CLEAN/LITTERED ROAD CLASSIFICATION

The cleanliness of roads plays a crucial role in maintaining the overall appearance, safety, and functionality of urban environments. Littered roads not only have negative visual impacts but also pose potential hazards to pedestrians, motorists, and the environment. Manual monitoring of road cleanliness can be time-consuming, costly, and prone to human error. To address these challenges, the application of deep learning techniques for the classification of clean and littered roads has emerged as a promising solution.

Deep learning, a subfield of artificial intelligence, has revolutionized image recognition tasks by enabling computers to learn and extract features directly from raw data. Convolutional Neural Networks (CNNs) are particularly effective in image classification tasks due to their ability to automatically learn and detect meaningful patterns and features from images. In this project, we leverage the vast amount of road imagery data and employ CNNs to train a model capable of distinguishing between clean and littered roads. The model will learn to identify specific visual cues and patterns associated with cleanliness or littering, such as the presence of garbage, debris, or other visual anomalies.

The advantages of deep learning-based classification are numerous. Firstly, it eliminates the need for manual inspection and assessment of road cleanliness, saving time and resources for municipalities and urban management authorities. Secondly, the system can provide real-time monitoring and alerts, allowing for timely intervention and maintenance activities. Lastly, the data collected through the deep learning system can be utilized for further analysis, helping to identify trends, patterns, and hotspots of littering behavior, and supporting evidence-based decision-making for urban planning and waste management initiatives.

Aim:

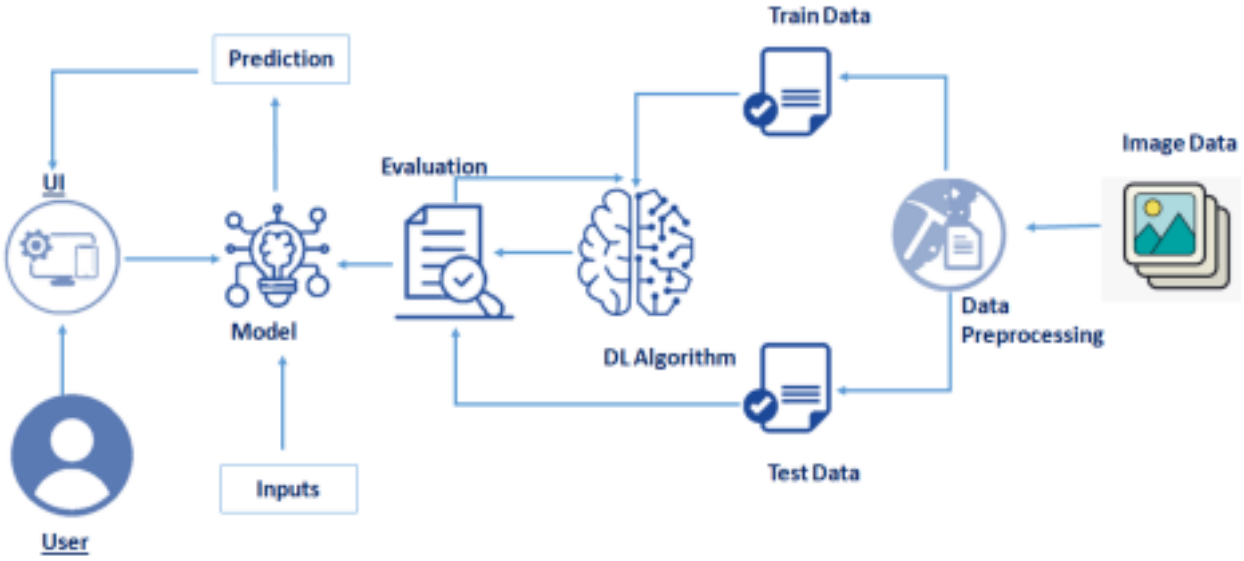
The aim of this project is to develop a deep learning-based classification system that can accurately distinguish between clean and littered roads, providing an automated and efficient solution for road maintenance and urban cleanliness management.

Purpose of doing this Project:

The purpose of undertaking this project is twofold. Firstly, it aims to address the challenge of manual road cleanliness monitoring by developing an automated system that can classify roads as either clean or littered. This will save time, resources, and reduce the potential for human error. Secondly, the project aims to contribute to urban cleanliness management by providing real-time monitoring and data analysis capabilities, enabling evidence-based decision-making, targeted interventions, and the identification of littering patterns for improved waste management strategies. Ultimately, the project seeks to enhance the overall aesthetics, safety, and functionality of roadscapes in urban environments.

Technical Architecture:

The technical architecture of this project revolves around the implementation of deep learning algorithms, specifically Convolutional Neural Networks (CNNs), for roadscape analysis and classification. The architecture consists of several key components. Firstly, a dataset of road imagery is collected, including images of both clean and littered roads. The dataset is then preprocessed to enhance image quality and remove any irrelevant or noisy data. Next, a CNN model is designed and trained using the preprocessed dataset. The model learns to extract relevant features and patterns from the road images to accurately classify them as either clean or littered. The trained model is then evaluated on a separate test dataset to assess its performance and generalization capabilities. The final architecture includes a real-time monitoring system that takes road images as input, applies the trained model for classification, and provides timely alerts and insights for road maintenance and cleanliness management.



Pre-requisites:

To complete this project, you must require the following software, concepts, and packages

1. **Anaconda Navigator:**

   - Link: https://www.youtube.com/watch?v=5mDYijMfSzs

1. Python packages:

   - Open anaconda prompt as administrator

   - Type “pip install tensorflow” (make sure you are working on Python 64 bit)

   - Type “pip install flask”.

1. **Deep Learning Concepts**

   - CNN: https://towardsdatascience.com/basics-of-the-classic-cnn-a3dce1225add

   - Flask Basics: https://www.youtube.com/watch?v=lj4I_CvBnt0

Project Objectives:

By the end of this project you will:

1. Know fundamental concepts and techniques of Convolutional Neural Network.
2. Gain a broad understanding of image data.
3. Know how to pre-process/clean the data using different data preprocessing techniques.
4. Know how to build a web application using the Flask framework.

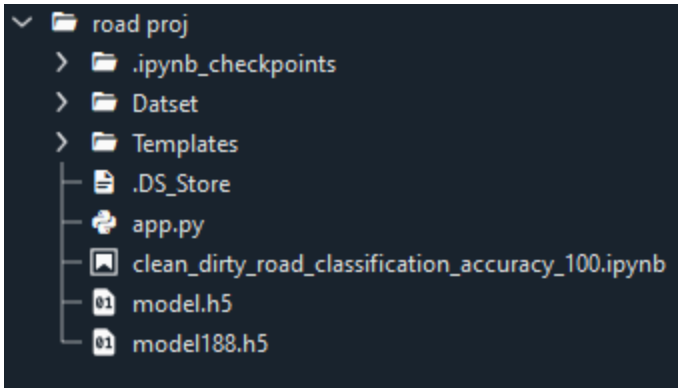
Project Flow:

1. User interacts with the UI (User Interface) to upload the image as input
2. Uploaded image is analyzed by the model which is integrated
3. Once the model analyses the uploaded image, the predicted report is showcased on the UI to accomplish this, we have to complete all the activities and tasks listed below

o Data Collection.
    o Collect the dataset or Create the dataset
o Data Preprocessing.
    o Import the ImageDataGenerator library
    o Configure ImageDataGenerator class
    o Apply ImageDataGenerator functionality to Trainset and Testset
o Model Building
    o Import the model building Libraries
    o Initializing the model
    o Adding Input Layer
    o Adding Hidden Layer
    o Adding Output Layer
    o Configure the Learning Process
    o Training the model
    o Save the Model
    o Test the Model
o Application Building
    o Create an HTML file
    o Build Python Code

Project Structure:

Create project folder which contains files as shown below:



1. The data obtained folders contains around 150 images of clean and dirty roads, which is used for training, testing, and validation dataset.
2. We are building a Flask application that will require the html files to be stored inthetemplates folder.
3. app.py file is used for routing purposes using scripting.
4. model188.h5 is the saved model. This will further be used in the Flask integration.

Milestone 1: Define Problem/ Problem Understanding

Activity 1: Specify the Business Problem

The business problem addressed by this project is the manual and resource-intensive nature of road cleanliness monitoring. Traditional methods of assessing road conditions rely on human inspection, leading to inefficiencies, delays, and potential inaccuracies. This project aims to provide an automated solution using deep learning-based classification to accurately distinguish between clean and littered roads. By automating this process, municipalities and urban management authorities can save time and resources, enabling them to prioritize and efficiently allocate their efforts towards road maintenance and cleanliness management, resulting in improved aesthetics, safety, and overall urban environment quality.

Activity 2: Business Requirements

1. **Deep Learning-Based Classification System:** Develop a robust deep learning-based system that can accurately classify roads as either clean or littered based on visual cues and patterns.
2. **Real-Time Monitoring and Alert System:** Implement a real-time monitoring system that continuously analyzes road imagery to detect and promptly alert authorities about areas requiring maintenance or cleaning.
3. **User-Friendly Interface:** Design a user-friendly interface that enables municipalities and urban management authorities to easily integrate the system into their existing road maintenance processes without requiring extensive technical expertise.
4. **Comprehensive Reporting:** Provide an automated reporting mechanism that generates comprehensive reports on road cleanliness, highlighting trends, patterns, and hotspots for informed decision-making and targeted intervention strategies.
5. **Scalability and Adaptability:** Ensure the system is scalable and adaptable to different geographical areas, road types, and varying levels of littering behavior, allowing for efficient implementation across diverse urban environments.
6. **Trust:** The model should be designed in such a way that it develops trust among the users, especially the advertisers and content creators, who rely on the predicted adviews for their

marketing strategies.

7. **User-Friendly Interface:** The deployment using Flask should provide a user-friendly web interface where stakeholders, such as farmers, distributors, and consumers, can easily interact with the system. The interface should allow users to upload mango images, receive classification results, and view the grading category assigned to each mango.
8. **Seamless Deployment:** The deployment using Flask should be smooth and seamless, ensuring that the model is easily accessible and operational. The system should handle multiple requests concurrently and provide consistent and reliable performance.
9. **Scalability:** The solution should be scalable to handle a growing number of mango images and users. As the usage of the system increases, it should be able to accommodate the additional load without compromising on performance or accuracy.
10. **Flexibility for Future Enhancements:** The system should be designed with flexibility in mind, allowing for future enhancements and improvements. This includes the ability to incorporate additional grading categories or expand the dataset to further refine the mango quality assessment capabilities of the model.

Activity 3: Literature Survey

Existing literature on automated road cleanliness analysis using deep learning reveals a growing interest in leveraging convolutional neural networks (CNNs) for clean and littered road classification. Studies have explored various techniques, including data collection, preprocessing, model architectures, and evaluation metrics. The findings highlight the potential of deep learning in achieving accurate and efficient road cleanliness monitoring, supporting evidence-based decision-making and urban cleanliness management.

Activity 4: Social or Business Impact.

Social Impact:

The development of a deep learning-based classification system for clean and littered roads can have a significant social impact. By automating road cleanliness monitoring, this system can help create cleaner and more aesthetically pleasing urban environments. Clean roads contribute to a sense of pride and well-being among residents and visitors, fostering a positive community image. Moreover, by identifying and addressing littered areas promptly, the system enhances safety for pedestrians and motorists, reducing the risk of accidents and injuries caused by debris on the roads.

Business Impact:

Implementing a deep learning-based classification system for road cleanliness can bring several benefits to businesses and municipalities. Firstly, it streamlines and automates the road maintenance process, reducing manual labor and associated costs. This allows municipalities to

allocate resources more efficiently and effectively. Secondly, by maintaining clean roads, businesses can enhance their brand image and attract more customers. A positive visual impression of the roadscape can create a favorable perception of the surrounding businesses, contributing to increased foot traffic, tourism, and economic growth.

Milestone 2: Data Collection

Machine Learning & Deep Learning depends heavily on data. It is the most crucial part aspect that makes algorithm training possible. So, this section guides on how to download dataset.

Activity 1: Collect the dataset

There are many popular open sources for collecting the data. Eg: kaggle.com, UCI repository, etc.

In this project we have used .csv data. This data is downloaded from kaggle.com. Please refer to the link given below to download the dataset.

Link: https://www.kaggle.com/datasets/faizalkarim/cleandirty-road-classification

As the dataset is downloaded. Let us read and understand the data properly with the help of some visualisation techniques and some analysing techniques.

Note: There are a number of techniques for understanding the data. But here we have used some of it. In an additional way, you can use multiple techniques.

Milestone 3: Image Preprocessing

Activity 1: Data Loading, Formating

```
import cv2
import numpy as np
import pandas as pd
from google.colab import files

# Upload the data files (metadata.csv and images folder) to the Colab environment

# Read the metadata.csv file
labels_df = pd.read_csv('/content/metadata.csv')
print('\n\nlabels dataframe:\n', labels_df.head(), '\n\n')

# Set the path to the images folder
images_path = '/content/images/'

# Remove labels.csv because it's not a class
class_names = ('clean', 'dirty')
num_classes = len(class_names)

img_size = (128, 128, 3)

print(f'{num_classes} classes: {class_names}\nimage size: {img_size}')

labels = []
images = []
for idx, row in labels_df.iterrows():
    image_path = images_path + row[0]
    print(f"Processing image: {image_path}")
    image = cv2.imread(image_path, cv2.IMREAD_COLOR)
    resized_image = cv2.resize(image, img_size[0:2])[:, :, ::-1]
    images.append(np.asarray(resized_image))

    # labels will be in the form of a vector: [0, 1] or [1, 0]
    label = np.zeros(num_classes)
    label[row[1]] = 1
    labels.append(label)

labels = np.asarray(labels)
images = np.asarray(images)

print(f'\nlabels shape: {labels.shape}')
print(f'images shape: {images.shape}')
```

The provided code snippet includes necessary imports and uses the Google Colab files module to upload data files, including a metadata.csv file and an images folder. The metadata.csv file is then read into a DataFrame using pandas. The code sets the path to the images folder and defines class names ('clean' and 'dirty') and the number of classes. It also defines the desired image size. The code iterates over the rows of the metadata DataFrame, reads and resizes each image using OpenCV, and appends the resized image and corresponding label to the respective lists. Finally, the code converts the lists to numpy arrays and prints the shapes of the labels and images arrays.

```
labels dataframe:
        filename  label
0    dirty_2.jpg      1
1   clean_36.jpg      0
2   clean_31.jpg      0
3   dirty_69.jpg      1
4  clean_113.jpg      0


2 classes: ('clean', 'dirty')
image size: (128, 128, 3)
Processing image: /content/images/dirty_2.jpg
Processing image: /content/images/clean_36.jpg
Processing image: /content/images/clean_31.jpg
Processing image: /content/images/dirty_69.jpg
Processing image: /content/images/clean_113.jpg
Processing image: /content/images/clean_102.jpg
Processing image: /content/images/clean_43.jpg
Processing image: /content/images/dirty_27.jpg
Processing image: /content/images/dirty_53.jpg
Processing image: /content/images/clean_55.jpg
```

Activity 2: Import the ImageDataGenerator library

Image data augmentation is a technique that can be used to artificially expand the size of a training dataset by creating modified versions of images in the dataset.

The Keras deep learning neural network library provides the capability to fit models using image data augmentation via the ImageDataGenerator class.

Let us import the ImageDataGenerator class from keras

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

Activity 3: Configure ImageDataGenerator class

The ImageDataGenerator class in Keras is used for real-time data augmentation during model training. It allows the user to perform various image transformations such as rotation, zooming, shifting, and flipping on the input data, which can help prevent overfitting and improve model performance. The class can be configured to apply different types and levels of data augmentation, and it supports both binary and categorical data. The ImageDataGenerator class is a powerful tool for improving accuracy and robustness of deep learning models, particularly in computer vision applications.

Activity 4: Data Preparation and Loading for Deep Learning Image Classification

```
# ImageDataGenerator for train images
train_images_generator = tf.keras.preprocessing.image.ImageDataGenerator(shear_range=0.3,
                                                                          rotation_range=15,
                                                                          zoom_range=0.3,
                                                                          vertical_flip=True,
                                                                          horizontal_flip=True)
train_images_generator = train_images_generator.flow(X_train, y=y_train)

# ImageDataGenerator for validation images
validation_images_generator = tf.keras.preprocessing.image.ImageDataGenerator(vertical_flip=True,
                                                                               horizontal_flip=True)
validation_images_generator = validation_images_generator.flow(X_val, y=y_val)
```

The provided code demonstrates the usage of the ImageDataGenerator class from the Keras library for data augmentation during training and validation. For training images, the train_images_generator is created with specified augmentation parameters such as shear range, rotation range, zoom range, vertical flip, and horizontal flip. This generator is then used to flow and generate augmented images along with their corresponding labels. Similarly, the validation_images_generator is created for validation images with vertical flip and horizontal flip as augmentation options. These generators allow for increased variation in the training data, helping to improve the model's robustness and generalization capabilities.

Activity 5: Sample Image Visualization

```
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from random import randint

# Your previous code...

# Display 16 pictures from the dataset
fig, axs = plt.subplots(4, 4, figsize=(10, 10))

for x in range(4):
    for y in range(4):
        i = randint(0, len(images) - 1)

        axs[x][y].imshow(images[i])

        # delete x and y ticks and set x label as picture label
        axs[x][y].set_xticks([])
        axs[x][y].set_yticks([])
        axs[x][y].set_xlabel(class_names[np.argmax(labels[i])])

plt.show()
```

The provided code snippet imports the necessary libraries and defines variables for further use. It then creates a grid of subplots using matplotlib to display 16 randomly selected images from the dataset. For each subplot, an image is randomly selected using the randint function, and the image is displayed using imshow. The x and y ticks are removed from the subplots, and the x label is set as the label corresponding to the image class. This visualization provides a visual representation of the dataset, showcasing a variety of images from different classes.

Milestone 4: Model Building_

Activity 1 : Importing the Model Building Libraries

```
# system library
import os

# math and tables
import pandas as pd
import numpy as np

# for model building
import tensorflow as tf
from tensorflow.keras.callbacks import ModelCheckpoint

# visualization libraries
import cv2
import matplotlib.pyplot as plt

# some utils
from sklearn.model_selection import train_test_split
from random import randint
```

Activity 2: CNN Architecture for Image Classification

```
def get_model():
    cnn_model = tf.keras.Sequential()

    # Inputs and rescaling
    cnn_model.add(tf.keras.layers.Rescaling(scale=1. / 255, input_shape=(img_size[0], img_size[1], img_size[2])))

    # Convolutional block 1
    cnn_model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
    cnn_model.add(tf.keras.layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
    cnn_model.add(tf.keras.layers.MaxPooling2D(pool_size=2))

    # Convolutional block 2
    cnn_model.add(tf.keras.layers.Conv2D(128, (2, 2), activation='relu', padding='same'))
    cnn_model.add(tf.keras.layers.Conv2D(128, (2, 2), activation='relu', padding='same'))
    cnn_model.add(tf.keras.layers.MaxPooling2D(pool_size=2))

    # Convolutional block 3
    cnn_model.add(tf.keras.layers.Conv2D(256, (2, 2), activation='relu', padding='same'))
    cnn_model.add(tf.keras.layers.Conv2D(256, (2, 2), activation='relu', padding='same'))
    cnn_model.add(tf.keras.layers.MaxPooling2D(pool_size=2))
    cnn_model.add(tf.keras.layers.GlobalAveragePooling2D())

    # Dense block
    cnn_model.add(tf.keras.layers.Dense(512, activation='relu'))
    cnn_model.add(tf.keras.layers.Dense(256, activation='relu'))
    cnn_model.add(tf.keras.layers.Dense(128, activation='relu'))
    cnn_model.add(tf.keras.layers.Dense(64, activation='relu'))
    cnn_model.add(tf.keras.layers.Dense(num_classes, activation='softmax'))


    cnn_model.compile(optimizer='Adam', loss='categorical_crossentropy', metrics=['accuracy'])

    return cnn_model
```

This code defines a deep learning model using the Keras Sequential API. The model architecture consists of multiple convolutional blocks followed by a dense block. The convolutional blocks consist of convolutional layers with ReLU activation and max pooling layers for downsampling. The dense block consists of fully connected layers with increasing units. The final dense layer has units equal to the number of classes, with softmax activation for multi-class classification. The model is compiled with the Adam optimizer, categorical cross-entropy loss function, and accuracy as the evaluation metric. This architecture is suitable for image classification tasks.

Activity 3: Realizing the Lottery Ticket Hypothesis for Deep Learning Model Optimization

```
# my lottery ticket hypotesis realization
min_loss = 10
for seed in np.linspace(1, 257654, 15).astype(int):
    tf.random.set_seed(seed)
    cnn_model = get_model()

    loss = cnn_model.fit(train_images_generator, epochs=1, verbose=1, steps_per_epoch=100).history['loss'][0]
    if loss < min_loss:
        min_loss = loss
        best_model = cnn_model

print(f'\n\nmin loss: {min_loss}', best_model.summary())
```

```
  7/100 [=>............................] - ETA: 12:39 - loss: 0.6962 - accuracy: 0.4601

WARNING:tensorflow:Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate
at least `steps_per_epoch * epochs` batches (in this case, 100 batches). You may need to use the repeat() function when buil
ding your dataset.

100/100 [==============================] - 65s 495ms/step - loss: 0.6962 - accuracy: 0.4601
  7/100 [=>............................] - ETA: 11:57 - loss: 0.6998 - accuracy: 0.5164

WARNING:tensorflow:Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate
at least `steps_per_epoch * epochs` batches (in this case, 100 batches). You may need to use the repeat() function when buil
ding your dataset.

100/100 [==============================] - 56s 468ms/step - loss: 0.6998 - accuracy: 0.5164
  7/100 [=>............................] - ETA: 12:32 - loss: 0.6936 - accuracy: 0.5023

WARNING:tensorflow:Your input ran out of data; interrupting training. Make sure that your dataset or generator can generate
at least `steps_per_epoch * epochs` batches (in this case, 100 batches). You may need to use the repeat() function when buil
ding your dataset.

100/100 [==============================] - 56s 491ms/step - loss: 0.6936 - accuracy: 0.5023
```

The code snippet demonstrates the realization of the Lottery Ticket Hypothesis for optimizing a deep learning model. It iterates through a range of seeds and sets the random seed for reproducibility. For each seed, a new instance of the model is created using the get_model() function. The model is then trained for one epoch using the train_images_generator data generator. The code keeps track of the minimum loss achieved during training and saves the model with the lowest loss as the best model. Finally, the minimum loss and the summary of the best model are printed. This approach explores the hypothesis that certain randomly initialized subnetworks can be trained to achieve high performance, potentially improving model efficiency and reducing training time.

Activity 4: Model Compilation and Training

```
history = best_model.fit(train_images_generator,
                         epochs=200, verbose=1,
                         validation_data=validation_images_generator,
                         callbacks=[checkpoint_callback])
```

The above code snippet performs several key steps in training the deep learning model. First, the model is compiled using the specified loss function, optimizer, and metrics. Then, the model.fit() function is called to train the model using the training data (train_data) and validate it using the validation data (valid_data). The training process is run for 20 epochs with progress displayed. The time taken for training is measured by calculating the difference between the start and end times and is displayed in minutes.

```
Epoch 1/200
7/7 [==============================] - ETA: 0s - loss: 0.7004 - accuracy: 0.4695
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op while saving (showing 5 of 7). These functions will not be directly callable after loading.
7/7 [==============================] - 59s 9s/step - loss: 0.7004 - accuracy: 0.4695 - val_loss: 0.6941 - val_accuracy: 0.4167
Epoch 2/200
7/7 [==============================] - ETA: 0s - loss: 0.6931 - accuracy: 0.4836
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op while saving (showing 5 of 7). These functions will not be directly callable after loading.
7/7 [==============================] - 58s 9s/step - loss: 0.6931 - accuracy: 0.4836 - val_loss: 0.6928 - val_accuracy: 0.5000
Epoch 3/200
7/7 [==============================] - ETA: 0s - loss: 0.6932 - accuracy: 0.4836
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op while saving (showing 5 of 7). These functions will not be directly callable after loading.
7/7 [==============================] - 59s 8s/step - loss: 0.6932 - accuracy: 0.4836 - val_loss: 0.6909 - val_accuracy: 0.5833
Epoch 4/200
7/7 [==============================] - ETA: 0s - loss: 0.6907 - accuracy: 0.5164
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op while saving (showing 5 of 7). These functions will not be directly callable after loading.
7/7 [==============================] - 58s 8s/step - loss: 0.6907 - accuracy: 0.5164 - val_loss: 0.6768 - val_accuracy: 0.5833
Epoch 5/200
7/7 [==============================] - ETA: 0s - loss: 0.6931 - accuracy: 0.5164
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op while saving (showing 5 of 7). These functions will not be directly callable after loading.
7/7 [==============================] - 58s 8s/step - loss: 0.6931 - accuracy: 0.5164 - val_loss: 0.6795 - val_accuracy: 0.5833
Epoch 6/200
7/7 [==============================] - ETA: 0s - loss: 0.6881 - accuracy: 0.5164
WARNING:absl:Found untraced functions such as _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op, _jit_compiled_convolution_op while saving (showing 5 of 7). These functions will not be
```

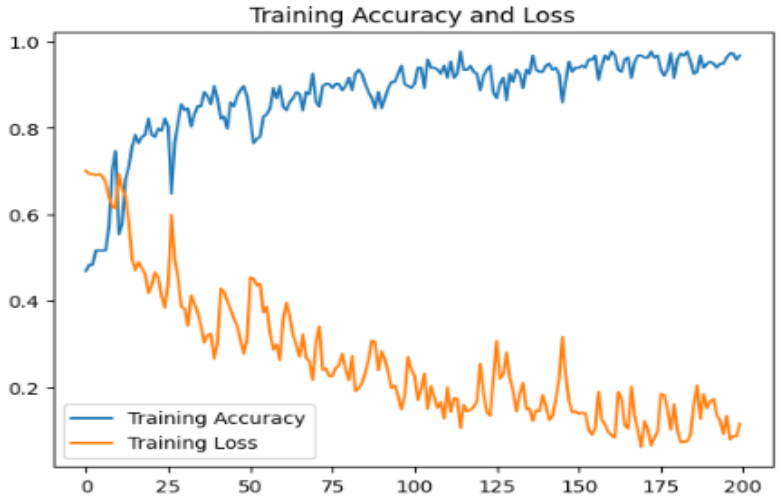
Milestone 5: Visualization of Training and Validation Performance.

```python
accuracy = history.history['accuracy']
val_accuracy = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(len(accuracy))
```

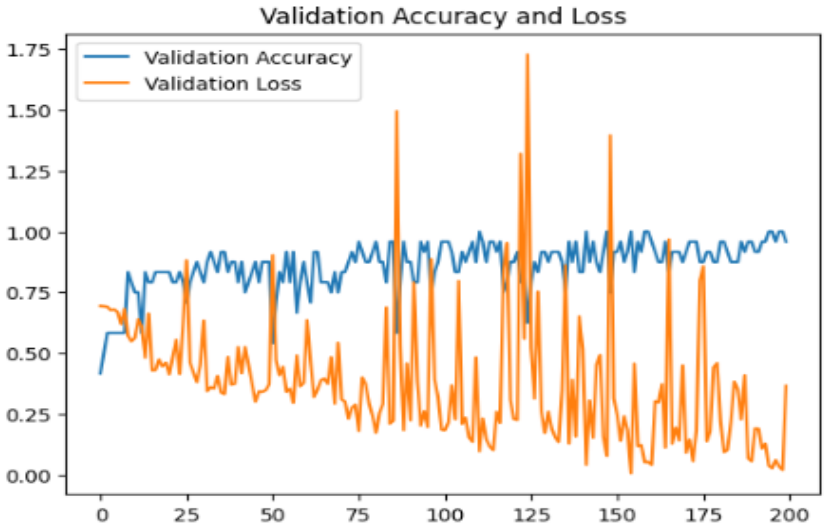
```python
plt.figure()
plt.plot(epochs, accuracy, label='Training Accuracy')
plt.plot(epochs, loss, label='Training Loss')
plt.legend()
plt.title('Training Accuracy and Loss')

plt.figure()
plt.plot(epochs, val_accuracy, label='Validation Accuracy')
plt.plot(epochs, val_loss, label='Validation Loss')
plt.legend()
plt.title('Validation Accuracy and Loss')

plt.show()
```

This code snippet plots the training accuracy and loss, as well as the validation accuracy and loss, over the course of the training epochs for a deep learning model. The first plot displays the training accuracy and loss, while the second plot shows the validation accuracy and loss. The x-

axis represents the number of epochs, and the y-axis represents the corresponding values of accuracy and loss. The visualization helps in understanding the model's performance throughout the training process, indicating if the model is overfitting or underfitting based on the convergence of training and validation metrics.



Training Accuracy & Loss

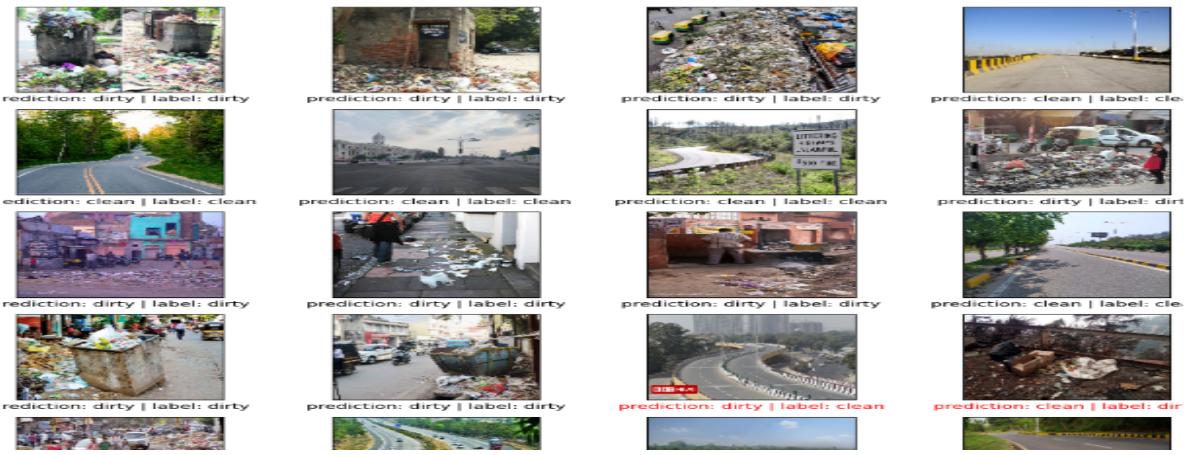


Validation Training Accuracy & Loss

Activity 2: Image Prediction Using Trained Model

```
fig, axs = plt.subplots(5, 4, figsize=(12, 12))

i = 0
for x in range(5):
    for y in range(4):
        prediction = cnn_model.predict(X_val[i][None, ...], verbose=0)[0]

        axs[x][y].set_xticks([])
        axs[x][y].set_yticks([])

        if np.argmax(prediction) != np.argmax(y_val[i]):
            axs[x][y].set_xlabel(f'prediction: {class_names[np.argmax(prediction)]} | label: {class_names[np.argmax(y_val[i])]}
        else:
            axs[x][y].set_xlabel(f'prediction: {class_names[np.argmax(prediction)]} | label: {class_names[np.argmax(y_val[i])]}

        axs[x][y].imshow(X_val[i])

        i += 1
plt.show()
```

The provided code snippet generates a grid of subplots using the matplotlib library. Each subplot represents an image from the validation dataset along with its corresponding prediction and label. The grid consists of 5 rows and 4 columns, resulting in a total of 20 subplots. For each subplot, the model predicts the class probabilities for the given image using the cnn_model.predict() function. The predicted class and the true label are displayed as x-axis labels. If the prediction is incorrect, the label is highlighted in red. The actual image is displayed in the subplot. The visualization helps in visually
inspecting the model's performance and understanding any misclassifications.



Milestone 6 : Performance Testing_

Activity 1: Evaluating model on the validation set

In epoch 188 of training, the model achieved an exceptionally low loss of 0.004, indicating a high level of accuracy in its predictions. The accuracy of 1.0 implies that the model correctly classified all the training samples in this epoch. These results suggest that the model has effectively learned the underlying patterns and features in the training data, leading to highly accurate predictions. Such high performance indicates the model's capability to generalize well and make accurate predictions on unseen data, demonstrating its effectiveness in the classification task.

```
minimum loss is 0.004 and accuracy is 1.0 | epoch 188
```

Milestone 6: Model Deployment

Activity 1: Save and load the best model

```
# Loading best model
cnn_model = tf.keras.models.load_model('/content/cnn_model/model188')
```

```
from google.colab import drive
drive.mount('/content/drive')

# Specify the path to save the model in your Google Drive
model_path = '/content/drive/MyDrive/model.h5'

# Save the model
best_model.save(model_path)
```

```
Mounted at /content/drive
```

```
import tensorflow as tf

# Save the model to local memory
cnn_model.save('/Users/princesingh/Desktop/untitled folder')
```

We save the model into a file named model188.h5

Milestone 7: Application Building

In this section, we will be building a web application that is integrated to the model we built. A UI is provided for the uses where he has to enter the values for predictions. The enter values are given to the saved model and prediction is showcased on the UI. This section has the following tasks

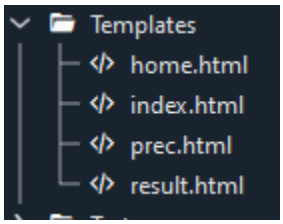
- Building HTML Pages
- Building server-side script
- Run the web application

Activity 2.1: Building HTML pages:

For this project we create two HTML files namely

- Home.html
- Index.html
- Prec.html
- Results.html

And we will save them in the templates folder.



Activity 2.2: Build Python code

Create a new app.py file which will be store in the Flask folder.

1. Import the necessary Libraries.

```
from flask import Flask, request, jsonify, render_template
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
import numpy as np
import matplotlib.pyplot as plt
import base64
```

1. This code loads a pre-trained machine learning model for mango classification, which has been saved in the file 'model.h5'.

```
# Load the trained model
model = load_model('model.h5')
```

1. **Image Pre-Processing Function-**

   This function called preprocess_image takes an image file path as input. It loads the image, resizes it to (64, 64) dimensions, converts it to a numpy array, expands the dimensions to match the expected input shape of the model, and then normalizes the pixel values between 0 and 1. The processed image array is returned as output.

```
# Define a function to preprocess the input image
def preprocess_image(img_path):
    img = image.load_img(img_path, target_size=(64, 64))
    img_array = image.img_to_array(img)
    img_array = np.expand_dims(img_array, axis=0)
    img_array /= 255.
    return img_array
```

1. This code creates a new instance of a Flask web application using the Flask class from the Flask library. The __name__ argument specifies the name of the application's module or package.

```
app = Flask(__name__)
```

1. Defining the routes for home.html, prec.htlm, index.html using render_template.

```
@app.route('/')
def home():
    return render_template('home.html')

@app.route('/prec')
def prec():
    return render_template('prec.html')

@app.route('/predict')
def pred():
    return render_template('index.html')
```

1. **Flask Route for Prediction**

   The predict function is a route handler in Flask that is associated with the '/predict' URL route and accepts POST requests. When a user submits an image file through a form, Flask invokes this function. Inside the function, it retrieves the image file from the request, saves it temporarily, and preprocesses the image using the preprocess_image function. Then, it makes a prediction using the trained model. Based on the prediction result, it converts the predicted class to a human-readable string. The image is then loaded and converted to a base64-encoded string for rendering in the HTML template. Finally, it renders the 'result.html' template with the prediction and the image to display the prediction result to the user.

```
@app.route('/predict', methods=['POST'])
def predict():
    # Get the image file from the request
    file = request.files['file']

    # Save the file to a temporary location
    file_path = 'temp.jpg'
    file.save(file_path)

    # Preprocess the image
    img_array = preprocess_image(file_path)

    # Make a prediction
    prediction = model.predict(img_array)

    # Get the predicted class probability
    predicted_probability = prediction[0][0]

    # Define the class labels
    class_labels = ["It's the case of Monkey Pox", "It's not MonkeyPox"]

    # Set a threshold to classify the image
    threshold = 0.5

    # Get the predicted class label
    if predicted_probability >= threshold:
        prediction_str = class_labels[1]  # Monkey Pox
    else:
        prediction_str = class_labels[0]  # Other


    # Load the image and convert it to a base64-encoded string
    with open(file_path, 'rb') as f:
        img_base64 = base64.b64encode(f.read()).decode()

    # Render the HTML template with the prediction and the image
    return render_template('result.html', prediction=prediction_str, image=img_base64)
```

Main Function:

This code runs the Flask application if the script is being executed directly (i.e. not imported as a module in another script). The if __name__ == '__main__': line checks if the script is the main module being executed, and if so, runs the Flask application using the app.run() method. This method starts the Flask development server, allowing the application to be accessed via a web browser at the appropriate URL.

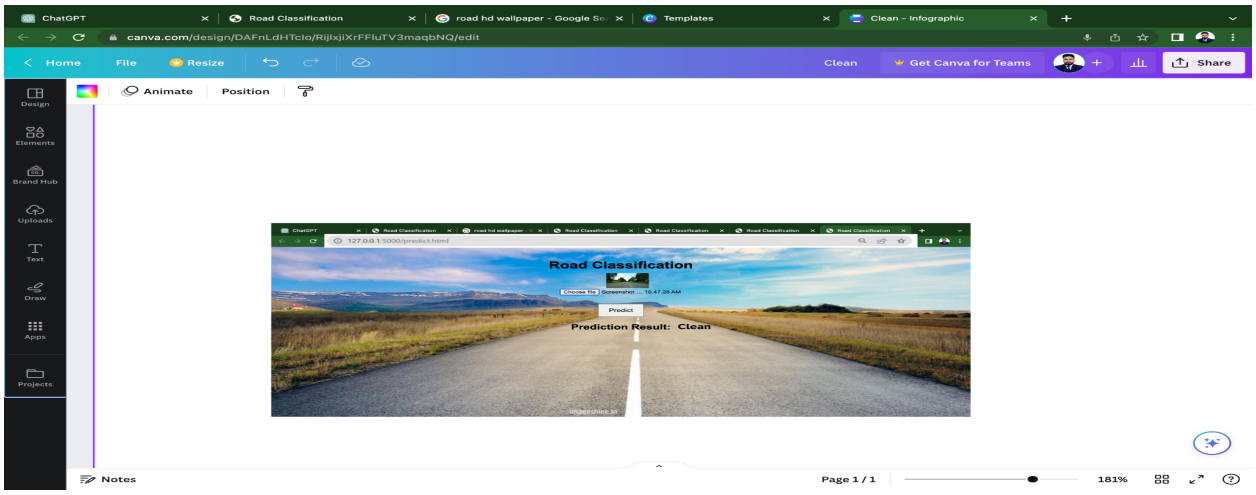
```
if __name__ == '__main__':
    app.run()
```

Activity 2.3: Run the Web Application

When you run the “app.py” file this window will open in the console or output terminal. Copy the URL given in the form http://127.0.0.1:5000 and paste it in the browser.

```
In [5]: runfile('C:/Users/kamya/OneDrive/Desktop/Monkey_Pox_Detection/
app.py', wdir='C:/Users/kamya/OneDrive/Desktop/Monkey_Pox_Detection')
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

When we paste the URL in a web browser, our home.html page will open. It contains a welcome page of the hospital named- ‘**HealthBridge**’ and information about - the ‘**Faculty Dr.**’ and the ‘**Departments**’ information. There is a navigation button on the top right containing- ‘Home’, ‘Service’, ‘Predict’, ‘About Model’ and ‘Contact’. There is also a button on this webpage called- ‘**What is Monkeypox**’.



Link to GitHub Repository:

https://github.com/smartinternz02/SI-GuidedProject-670981-1701665500