot(history.history['val_loss'], label='Validation Loss')
        plt.title('Loss Over Epochs')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()
        plt.show()

        # Save the model with the number of epochs in the filename
        save_path = os.path.join(model_dir, f"AFL_{epochs}.h5")
        model.save(save_path)
        print(f"Model saved successfully at: {save_path}")
else:
    print("No models found. Training a new model...")

    # Ask user for number of epochs
    epochs = int(input("Enter the number of epochs for training: "))

    # Compile the model
    model.compile(
        optimizer='adam',
        loss='sparse_categorical_crossentropy',
        metrics=['accuracy']
    )

    # Define batch size
    batch_size = 32

    # Data Augmentation for Training Data
    data_generator = ImageDataGenerator(
        width_shift_range=0.1,  # Random horizontal shift
        height_shift_range=0.1,  # Random vertical shift
        horizontal_flip=True     # Random horizontal flip
    )

    # Create a data generator for the training dataset
    train_generator = data_generator.flow(x_train, y_train,
batch_size=batch_size)

    # Steps per epoch
```

```
    steps_per_epoch = x_train.shape[0] // batch_size

    # Train the model using augmented data
    history = model.fit(
        train_generator,
        validation_data=(x_test, y_test),
        steps_per_epoch=steps_per_epoch,
        epochs=epochs,
        verbose=1
    )

    # Save the model with the number of epochs in the filename
    save_path = os.path.join(model_dir, f"AFL_{epochs}.h5")
    model.save(save_path)
    print(f"Model saved successfully at: {save_path}")
```

```
Available models:
0: Train a new model
1: AFL_15T.h5
2: AFL_1T.h5
'../Models/AFL_15T.h5' loaded successfully.
```

Predicting a Random Test Image

A random image from the test dataset is displayed with its original label. The image is reshaped, and the model predicts its label, which is then compared to the original label.

```
# Select a random image from the test dataset
image_number = random.randint(0, x_test.shape[0] - 1)

# Display the selected image
plt.imshow(x_test[image_number])
plt.title(f"Original Label: {class_names[y_test[image_number]]}")
plt.axis("off")
plt.show()

# Load the selected image into an array
image_array = np.array(x_test[image_number])

# Reshape the image to match the input shape expected by the model
reshaped_image = image_array.reshape(1, 32, 32, 3)

# Predict the label using the model
predicted_label = class_names[model.predict(reshaped_image).argmax()]

# Load the original label
original_label = class_names[y_test[image_number]]

# Display the result
```

```
print(f"Original label: {original_label}")
print(f"Predicted label: {predicted_label}")
```


```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 187ms/step
Original label: Bird
Predicted label: Dog
```

Visualizing Model Explanations with Integrated Gradients

This section demonstrates how to use Integrated Gradients (IG) to explain model predictions:

1. **Integrated Gradients Setup:**
   - IG is initialized with parameters such as the number of steps, method, and internal batch size.
2. **Select a Test Instance:**
   - A random test image is selected from the dataset. Specific indices can be preferred for reproducibility.
3. **Generate Attributions:**
   - A baseline image (black) is used to calculate attributions for the selected instance, focusing on the true class label.

4. **Visualization:**
   - The original image and the attribution heatmap are visualized side by side. The heatmap highlights the regions of the image that contributed to the model's prediction.

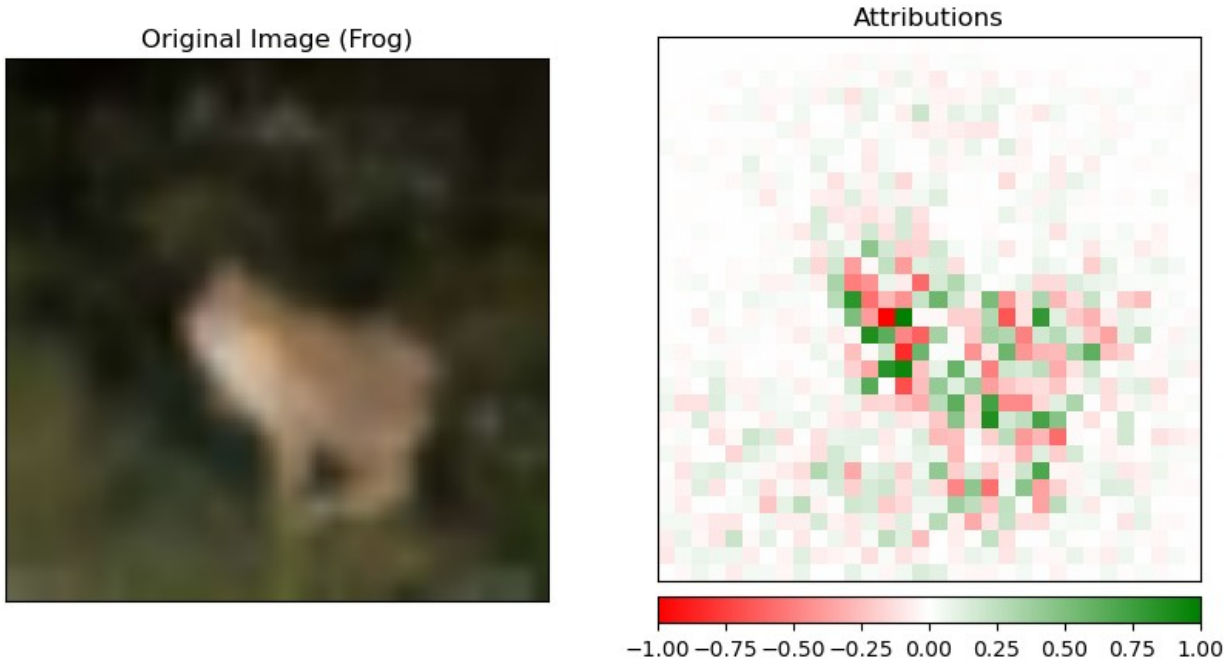
This approach helps to understand which parts of the image the model relies on for its decisions.

```
# Set up Integrated Gradients
n_steps = 50
method = "gausslegendre"
internal_batch_size = 50
ig = IntegratedGradients(
    model,
    n_steps=n_steps,
    method=method,
    internal_batch_size=internal_batch_size
)

# Select a random instance to explain
i = random.randint(0, len(x_test) - 1)  # Random index from the test
dataset
print(f"Selected instance index: {i}") # Preferred Indexes: 2864,
2003, 4028, 86644, 91, 2741, 5238, 5441, 98, 5113
instance = np.expand_dims(x_test[i], axis=0)

# Use the true class label as the target
true_label = int(y_test[i])  # Ensure the target is an integer

# Generate attributions using Integrated Gradients
baseline = np.zeros(instance.shape)  # Black image as baseline
explanation = ig.explain(instance, baselines=baseline,
target=true_label)
attrs = explanation.attributions[0]  # Get the attributions for the
selected instance

# Upscale the original image for better visualization (optional)
original_image = tf.image.resize(x_test[i], size=(128, 128)).numpy()

# Visualize original image and attributions
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 5))

# Visualize the original image
visualize_image_attr(
    attr=None,
    original_image=original_image,
    method='original_image',
    title=f'Original Image ({class_names[true_label]})',
    plt_fig_axis=(fig, ax[0]),
    use_pyplot=False
)
```

```
# Visualize the attributions
visualize_image_attr(
    attr=attrs.squeeze(),
    original_image=original_image,
    method='heat_map',
    sign='all',  # Show both positive and negative contributions
    show_colorbar=True,
    title='Attributions',
    plt_fig_axis=(fig, ax[1]),
    use_pyplot=True
)

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

```
Selected instance index: 5113
```

```
2025-01-07 17:03:15.919054: I
tensorflow/core/framework/local_rendezvous.cc:405] Local rendezvous is
aborting with status: OUT_OF_RANGE: End of sequence
```




```
<Figure size 640x480 with 0 Axes>
```

Discussion of the results obtained

The **Integrated Gradients (IG)** visualization for the selected instance (index 5113) provides a detailed explanation of the model's decision-making process. **IG** is a technique used to interpret

neural network predictions by attributing importance to input features (e.g., pixels in an image). It computes these attributions by accumulating gradients along a path from a baseline input, such as a black image, to the actual input. This method adheres to mathematical principles like **sensitivity** and **implementation invariance**, ensuring that the attributions are reliable and consistent. **IG** is particularly valuable for enhancing model interpretability by highlighting the features most influential to its output.

In the visualization, **green** and **red regions** signify the importance of specific pixels to the model's prediction. **Green areas** represent *positive contributions*, indicating that these regions increase the model's confidence in its classification. Conversely, **red areas** represent *negative contributions*, meaning they detract from the model's confidence. The attribution map's boxy nature arises from the pixelated structure of **CIFAR-10 images**, which are low resolution (32x32 pixels). Each pixel represents a significant portion of the image, and the blocky attributions align with this granularity, providing a meaningful coarse-grained analysis.

The results show that the model likely focused on specific **green-highlighted regions** that correspond to distinctive features of the frog, such as its body outline or texture. **Red regions**, often sparse and located near the edges or background, signify areas that are less relevant or even distracting to the model's prediction. The **high concentration of green pixels in the central area** of the image demonstrates that the model effectively prioritizes the object's most relevant features, aligning with human intuition.

This visualization exemplifies how **Integrated Gradients** enhances model transparency by offering a clear explanation of its decision-making process. By distinguishing between foreground and background elements and **de-emphasizing irrelevant features**, **IG** enables debugging and builds trust in deep learning models. Such insights are invaluable for improving model reliability, particularly in applications requiring high levels of interpretability.

Visualizing Model Explanations with Grad-CAM

This section illustrates how to use Grad-CAM to generate visual explanations for model predictions:

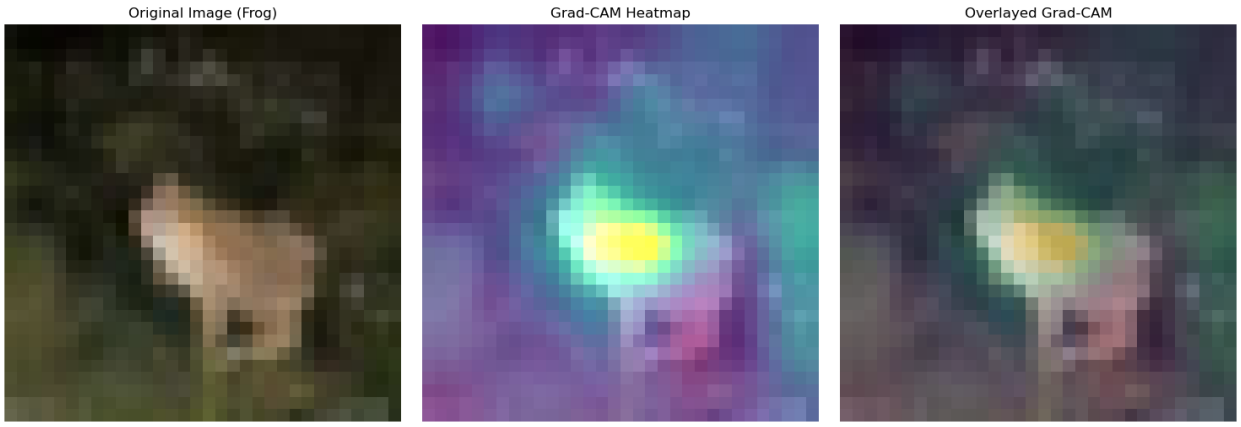
1. **Select a Test Instance:**
   - A random image from the test dataset is chosen, with the true class label identified for generating explanations.
2. **Target Layer Specification:**
   - The target convolutional layer (`Conv2D_Block3_Layer1`) is specified to focus on feature maps relevant to the prediction.
3. **Grad-CAM Initialization and Explanation:**
   - Grad-CAM is used to compute a heatmap that highlights areas of the image contributing to the prediction for the true class.
4. **Visualization:**
   - Three visualizations are presented:
     - The original image.
     - The Grad-CAM heatmap, which indicates important regions.
     - An overlay of the heatmap on the original image for better interpretability.

This technique helps to localize regions in the image that influence the model's decision-making process.

```
print(f"Selected instance index for Grad-CAM: {i}")

# Prepare the input instance and metadata
instance = np.expand_dims(x_test[i], axis=0)  # Expand dimensions for
model input
true_label = int(y_test[i])  # Convert true class to integer
original_image = x_test[i]  # Keep the original image for
visualization

# Specify the target convolutional layer
target_layer_name = "Conv2D_Block3_Layer1"  # Update to match your
model's layer name
target_layer = model.get_layer(target_layer_name)

# Initialize Grad-CAM
grad_cam = GradCAM()

# Generate the Grad-CAM explanation
explanation = grad_cam.explain(
    validation_data=(instance, None),  # Model input, no labels
required
    model=model,
    class_index=true_label,  # Target class for explanation
    layer_name=target_layer.name  # Specify target layer
)

# Process the heatmap
heatmap = np.maximum(explanation, 0)  # Clip negative values
heatmap = heatmap / np.max(heatmap)  # Normalize to [0, 1]

# Resize the heatmap to match the original image size
heatmap_resized = tf.image.resize(heatmap, (original_image.shape[0],
original_image.shape[1])).numpy()

# Create the overlayed image
overlayed_image = 0.6 * original_image + 0.4 * heatmap_resized  #
Blend original and heatmap

# Visualize the results
fig, ax = plt.subplots(1, 3, figsize=(15, 5))

# Display the original image
ax[0].imshow(original_image)
ax[0].set_title(f"Original Image ({class_names[true_label]})")
ax[0].axis("off")

# Display the Grad-CAM heatmap
```

```
ax[1].imshow(heatmap_resized, cmap="jet")
ax[1].set_title("Grad-CAM Heatmap")
ax[1].axis("off")

# Display the overlayed image
ax[2].imshow(overlayed_image)
ax[2].set_title("Overlayed Grad-CAM")
ax[2].axis("off")

# Adjust layout and display
plt.tight_layout()
plt.show()
```

```
Selected instance index for Grad-CAM: 5113
```



Discussion of the results obtained

The **Grad-CAM** visualization for the selected instance (index 5113) provides a clear understanding of the model's decision-making process. **Grad-CAM (Gradient-weighted Class Activation Mapping)** generates a heatmap highlighting the spatial regions of the input image that are most relevant to the model's prediction. This method leverages gradients flowing into the final convolutional layers to identify and visualize the areas of the image contributing the most to the predicted class. Unlike pixel-level attribution techniques like Integrated Gradients, Grad-CAM provides a broader, spatially coherent explanation by focusing on feature maps.

In the results, the **original image** on the left shows the input labeled as "Frog," while the **Grad-CAM heatmap** in the center highlights the areas of highest importance in yellow and green. The yellow regions represent the strongest positive contributions to the "Frog" prediction, primarily focusing on the central part of the image, aligning with the frog's body or distinctive features. The **overlayed Grad-CAM** image on the right combines the heatmap with the original image, offering a seamless representation of the model's focus areas. Background regions, such as those in the periphery, are de-emphasized (appearing darker or in blue), demonstrating that the model correctly disregards irrelevant parts of the image.

The Grad-CAM heatmap provides a coarse but spatially accurate attribution, with smoother transitions due to its operation on convolutional feature maps. This makes it less granular than

methods like Integrated Gradients but sufficient to highlight the critical regions influencing the model's decision. By focusing on the central areas of the image, the visualization confirms that the model is leveraging the frog's features, such as shape and texture, to make an informed classification.

The overlayed visualization enhances interpretability by merging the heatmap with the input image, making it evident how the model aligns its prediction with the visual features of the frog. This demonstration of Grad-CAM highlights its utility in debugging and validating convolutional models, as it provides valuable insights into the spatial reasoning behind the model's predictions.