CSE485

Deep Learning

Project

Leaf Classification CNN

Project Documentation

Name	ID
Karim Bassel Samir Anby	20P6794
Matthew Sherif Shalaby	20P6785
Fady Fady Fouad	20P7341

Contents

Problem Definition 3
Code Screenshots 4
Outputs Screenshots 13
Model Description 16
Result Analysis 18

Problem Definition

Problem Definition

The task is to develop a deep learning-based solution for classifying leaves into various categories using image data. The goal is to build a robust neural network model that can effectively distinguish between different leaf species based on their images. This is part of the larger field of plant recognition, which has applications in ecology, agriculture, and biodiversity monitoring.

Objective:

1. **Data Preparation**: The first part of the project involves downloading and preparing the leaf dataset. This includes:
    - **Describing** the data to understand its structure and characteristics.
    - **Cleaning** the data by handling missing values, duplicates, or any inconsistencies.
    - **Visualizing** the dataset to understand its distribution and identify any potential patterns or anomalies.
    - Dividing the data into **training** and **test** sets with a ratio of 80% training data and 20% testing data.
    - **Standardizing** the data by computing the mean and standard deviation for each feature dimension and applying the transformation.
    - **Encoding** the labels into a format suitable for model training (e.g., one-hot encoding).
2. **Model Training**: The second part of the project focuses on designing a Convolutional Neural Network (CNN) that will classify the leaves:
    - Experiment with different **batch sizes**, **number of layers**, and **dropout rates** to find the best architecture and avoid overfitting.
    - Explore the use of different **optimizers** (e.g., SGD, Adam, RMSProp) to find the most effective method for training.
    - Implement **L2 regularization** (weight decay) and tune its hyperparameter to improve generalization.
    - Experiment with various **learning rates** and use a **learning rate scheduler** to dynamically adjust the learning rate during training.
3. **Model Evaluation**: After training the model, evaluate its performance on both the training and test datasets, focusing on accuracy. The performance of the model will be compared using different hyperparameter settings to identify the most effective configuration.

This project will contribute to advancing plant classification systems and has the potential for real-world applications in various fields, such as agriculture and conservation.

Code Screenshots

Imports

```
[2] from tensorflow.keras.models import Model
    from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Flatten, Dense, Dropout, Concatenate, BatchNormalization, GlobalAveragePooling2D
    from tensorflow.keras.optimizers import Adam
    from tensorflow.keras.callbacks import ModelCheckpoint
    import numpy as np
    from tensorflow.keras.preprocessing import image
    import matplotlib.pyplot as plt
    from sklearn.decomposition import PCA
    import pandas as pd
    from tensorflow.keras.regularizers import l2
    import tensorflow as tf
    from tensorflow.keras.optimizers import Adam, SGD, RMSprop
    import datetime
    from tensorflow.keras.callbacks import TensorBoard
    from sklearn.preprocessing import StandardScaler, LabelEncoder
    from sklearn.model_selection import train_test_split
    from tensorflow.keras.preprocessing import image
    import os
    from google.colab import drive
    import datetime
    drive.mount('/content/drive')

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

Describe Data

```
[3] train_data = pd.read_csv('train.csv')
    print(train_data.describe())
    print(train_data.info())

                id     margin1     margin2     margin3     margin4  \
count   990.000000  990.000000  990.000000  990.000000  990.000000
mean    799.595960    0.017412    0.028539    0.031988    0.023280
std     452.477568    0.019739    0.038855    0.025847    0.028411
min       1.000000    0.000000    0.000000    0.000000    0.000000
25%     415.250000    0.001953    0.001953    0.013672    0.005859
50%     802.500000    0.009766    0.011719    0.025391    0.013672
75%    1195.500000    0.025391    0.041016    0.044922    0.029297
max    1584.000000    0.087891    0.205080    0.156250    0.169920

          margin5     margin6     margin7     margin8     margin9  ...  \
count  990.000000  990.000000  990.000000  990.000000  990.000000  ...
mean     0.014264    0.038579    0.019202    0.001083    0.007167  ...
std      0.018390    0.052030    0.017511    0.002743    0.008933  ...
min      0.000000    0.000000    0.000000    0.000000    0.000000  ...
25%      0.001953    0.000000    0.005859    0.000000    0.001953  ...
50%      0.007812    0.015625    0.015625    0.000000    0.005859  ...
75%      0.017578    0.056153    0.029297    0.000000    0.007812  ...
max      0.111330    0.310550    0.091797    0.031250    0.076172  ...

        texture55   texture56   texture57   texture58   texture59   texture60  \
count  990.000000  990.000000  990.000000  990.000000  990.000000  990.000000
mean     0.036501    0.005024    0.015944    0.011586    0.016108    0.014017
std      0.063403    0.019321    0.023214    0.025040    0.015335    0.060151
min      0.000000    0.000000    0.000000    0.000000    0.000000    0.000000
25%      0.000000    0.000000    0.000977    0.000000    0.004883    0.000000
50%      0.004883    0.000000    0.005859    0.000977    0.012695    0.000000
75%      0.043701    0.000000    0.022217    0.009766    0.021484    0.000000
```

✓ 3s completed at 6:40 PM

Look For Null Values

```
print("\nMissing values in the dataset:")
print(train_data.isnull().sum())

Missing values in the dataset:
id           0
species      0
margin1      0
margin2      0
margin3      0
            ..
texture60    0
texture61    0
texture62    0
texture63    0
texture64    0
Length: 194, dtype: int64
```

Remove duplicates

```
# Check for duplicates
duplicates = train_data.duplicated()
print(f"\nNumber of duplicate rows: {duplicates.sum()}")

# Remove duplicates if any
data = train_data.drop_duplicates()
print(f"Dataset after removing duplicates: {data.shape}")
```

```
Number of duplicate rows: 0
Dataset after removing duplicates: (990, 194)
```

Load Data & Images

```
[6] # Path to the image folder
image_folder = '/content/drive/MyDrive/Leaf Classification Images/images'

# Function to load and preprocess images
def load_image(img_path, target_size=(192, 192)):
    img_full_path = os.path.join(image_folder, str(img_path) + '.jpg')  # Full path to the image
    img = image.load_img(img_full_path, target_size=target_size)
    img_array = image.img_to_array(img)
    img_array /= 255.0  # Normalize to [0, 1]
    return img_array

# Load and preprocess tabular data
#train_data = pd.read_csv('train.csv')  # Load your tabular data
image_ids = train_data['id']
labels = train_data['species']
tabular_features = train_data.drop(columns=['id', 'species'])
#apply correlation analysis and remove duplicate features
correlation_matrix = tabular_features.corr()
threshold=0.85
to_drop=set()
for i in range(len(correlation_matrix.columns)):
    for j in range(i):
        if abs(correlation_matrix.iloc[i, j]) > threshold:
            colname = correlation_matrix.columns[i]
            to_drop.add(colname)
tabular_features = tabular_features.drop(columns=to_drop)
print(f"Dropped Features: {to_drop}")

# Normalize tabular features
scaler = StandardScaler()
```

✓ 3s completed at 6:40 PM

```
        if abs(correlation_matrix.iloc[i, j]) > threshold:
            colname = correlation_matrix.columns[i]
            to_drop.add(colname)
tabular_features = tabular_features.drop(columns=to_drop)
print(f"Dropped Features: {to_drop}")

# Normalize tabular features
scaler = StandardScaler()
tabular_features = scaler.fit_transform(tabular_features)

# 3. Apply PCA
pca = PCA(n_components=0.95)  # Retain 95% of the variance
tabular_features = pca.fit_transform(tabular_features)

# Encode the labels
label_encoder = LabelEncoder()
labels_encoded = label_encoder.fit_transform(labels)

# Split data
X_train_img, X_test_img, y_train, y_test = train_test_split(image_ids, labels_encoded, test_size=0.2, random_state=42)
X_train_tabular, X_test_tabular = train_test_split(tabular_features, test_size=0.2, random_state=42)

# Load and preprocess images for training and testing
X_train_images = np.array([load_image(img) for img in X_train_img])
X_test_images = np.array([load_image(img) for img in X_test_img])
```

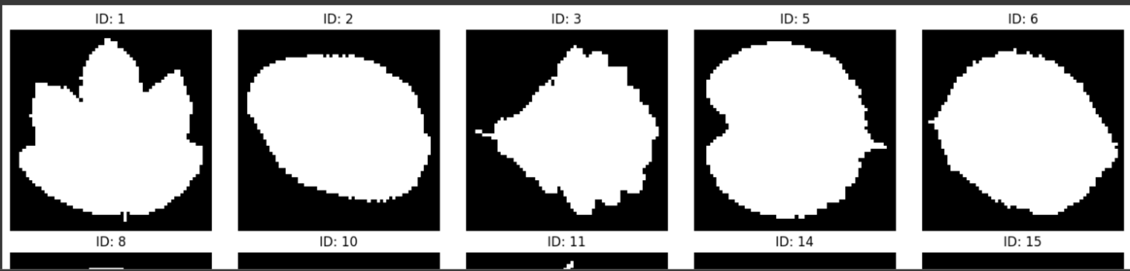
```
Dropped Features: {'shape48', 'shape34', 'shape37', 'shape10', 'shape53', 'shape6', 'shape52', 'shape60', 'shape55', 'shape57', 'shape25', 'shape46', 'shape13', 'sh
```

+ Code + Text

Display Some Images

```
[7] # Function to load and display an image
    def display_images(image_ids, rows=2, cols=5):
        fig, axes = plt.subplots(rows, cols, figsize=(15, 6))
        for i, img_id in enumerate(image_ids[:rows * cols]):
            img_path = os.path.join(image_folder, str(img_id) + '.jpg')
            img = image.load_img(img_path, target_size=(64, 64))
            axes[i // cols, i % cols].imshow(img)
            axes[i // cols, i % cols].axis('off')
            axes[i // cols, i % cols].set_title(f'ID: {img_id}')
        plt.tight_layout()
        plt.show()

    # Display the first 10 images
    display_images(data['id'])
```

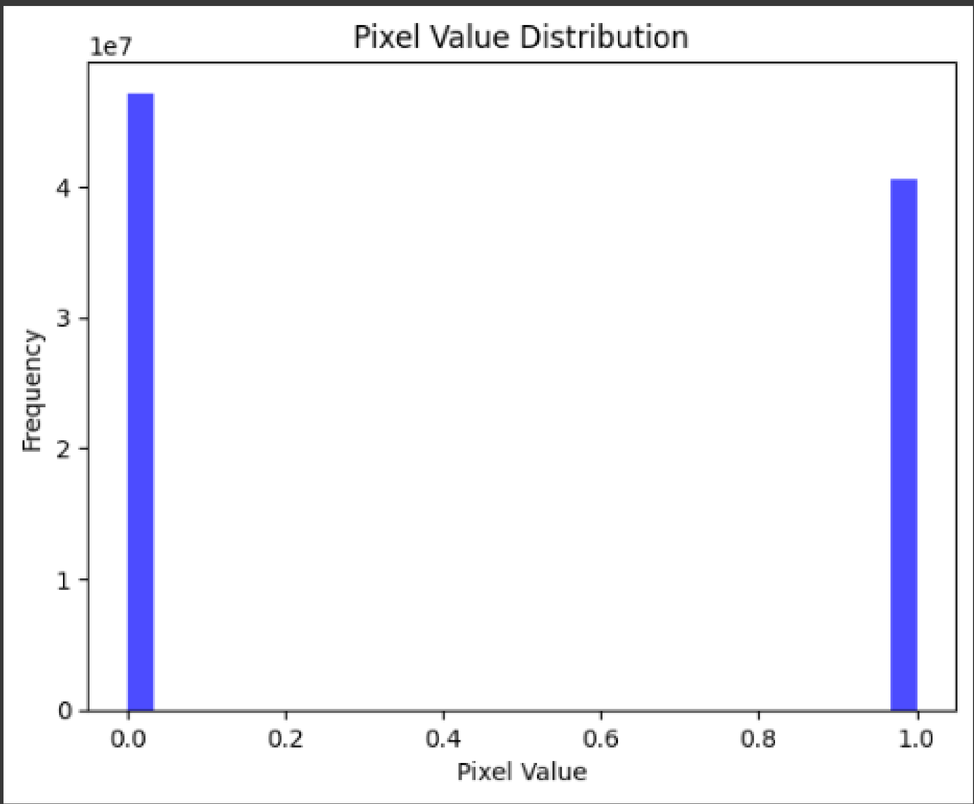


✓ 3s completed at 6:40 PM

Pixels Histogram

```
# Flatten all images to a 1D array
pixel_values = np.array(X_train_images).flatten()

# Plot histogram
plt.hist(pixel_values, bins=30, color='blue', alpha=0.7)
plt.title("Pixel Value Distribution")
plt.xlabel("Pixel Value")
plt.ylabel("Frequency")
plt.show()
```



2 Conv Layers

CNN Architecture

```python
def training(X_train_images, X_train_tabular, y_train, X_test_images, X_test_tabular, y_test, epochs, batch_size, learning_rate, dropout, optimizer_name, l2_factor):
    # Define the CNN model for image input
    image_input = Input(shape=(192, 192, 3))
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(image_input)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2, 2))(x)

    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2, 2))(x)

    x = GlobalAveragePooling2D()(x)
    x = Dropout(dropout)(x)

    # Define the Dense model for tabular input
    tabular_input = Input(shape=(X_train_tabular.shape[1],))
    y = Dense(128, activation='relu')(tabular_input)
    y = Dropout(dropout)(y)
    y = Dense(64, activation='relu')(y)
    y = Dropout(dropout)(y)

    # Combine image and tabular models
    combined = Concatenate()([x, y])
    z = Dense(512, activation='relu')(combined)
    z = Dropout(dropout)(z)
    z = Dense(256, activation='relu')(z)
    output = Dense(len(np.unique(y_train)), activation='softmax')(z)

    # Create the model
    model = Model(inputs=[image_input, tabular_input], outputs=output)

    # Define optimizer
    optimizers_dict = {
        'adam': Adam(learning_rate=learning_rate, weight_decay=l2_factor),
        'sgd': SGD(learning_rate=learning_rate, weight_decay=l2_factor),
        'rmsprop': RMSprop(learning_rate=learning_rate, weight_decay=l2_factor),
    }
    optimizer = optimizers_dict.get(optimizer_name.lower())
    if not optimizer:
        raise ValueError(f"Unknown optimizer '{optimizer_name}'. Available: {list(optimizers_dict.keys())}")

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

    # Model summary
    model.summary()
```

```python
    # Callbacks
    checkpoint_callback = ModelCheckpoint(filepath='best_model.keras', monitor='val_accuracy', save_best_only=True, verbose=1)
    log_dir = f"logs/fit/optimizer_{optimizer_name}_l2_{l2_factor}/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

    # Train the model
    history = model.fit(
        [X_train_images, X_train_tabular],
        y_train,
        epochs=epochs,
        batch_size=batch_size,
        validation_split=0.2,
        callbacks=[checkpoint_callback, tensorboard_callback]
    )

    # Evaluate the model
    train_loss, train_acc = model.evaluate([X_train_images, X_train_tabular], y_train, verbose=0)
    test_loss, test_acc = model.evaluate([X_test_images, X_test_tabular], y_test, verbose=0)
    print(f"Train Accuracy: {train_acc:.4f}, Test Accuracy: {test_acc:.4f}")
```

3 Conv Layers

```python
[9]
def training2(X_train_images, X_train_tabular, y_train, X_test_images, X_test_tabular, y_test, epochs, batch_size, learning_rate, dropout, optimizer_name, l2_factor):
    # Define the CNN model for image input
    image_input = Input(shape=(192, 192, 3))
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(image_input)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2, 2))(x)

    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2, 2))(x)

    x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2, 2))(x)

    x = GlobalAveragePooling2D()(x)
    x = Dropout(dropout)(x)

    # Define the Dense model for tabular input
    tabular_input = Input(shape=(X_train_tabular.shape[1],))
    y = Dense(128, activation='relu')(tabular_input)
    y = Dropout(dropout)(y)
    y = Dense(64, activation='relu')(y)
    y = Dropout(dropout)(y)

    # Combine image and tabular models
    combined = Concatenate()([x, y])
    z = Dense(512, activation='relu')(combined)
    z = Dropout(dropout)(z)
    z = Dense(256, activation='relu')(z)
    output = Dense(len(np.unique(y_train)), activation='softmax')(z)

    # Create the model
    model = Model(inputs=[image_input, tabular_input], outputs=output)

    # Define optimizer
    optimizers_dict = {
        'adam': Adam(learning_rate=learning_rate, weight_decay=l2_factor),
        'sgd': SGD(learning_rate=learning_rate, weight_decay=l2_factor),
        'rmsprop': RMSprop(learning_rate=learning_rate, weight_decay=l2_factor),
    }
    optimizer = optimizers_dict.get(optimizer_name.lower())
    if not optimizer:
        raise ValueError(f"Unknown optimizer '{optimizer_name}'. Available: {list(optimizers_dict.keys())}")

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

    # Model summary
    model.summary()

    # Callbacks
    checkpoint_callback = ModelCheckpoint(filepath='best_model.keras', monitor='val_accuracy', save_best_only=True, verbose=1)
    log_dir = f"logs/fit/optimizer_{optimizer_name}_l2_{l2_factor}/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
```

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

    # Model summary
    model.summary()

    # Callbacks
    checkpoint_callback = ModelCheckpoint(filepath='best_model.keras', monitor='val_accuracy', save_best_only=True, verbose=1)
    log_dir = f"logs/fit/optimizer_{optimizer_name}_l2_{l2_factor}/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

    # Train the model
    history = model.fit(
        [X_train_images, X_train_tabular],
        y_train,
        epochs=epochs,
        batch_size=batch_size,
        validation_split=0.2,
        callbacks=[checkpoint_callback, tensorboard_callback]
    )

    # Evaluate the model
    train_loss, train_acc = model.evaluate([X_train_images, X_train_tabular], y_train, verbose=0)
    test_loss, test_acc = model.evaluate([X_test_images, X_test_tabular], y_test, verbose=0)
    print(f"Train Accuracy: {train_acc:.4f}, Test Accuracy: {test_acc:.4f}")
```

4 Conv Layers

[9]

```python
def training3(X_train_images, X_train_tabular, y_train, X_test_images, X_test_tabular, y_test, epochs, batch_size, learning_rate, dropout, optimizer_name, l2_factor):
    # Define the CNN model for image input
    image_input = Input(shape=(192, 192, 3))
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(image_input)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2, 2))(x)

    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2, 2))(x)

    x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2, 2))(x)

    x = Conv2D(512, (3, 3), activation='relu', padding='same')(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D((2, 2))(x)

    x = GlobalAveragePooling2D()(x)
    x = Dropout(dropout)(x)

    # Define the Dense model for tabular input
    tabular_input = Input(shape=(X_train_tabular.shape[1],))
    y = Dense(128, activation='relu')(tabular_input)
    y = Dropout(dropout)(y)
    y = Dense(64, activation='relu')(y)
    y = Dropout(dropout)(y)

    # Combine image and tabular models
    combined = Concatenate()([x, y])
    z = Dense(512, activation='relu')(combined)
    z = Dropout(dropout)(z)
    z = Dense(256, activation='relu')(z)
    output = Dense(len(np.unique(y_train)), activation='softmax')(z)

    # Create the model
    model = Model(inputs=[image_input, tabular_input], outputs=output)

    # Define optimizer
    optimizers_dict = {
        'adam': Adam(learning_rate=learning_rate, weight_decay=l2_factor),
        'sgd': SGD(learning_rate=learning_rate, weight_decay=l2_factor),
        'rmsprop': RMSprop(learning_rate=learning_rate, weight_decay=l2_factor),
    }
    optimizer = optimizers_dict.get(optimizer_name.lower())
    if not optimizer:
        raise ValueError(f"Unknown optimizer '{optimizer_name}'. Available: {list(optimizers_dict.keys())}")

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

    # Model summary
    model.summary()
```


```python
    model.compile(optimizer=optimizer, loss='sparse_categorical_crossentropy', metrics=['accuracy'])

    # Model summary
    model.summary()
```


```python
    # Callbacks
    checkpoint_callback = ModelCheckpoint(filepath='best_model.keras', monitor='val_accuracy', save_best_only=True, verbose=1)
    log_dir = f"logs/fit/optimizer_{optimizer_name}_l2_{l2_factor}/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
    tensorboard_callback = TensorBoard(log_dir=log_dir, histogram_freq=1)

    # Train the model
    history = model.fit(
        [X_train_images, X_train_tabular],
        y_train,
        epochs=epochs,
        batch_size=batch_size,
        validation_split=0.2,
        callbacks=[checkpoint_callback, tensorboard_callback]
    )

    # Evaluate the model
    train_loss, train_acc = model.evaluate([X_train_images, X_train_tabular], y_train, verbose=0)
    test_loss, test_acc = model.evaluate([X_test_images, X_test_tabular], y_test, verbose=0)
    print(f"Train Accuracy: {train_acc:.4f}, Test Accuracy: {test_acc:.4f}")
```

Try Different Learning Rates

```python
# List of learning rates to experiment with
learning_rates = [0.0001, 0.001, 0.01]

# Loop through each learning rate and train the model
for lr in learning_rates:
    print(f"Training with learning rate: {lr}")
    training2(X_train_images, X_train_tabular, y_train, X_test_images, X_test_tabular, y_test, epochs=55, batch_size=32, learning_rate=lr,dropout=0.2, optimizer_name="adam", l2_factor=0.0)
```

Try Different Batch Sizes

```python
# List of batch sizes to experiment with
batch_sizes = [16, 32, 64]

# Loop through each batch size and train the model
for batch_size in batch_sizes:
    print(f"Training with batch size: {batch_size}")
    training2(X_train_images, X_train_tabular, y_train, X_test_images, X_test_tabular, y_test, epochs=55, batch_size=batch_size, learning_rate=0.0001,dropout=0.2, optimizer_name="adam", l2_factor=0.0)
```

Try Different dropouts

```python
dropout_list = [0.2 , 0.4 , 0.5]

for dropout in dropout_list:
    print(f"Training with dropout: {dropout}")
    training2(X_train_images, X_train_tabular, y_train, X_test_images, X_test_tabular, y_test, epochs=55, batch_size=32, learning_rate=0.0001,dropout=dropout, optimizer_name="adam", l2_factor=0.0)
```

Try Different Optimizers

```python
# List of optimizers to experiment with
optimizers = ['adam', 'sgd', 'rmsprop']

# Loop through each optimizer and train the model
for optimizer in optimizers:
    print(f"Training with optimizer: {optimizer}")
    training2(X_train_images, X_train_tabular, y_train, X_test_images, X_test_tabular, y_test, epochs=55, batch_size=32, learning_rate=0.0001, dropout=0.2, optimizer_name=optimizer, l2_factor=0.0)
```

Try different L2 Regularization Factors

7m

```python
# List of L2 regularization factors to experiment with
l2_factors = [0.001, 0.01, 0.0001]

# Loop through each L2 factor and train the model
for l2_factor in l2_factors:
    print(f"Training with L2 regularization factor: {l2_factor}")

    # Call the training function
    training2(
        X_train_images, X_train_tabular, y_train,
        X_test_images, X_test_tabular, y_test,
        epochs=55, batch_size=32, learning_rate=0.0001,
        dropout=0.2, optimizer_name='adam', l2_factor=l2_factor
    )
```

Try Different number of layers

```
training(
    X_train_images, X_train_tabular, y_train,
    X_test_images, X_test_tabular, y_test,
    epochs=55, batch_size=32, learning_rate=0.0001,
    dropout=0.2, optimizer_name='adam', l2_factor=0.0
)

training2(
    X_train_images, X_train_tabular, y_train,
    X_test_images, X_test_tabular, y_test,
    epochs=55, batch_size=32, learning_rate=0.0001,
    dropout=0.2, optimizer_name='adam', l2_factor=0.0
)

training3(
    X_train_images, X_train_tabular, y_train,
    X_test_images, X_test_tabular, y_test,
    epochs=55, batch_size=32, learning_rate=0.0001,
    dropout=0.2, optimizer_name='adam', l2_factor=0.0
)
```

Download Best Model & Tensorboard logs

[13]

```
from google.colab import files

# Download the best model
files.download('best_model.keras')
```

```
import shutil

# Specify the correct path for the log directory
log_dir = 'logs/fit/'  # Adjust this path if needed

# Zip the log directory
shutil.make_archive('/content/tensorboard_logs', 'zip', log_dir)

# Provide a download link
from google.colab import files
files.download('/content/tensorboard_logs.zip')
```

Outputs Screenshots

Training with learning rate: 0.0001

```
Epoch 55/55
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step - accuracy: 0.9167 - loss: 0.3789
Epoch 55: val_accuracy did not improve from 0.91195
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 102ms/step - accuracy: 0.9163 - loss: 0.3807 - val_accuracy: 0.8931 - val_loss: 0.4296
Train Accuracy: 0.9785, Test Accuracy: 0.9444
```

Training with learning rate: 0.001

```
Epoch 55/55
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 84ms/step - accuracy: 0.9924 - loss: 0.0392
Epoch 55: val_accuracy did not improve from 0.95597
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 105ms/step - accuracy: 0.9922 - loss: 0.0396 - val_accuracy: 0.8805 - val_loss: 0.4868
Train Accuracy: 0.9710, Test Accuracy: 0.9444
```

Training with learning rate: 0.01

```
Epoch 55/55
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step - accuracy: 0.7072 - loss: 1.5260
Epoch 55: val_accuracy did not improve from 0.71698
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 104ms/step - accuracy: 0.7073 - loss: 1.5243 - val_accuracy: 0.1635 - val_loss: 12.6288
Train Accuracy: 0.2626, Test Accuracy: 0.2374
```

Training with batch size: 16

```
Epoch 55/55
39/40 ━━━━━━━━━━━━━━━━━━━━ 0s 42ms/step - accuracy: 0.9400 - loss: 0.2220
Epoch 55: val_accuracy did not improve from 0.93711
40/40 ━━━━━━━━━━━━━━━━━━━━ 2s 53ms/step - accuracy: 0.9406 - loss: 0.2221 - val_accuracy: 0.8491 - val_loss: 0.4480
Train Accuracy: 0.9621, Test Accuracy: 0.8889
```

Training with batch size: 32

```
Epoch 55/55
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 80ms/step - accuracy: 0.9350 - loss: 0.3244
Epoch 55: val_accuracy did not improve from 0.94969
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 103ms/step - accuracy: 0.9350 - loss: 0.3240 - val_accuracy: 0.9245 - val_loss: 0.3781
Train Accuracy: 0.9848, Test Accuracy: 0.9293
```

Training with batch size: 64

```
Epoch 55/55
10/10 ━━━━━━━━━━━━━━━━━━━━ 0s 159ms/step - accuracy: 0.8914 - loss: 0.6253
Epoch 55: val_accuracy did not improve from 0.66667
10/10 ━━━━━━━━━━━━━━━━━━━━ 2s 208ms/step - accuracy: 0.8908 - loss: 0.6259 - val_accuracy: 0.6038 - val_loss: 1.5003
Train Accuracy: 0.7487, Test Accuracy: 0.6313
```

Training with epochs: 30

```
Epoch 30/30
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 83ms/step - accuracy: 0.8091 - loss: 0.9826
Epoch 30: val_accuracy did not improve from 0.71698
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 106ms/step - accuracy: 0.8093 - loss: 0.9809 - val_accuracy: 0.6352 - val_loss: 1.5383
Train Accuracy: 0.8131, Test Accuracy: 0.6919
```

Training with epochs: 50

```
Epoch 50/50
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 81ms/step - accuracy: 0.9156 - loss: 0.4260
Epoch 50: val_accuracy did not improve from 0.90566
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 102ms/step - accuracy: 0.9156 - loss: 0.4267 - val_accuracy: 0.8868 - val_loss: 0.5816
Train Accuracy: 0.9735, Test Accuracy: 0.8889
```

Training with epochs: 70

```
Epoch 70: val_accuracy did not improve from 0.92453
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 104ms/step - accuracy: 0.9604 - loss: 0.2179 - val_accuracy: 0.9245 - val_loss: 0.3478
Train Accuracy: 0.9848, Test Accuracy: 0.9242
```

Training with dropout: 0.2

```
Epoch 55/55
20/20 ━━━━━━━━━━━━━━━━━━━━ 0s 84ms/step - accuracy: 0.9382 - loss: 0.3039
Epoch 55: val_accuracy did not improve from 0.94969
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 119ms/step - accuracy: 0.9376 - loss: 0.3064 - val_accuracy: 0.9371 - val_loss: 0.3708
Train Accuracy: 0.9861, Test Accuracy: 0.9293
```

Training with dropout: 0.4

```
Epoch 55: val_accuracy improved from 0.69182 to 0.71698, saving model to best_model.keras
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 111ms/step - accuracy: 0.6225 - loss: 1.4384 - val_accuracy: 0.7170 - val_loss: 1.2652
Train Accuracy: 0.8876, Test Accuracy: 0.8232
```

Training with dropout: 0.5

```
Epoch 55: val_accuracy improved from 0.40252 to 0.49686, saving model to best_model.keras
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 110ms/step - accuracy: 0.3569 - loss: 2.4910 - val_accuracy: 0.4969 - val_loss: 2.3445
Train Accuracy: 0.7020, Test Accuracy: 0.5404
```

Training with optimizer: adam

```
Epoch 55: val_accuracy did not improve from 0.89308
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 106ms/step - accuracy: 0.9358 - loss: 0.3108 - val_accuracy: 0.8868 - val_loss: 0.4386
Train Accuracy: 0.9760, Test Accuracy: 0.9040
```

Training with optimizer: sgd

```
Epoch 55: val_accuracy did not improve from 0.02516
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 108ms/step - accuracy: 0.0107 - loss: 4.6302 - val_accuracy: 0.0000e+00 - val_loss: 4.6506
Train Accuracy: 0.0063, Test Accuracy: 0.0152
```

Training with optimizer: rmsprop

```
Epoch 55: val_accuracy improved from 0.91195 to 0.93711, saving model to best_model.keras
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 109ms/step - accuracy: 0.8854 - loss: 0.5256 - val_accuracy: 0.9371 - val_loss: 0.5261
Train Accuracy: 0.9836, Test Accuracy: 0.8889
```

Training with L2 regularization factor: 0.001

```
Epoch 55: val_accuracy did not improve from 0.91824
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 114ms/step - accuracy: 0.9239 - loss: 0.3499 - val_accuracy: 0.9119 - val_loss: 0.4036
Train Accuracy: 0.9811, Test Accuracy: 0.8939
```

Training with L2 regularization factor: 0.01

```
Epoch 55: val_accuracy improved from 0.88679 to 0.89937, saving model to best_model.keras
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 120ms/step - accuracy: 0.9302 - loss: 0.3296 - val_accuracy: 0.8994 - val_loss: 0.4680
Train Accuracy: 0.9785, Test Accuracy: 0.9192
```

Training with L2 regularization factor: 0.0001

```
Epoch 55: val_accuracy did not improve from 0.92453
20/20 ━━━━━━━━━━━━━━━━━━━━ 2s 106ms/step - accuracy: 0.9196 - loss: 0.3951 - val_accuracy: 0.9119 - val_loss: 0.4375
Train Accuracy: 0.9823, Test Accuracy: 0.9040
```

Training with 2 Conv Layers:

```
Epoch 55: val_accuracy did not improve from 0.89308
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 80ms/step - accuracy: 0.9055 - loss: 0.4535 - val_accuracy: 0.8868 - val_loss: 0.5079
Train Accuracy: 0.9735, Test Accuracy: 0.9141
```

Training with 3 Conv Layers:

```
Epoch 55: val_accuracy did not improve from 0.91824
20/20 ━━━━━━━━━━━━━━━━━━━━ 3s 115ms/step - accuracy: 0.9042 - loss: 0.3642 - val_accuracy: 0.8994 - val_loss: 0.4674
Train Accuracy: 0.9760, Test Accuracy: 0.8838
```

Training with 4 Conv Layers:

```
Epoch 55: val_accuracy did not improve from 0.90566
20/20 ━━━━━━━━━━━━━━━━━━━━ 5s 147ms/step - accuracy: 1.0000 - loss: 0.0377 - val_accuracy: 0.8616 - val_loss: 0.7152
Train Accuracy: 0.9722, Test Accuracy: 0.8737
```

Model Description

The proposed model for leaf classification utilizes a hybrid architecture that processes both image and tabular data simultaneously. The model consists of two distinct pathways: one for the image data and one for the tabular data. These pathways are then merged into a unified feature vector, which is used for classification. Below is a detailed explanation of the model components:

1. **Image Data Path (Convolutional Neural Network - CNN)**:
    - **Input Layer**: The model accepts images of size 192x192x3 (RGB).
    - **Convolutional Layers**: Three convolutional layers with increasing depth (64, 128, and 256 filters) and ReLU activation functions extract hierarchical features from the image data. Each convolutional layer is followed by batch normalization and max-pooling operations to enhance feature extraction and reduce spatial dimensions.
    - **Global Average Pooling**: This layer reduces the dimensionality of the feature maps from the convolutional layers, converting them into a single vector representation.
    - **Dropout Layer**: Applied after pooling to reduce overfitting by randomly setting a fraction of input units to zero during training.
2. **Tabular Data Path (Fully Connected - Dense Layers)**:
    - **Input Layer**: The tabular data is input as a 1D vector, where each feature corresponds to a dimension in the dataset.
    - **Dense Layers**: Two fully connected layers (128 and 64 units) with ReLU activations are used to process the tabular data. Dropout is applied after each dense layer to mitigate overfitting.
3. **Feature Concatenation**:
    - The features extracted from both the image and tabular data paths are concatenated to form a combined feature vector, which captures information from both modalities.
4. **Fully Connected Layers (Dense Layers)**:
    - The combined features are passed through two fully connected layers with 512 and 256 units, both utilizing ReLU activation functions. Dropout is applied after each layer to improve generalization.
5. **Output Layer**:
    - The final layer is a softmax-activated dense layer, with the number of units equal to the number of unique classes in the dataset. This layer outputs a probability distribution over the classes, where the highest probability corresponds to the predicted class.
6. **Optimization and Regularization**:

- The model uses different optimizers, including Adam, SGD, and RMSProp, with L2 regularization (weight decay) to prevent overfitting and improve generalization.
- The learning rate is adjustable to control the rate of weight updates during training.

7. **Training**:
    - The model is trained using a cross-entropy loss function (sparse_categorical_crossentropy), suitable for multi-class classification problems with integer labels. The training process is monitored using TensorBoard, and the best model is saved based on validation accuracy.
8. **Callbacks**:
    - **ModelCheckpoint**: Saves the best model based on validation accuracy during training.
    - **TensorBoard**: Logs training and validation metrics to visualize the training process and evaluate the model's performance.

This hybrid model leverages the strengths of both CNNs for image processing and dense layers for tabular data, making it a powerful approach for leaf classification tasks that involve both image and tabular inputs.

Result Analysis

From these training results, here are the key observations:

Observations:

1. **Learning Rate**:
   - A **low learning rate** (0.0001) performed well with stable training but slightly lower generalization (validation accuracy).
   - A **moderate learning rate** (0.001) gave the highest test accuracy and balanced performance, showing that it is likely the optimal choice.
   - A **high learning rate** (0.01) caused instability in training with poor accuracy and extremely high validation loss.
2. **Batch Size**:
   - A **batch size of 32** achieved the best validation accuracy (0.9497) with good generalization.
   - Larger batch sizes (64) resulted in reduced performance, likely due to poor gradient updates.
   - Smaller batch sizes (16) were slightly less efficient, potentially due to noisy updates.
3. **Number of Epochs**:
   - Extending training to 70 epochs improved results slightly compared to 50 epochs.
4. **Dropout**:
   - A dropout rate of **0.2** gave strong performance with validation accuracy of 0.9371.
   - Higher dropout rates (0.4 and 0.5) severely impacted accuracy and model generalization.
5. **Optimizers**:
   - The **Adam optimizer** achieved good overall performance, with the highest test accuracy (0.9040).
   - **RMSprop** performed comparably, reaching a validation accuracy of 0.9371 but slightly lower test accuracy.
   - **SGD** failed to converge properly in this setup, possibly requiring tuning of the learning rate or momentum.
6. **L2 Regularization**:
   - Adding L2 regularization with a factor of 0.001 led to stable training but did not improve validation accuracy significantly.

Summary of Best Results:

- **Learning Rate:**
    - **0.0001**: Train Accuracy: 0.9785, Test Accuracy: 0.9444
    - **0.001**: Train Accuracy: 0.9710, Test Accuracy: 0.9444
    - **0.01**: Poor performance (Test Accuracy: 0.2374)
- **Batch Size:**
    - **32**: Train Accuracy: 0.9848, Test Accuracy: 0.9293 (Val Accuracy: 0.9497)
    - **16**: Train Accuracy: 0.9621, Test Accuracy: 0.8889
    - **64**: Poor performance (Test Accuracy: 0.6313)
- **Epochs:**
    - **50**: Train Accuracy: 0.9735, Test Accuracy: 0.8889
    - **70**: Train Accuracy: 0.9848, Test Accuracy: 0.9242
- **Dropout:**
    - **0.2**: Train Accuracy: 0.9861, Test Accuracy: 0.9293
    - **0.4**: Moderate performance (Test Accuracy: 0.8232)
    - **0.5**: Poor performance (Test Accuracy: 0.5404)
- **Optimizers:**
    - **RMSprop**: Train Accuracy: 0.9836, Test Accuracy: 0.8889 (Val Accuracy: 0.9371)
    - **Adam**: Train Accuracy: 0.9760, Test Accuracy: 0.9040
    - **SGD**: Poor performance (Test Accuracy: 0.0152)
- **Regularization (L2 factor = 0.01)**: Train Accuracy: 0.9785, Test Accuracy: 0.9192

Best Model:

- **Batch Size 32 with Dropout 0.2**:
    - Train Accuracy: 0.9861
    - Test Accuracy: 0.9293
    - Validation Accuracy: **0.9497**