Tallapelli Saicharan 1 minute ago 13 min read

Image Classification: Mango Disease Image Dataset

Objective:

The aim of this project is to create an image classifier for mango disease image dataset that contains seven classes of mango disease and one class of healthy namely:

1. Anthracnose
2. Bacterial Canker
3. Cutting Weevil
4. Die Back
5. Gall Midge
6. Healthy
7. Powdery Mildew
8. Sooty Mould

By the end of the blog, we will be able to create a model with satisfactorily accuracy. We'll also see the comparison between accuracies of different models.

We'll use both traditional machine learning and modern deep learning techniques. So, without any further delay let's begin.

Introduction:

Computer Vision is an emerging field of artificial intelligence in which developments are happening at rapid pace. The computer vision deals with images, videos and other forms of media which includes pixels or some sort of image properties. There are many tasks of computer vision one of them is Image Classification in which algorithms are trained to label input images to their particular class. This can be done using both machine learning and deep learning techniques.

The programming language that we are going to use is python. And there are some libraries which we've to include in order to accomplish our objective:

1. Pandas (to store results in csv format).
2. Matplotlib and Seaborn (to visualize outputs).
3. Scikit-learn (for implementing machine learning algorithms).
4. OpenCV (to work with images).
5. NumPy (to handle multidimensional arrays and metrices operations).
6. TensorFlow (A deep learning library to use deep learning models).

Now let's define steps that we're going to follow throughout the whole project. The machine learning algorithms that we're going to use are K-Nearest Neighbors, Support Vector Machine, Random Forest, Decision Tree, Naïve Bayes, Logistic Regression. And for deep learning we're going to use Vanilla CNN and Efficient Net V2.

Now let's dive into steps:

1. Importing necessary modules.
2. Loading the dataset.
3. Label encoding and splitting the data.
4. Data Preprocessing.
5. Model Training.

1) Importing necessary modules.

```
# Importing Necessary Modules
import os
import re
import cv2
import time
import random
import warnings
import pandas as pd
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from pathlib import Path
from PIL import Image
from tqdm import tqdm

from sklearn.naive_bayes import GaussianNB
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix, classification_report,
f1_score

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Dense, Activation,Dropout,Conv2D,
MaxPooling2D,BatchNormalization
from tensorflow.keras.optimizers import Adam, Adamax
from tensorflow.keras.metrics import categorical_crossentropy
from tensorflow.keras import regularizers
from tensorflow.keras.models import Model
from tensorflow.keras import backend as K
```

```
%matplotlib inline
warnings.filterwarnings('ignore')
```

2) Loading the Data:

The dataset for mango disease is available on Kaggle. Basically, it contains 8 classes . We'll use all images in the dataset and then resize it to 128 and 256 according to the need.
Let's define some important variables:

```
#variables to easily perform iterations
image_size_to_train = (128, 128)

#we will store all the results in each iteration in a
report = {}
training_times = {}
best_params = {}
training_hyper_times = {}

vis_path='visualizations/{}'.format(image_size_to_train[0])
Path(vis_path).mkdir(parents=True, exist_ok=True)
```

Image_size_to_train : this variable contains the size of image for training.
Report: this dictionary will contain accuracy according to each model.
Training_times: this dictionary will contain the time required to train each model.
Best_params: this dictionary will contain the best parameters we'll get after hyperparameter tuning.
Training_hyper_times: this dictionary will contain the time required for hyperparameter tuning of each model.

Now, after defining variables we'll go through the dataset to find it's length and store path of each image in a variable image_path and store all labels in labels.

```
dataset_path="archive"
image_paths = []
labels = []
for sub_fol in os.listdir(dataset_path):
  for img in os.listdir(f"{dataset_path}/{sub_fol}"):
    image_paths.append(f"{dataset_path}/{sub_fol}/{img}")
    labels.append(sub_fol)

unique_labels = os.listdir(dataset_path)
print("Number of images found:", len(image_paths))
print("Number of unique labels:",len(unique_labels))
print("Unique labels:", unique_labels)
```

As we've found all labels, but machine learning and deep learning algorithms won't work on categorical features we have to convert it to numbers so we'll use label encoding.

```
#convert labels to numbers using LabelEncoder
le = LabelEncoder()
le.fit(labels)
labels = le.transform(labels)

#get labels names
labels_to_names = dict(zip(le.transform(le.classes_), le.classes_))
print("Labels to names:",labels_to_names)
```

3) Splitting data

It's important to test model accuracy after training and to do so we need to have a test data also. So, for this reason we'll be splitting our data into 70:30 ratio in which 70% will be training data and 30% will be testing data.

```
#now split the data into train and test sets in the ratio of 80:20
X_train, X_test, y_train, y_test = train_test_split(image_paths,
labels, test_size=0.3, train_size = 0.7, random_state=42)
```

4) Data Preprocessing

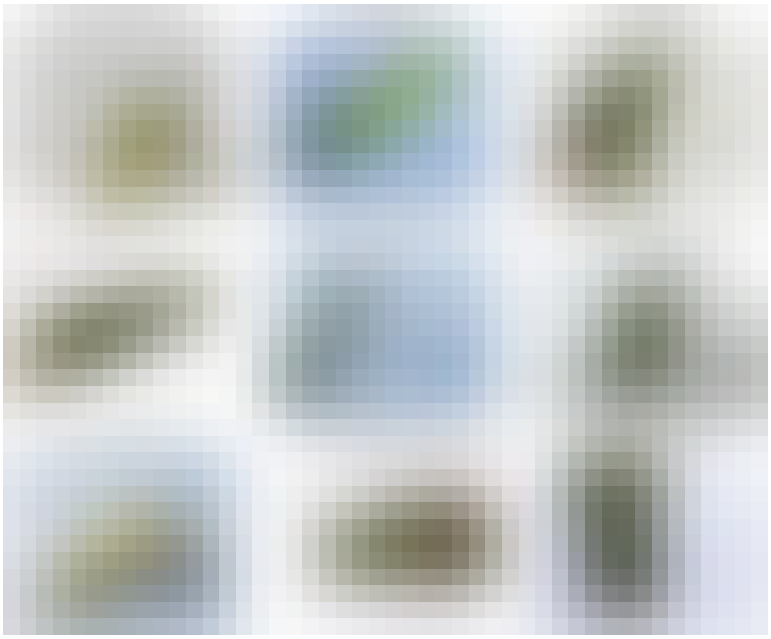
Now, we're having image paths, we'll convert each image to a numpy array and reshape and resize it according to our need so that it'll be useful for our machine learning algorithms.

```
#now we will create a function to load the images and resize them to
128X128
def process_images(paths, labels, resize = (128, 128), flatten=True,
verbose=True):
    X = []
    y = []
    for i, path in enumerate(paths):
        if verbose:
            print("Processing image number:", i)
        try:
            img = cv2.imread(path)
            img = cv2.cvtColor(img,cv2.COLOR_BGR2RGB)
            img = cv2.resize(img, resize)
```

```
                X.append(img)
                y.append(labels[i])

            except:
                print("Error in image: ", path)

        X = np.array(X)

        if flatten:
            X = X.reshape(X.shape[0],-1)

        return X,y

#now we will load the train and test images
print("Loading Training Images into memory")
X_train, y_train = process_images(X_train, y_train, resize =
image_size_to_train, flatten=True, verbose=True)

print("Loading Test Images into memory")
X_test,y_test=process_images(X_test, y_test, resize =
image_size_to_train,flatten=True, verbose=True)
```

Before moving further into model training we'll have to make sure that we've successfully loaded data into our memory. Therefore, we'll randomly visualize 9 images with their labels.

```
#visualize the resized images again
fig = plt.figure(figsize = (10,10))

for i in range(9):
    ax = fig.add_subplot(3,3,i+1)
    random_index = random.randint(0,len(X_train))
    img =
X_train[random_index].reshape(image_size_to_train[0],image_size_to_trai
n[1],3)
    ax.imshow(img)
    ax.set_title(labels_to_names[y_train[random_index]])
    ax.axis('off')
```



We can try running code multiple times to get different images.

5)Model Training

Model training is an important process in machine learning cause the accuracy of our project depends on it. We've used some references to train our models. We have selected 6 machine learning algorithms. And to increase their accuracy we've used a hyper parameter tuning technique called GridSearch CV, we didn't saw a greater improvement in results though but there are slight improvements. We can improve results more by increasing parameters while tuning our models.

Below is the code where we've created two dictionaries the first one contains the instances of the models and the second one contains parameters grid. We can improve accuracy by taking high values in parameters grid but hyperparameter is a time-consuming process.

```
models = {
    "Logistic Regression": LogisticRegression(),
    "Naive Bayes": GaussianNB(),
    "KNN": KNeighborsClassifier(),
    "Decision Tree": DecisionTreeClassifier(),
    "Random Forest": RandomForestClassifier(),
    "SVM" : SVC()
    }
param_grid = {
    "KNN": dict(n_neighbors = [5, 7, 11, 13], weights = ['uniform',
'distance']),
    "SVM": dict(gamma = [1, 10, 100], kernel = ["rbf", "poly"], C =
[10, 50, 100]),
    "Random Forest": dict(n_estimators = np.arange(100, 300, 50),
criterion = ["gini", "entropy", "log_loss"]),
    "Logistic Regression": None,
```

```
    "Decision Tree": dict(max_depth = np.arange(100, 350, 50),
criterion = ["gini", "entropy", "log_loss"]),
    "Naive Bayes": None
    }
```

At the beginning , we had created some empty dictionaries and in the below code we're going to use it to store results , training time of models and training time of models while they are tuning their hyperparameters. Below is the code where we tuned each and every model and calculated its accuracy.

```
def hyper_parameter_tuning():
  for idx in range(len(list(models))):
    model = list(models.values())[idx]
    params = param_grid[list(models.keys())[idx]]
    print(f'Working on {list(models.keys())[idx]} algorithm...')
    if params is not None:
      start_time = time.time()
      gs = GridSearchCV(model, params, cv = 3, n_jobs = -1)
      gs.fit(X_train, y_train)
      training_hyper_times[list(models.keys())[idx]] = time.time() -
start_time

      best_params[list(models.keys())[idx]] = gs.best_params_

      start_time = time.time()
      model.set_params(**gs.best_params_)
      model.fit(X_train, y_train)
      training_times[list(models.keys())[idx]] = time.time() -
start_time

    else:
      start_time = time.time()
      model.fit(X_train, y_train)
      training_hyper_times[list(models.keys())[idx]] = None
      training_times[list(models.keys())[idx]] = time.time() -
start_time

    y_train_pred = model.predict(X_train)
    y_test_pred = model.predict(X_test)

    train_model_score = accuracy_score(y_train, y_train_pred) * 100
    test_model_score = accuracy_score(y_test, y_test_pred) * 100
```

```
        print(f"Accuracy for {list(models.keys())[idx]} is:
    {test_model_score}")

        report[list(models.keys())[idx]] = test_model_score

    return report
```

This code performs hyperparameter tuning for a set of machine learning models using grid search cross-validation. The goal of hyperparameter tuning is to find the best set of hyperparameters (parameters that are not learned from the data but are set prior to training) for a given model, which can improve its performance on a particular task.
The code starts by defining a dictionary models that maps the name of a model (as a string) to an instance of the corresponding scikit-learn estimator class. The models being used in this case are logistic regression, Naive Bayes, k-nearest neighbors (KNN), decision tree, random forest, and support vector machine (SVM).
Another dictionary param_grid is also defined, which maps the name of a model to a dictionary of hyperparameters and their values to be searched over using grid search. For example, for the KNN model, the hyperparameters being tuned are the number of neighbors and the weighting function. For SVM, the hyperparameters being tuned are the kernel type, regularization parameter C, and gamma.
The hyper_parameter_tuning() function then iterates over the models and their respective hyperparameter grids, and performs grid search cross-validation to find the best hyperparameters. If a model has no hyperparameters to tune, it simply fits the model to the training data with default parameters.
For each model, the function records the best hyperparameters, the time it takes to tune the hyperparameters, the time it takes to train the model with the best hyperparameters, and the accuracy score on the test set. The results are stored in dictionaries and returned as a report.

Now we'll look into details of some of the algorithms.

1. K-Nearest Neighbors

It's a machine learning algorithm mostly used for classification. It is a distance-based algorithm and uses a parameter n_neighbors i.e., number of neighbors for any point.

2. Support Vector Machine

It's a supervised machine learning algorithm which uses hyperplanes and decision boundaries for separation of classes and mainly it is used for classification.

3. Random Forest Classifier

It's an ensemble learning technique used mainly for classification. The advantage of this algorithm is that it uses multiple decision tree in a parallel manner simultaneously.

4. Logistic Regression

The main use of logistic regression if for binary classification but sometimes we use it for multiclass classification also.

5. Decision Tree

It's a supervised machine learning algorithm which uses concepts like gini_index and entropy. It is also mainly used for classification.

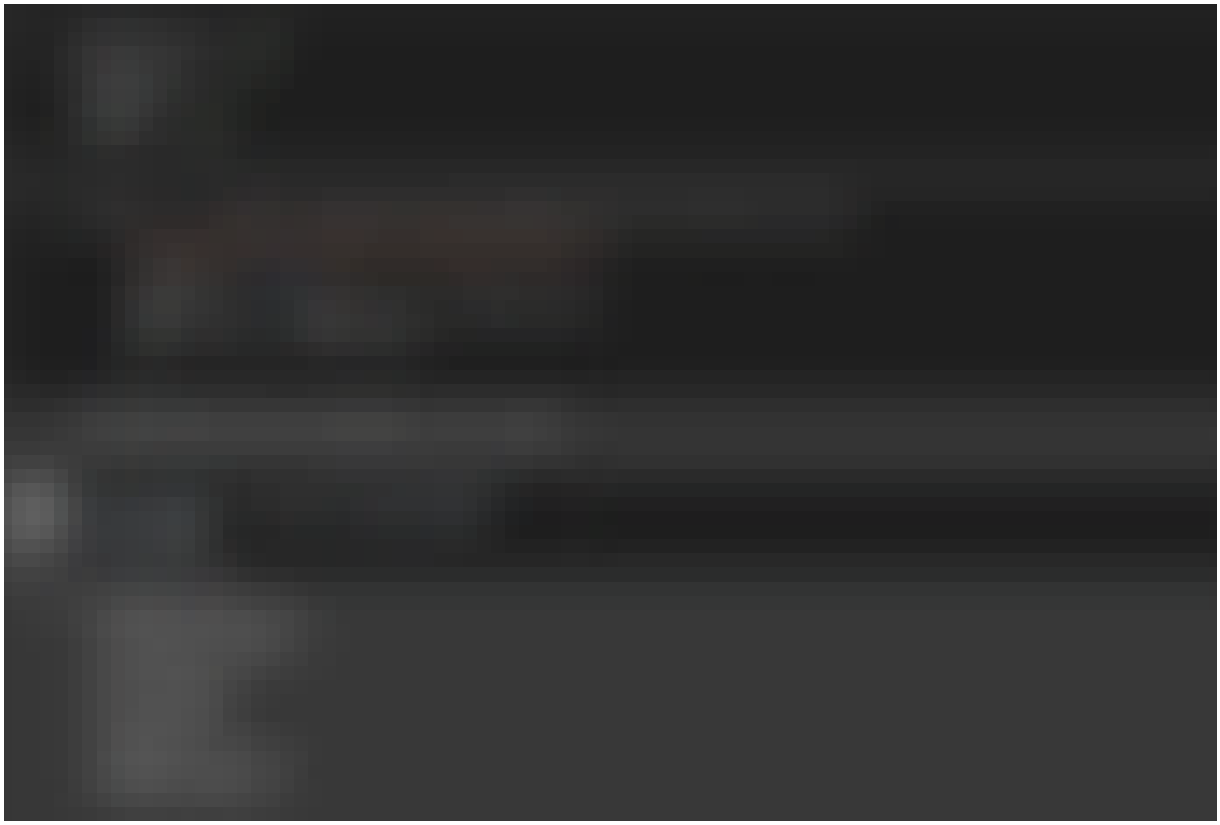
6. Naïve Bayes

It's also a supervised learning algorithm but it's based on bayes theorem. Bayes theorem is based on conditional probability. In naïve bayes algorithm strictly assumes that the features in dataset are strictly independent of each other.

7. Convolutional Neural Network (CNN)

It is a type deep learning neural network which is designed to process data such as images. This type of neural networks are very good at recognizing patterns in input images, such as lines, gradients, circles, and even eyes and faces. This is the property that makes convolutional neural networks so powerful for computer vision.
For this project we're going to use two of its architecture namely Vanilla CNN and Efficient Net V2.

Data preprocessing will be different for developing CNN models from that of machine learning models as shown below:



In the above three cells we are doing three things:

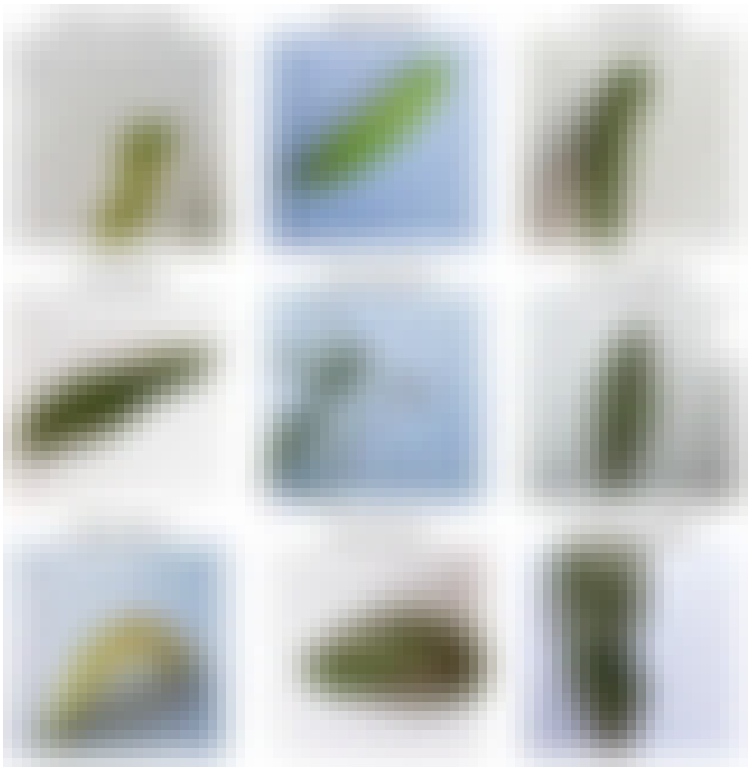
1. In the first cell we're defining parameters i.e., image size, batch size, number of channels our image is having, and epochs.
2. In the seconds cell we're creating dataset and loading it into the memory using tensorflow.
3. In the third cell we're defining class_names for storing name of mango disease.

Below we're trying to visualize our data by taking a batch from generated dataset above.

```
plt.figure(figsize = (10,10))
for image_batch, label_batch in dataset.take(1):
  for i in range(12):
```

```
    plt.subplot(3,4,i+1)
    plt.imshow(image_batch[i].numpy().astype('uint8'))
    plt.axis("off")
    plt.title(class_names[label_batch[i]])
```

output:



To train our model we need to have train, test and validation set of data. So for that purpose we've defined a function to create train, test and validation set of data

```
def get_dataset_partitions_tf(ds, train_split = 0.8, val_split = 0.1,
test_split = 0.1, shuffle = True, shuffle_size = 10000):
  if shuffle:
    ds = ds.shuffle(shuffle_size, seed = 12)

  ds_size = len(ds) # taking number of batches into account
  train_size = int(ds_size * train_split) # evaluating number of
batches to be in training dataset
  val_size = int(ds_size * val_split) # evaluating number of batches to
be in validation dataset

  train_ds = ds.take(train_size) # creating training dataset partition
  #ds.skip(train_size) # skipping batches which are in training dataset

  val_ds = ds.skip(train_size).take(val_size) # creating validation
```

```
    dataset partition
      test_ds = ds.skip(train_size).skip(val_size) # creating testing
    dataset partition

      return train_ds, val_ds, test_ds

    train_ds, val_ds, test_ds = get_dataset_partitions_tf(dataset)
```

Next, we created resize_rescale and data augmentation layer to be used while training and it can provide an edge for accuracy of our model.

```
    resize_rescale = tf.keras.Sequential([
        layers.experimental.preprocessing.Resizing(IMAGE_SIZE, IMAGE_SIZE),
        layers.experimental.preprocessing.Rescaling(1.0/255)
        ])

    data_augmentation = tf.keras.Sequential([

    layers.experimental.preprocessing.RandomFlip("horizontal_and_vertical")
    ,
        layers.experimental.preprocessing.RandomRotation(0.2)
        ])
    input_shape = (BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, CHANNEL)
    number_of_classes = len(class_names)
```

This code defines two preprocessing pipelines using the tf.keras.Sequential() function to create sequential models in TensorFlow.
The first pipeline, resize_rescale, consists of two preprocessing layers: Resizing() and Rescaling(). Resizing() is used to resize the input image to a specific size (specified by IMAGE_SIZE in the code) and Rescaling() is used to scale pixel values between 0 and 1. Resizing() is needed because the input images may not be of the same size and this can cause issues during training. By resizing the images, we can ensure that all images have the same size, which makes the model training more efficient. Rescaling() is used to normalize the pixel values between 0 and 1, which helps the model to converge faster during training.
The second pipeline, data_augmentation, is used for data augmentation. Data augmentation is a technique used to artificially increase the size of the training dataset by creating new images from the existing ones. This is done by applying various transformations to the images, such as rotating, flipping, or shifting. Data augmentation helps to prevent overfitting and improve the generalization ability of the model. In this code, RandomFlip() and RandomRotation() are used to randomly flip the images horizontally and vertically, and randomly rotate the images by a maximum of 0.2 radians.
Both pipelines are created as sequential models, which means that the input is passed through each layer in sequence. These pipelines can then be used as input layers in a neural network model, allowing for efficient preprocessing and data augmentation during training.

Now we'll code for callbacks:

we'll create callback function to manipulate parameters if there's no change in accuracies. It'll do the following things :

a. It'll stop training if there no longer changes in accuracies or losses.
b. It'll change learning rate if there'll not be much change in accuracies and losses.
c. Once stopped it'll save our model to a specified folder.

```
metric="val_categorical_accuracy"

def create_callbacks(metric = metric, filename = "base_model_256.h5",
patience=5):
     Path("models").mkdir(parents=True, exist_ok =True)
     cpk_path ="models/"+filename
     checkpoint = tf.keras.callbacks.ModelCheckpoint(
        filepath = cpk_path,
        monitor= metric,
        mode='auto',
        save_best_only=True,
        verbose=1,
    )
     earlystop = tf.keras.callbacks.EarlyStopping(
        monitor = metric,
        mode='auto',
        patience=patience,
        verbose=1
    )
     reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(
        monitor = metric,
        mode='auto',
        patience=patience//2,
        verbose=1,
        factor=0.5
    )
     callbacks = [checkpoint, earlystop, reduce_lr]
     return callbacks
```

Code for Vanilla CNN:

```
tf.keras.backend.clear_session()
with tf.device('/device:GPU:1'):
  vanilla_cnn = tf.keras.Sequential([
          # resize_rescale,
          data_augmentation,
          tf.keras.layers.Conv2D(32, 3, activation='relu', input_shape
= input_shape),
          tf.keras.layers.MaxPool2D(),
          tf.keras.layers.Conv2D(64, 3, activation='relu'),
          tf.keras.layers.MaxPool2D(),
          tf.keras.layers.Conv2D(128, 3, activation='relu'),
          tf.keras.layers.MaxPool2D(),
          tf.keras.layers.Conv2D(256, 3, activation='relu'),
          tf.keras.layers.MaxPool2D(),
          tf.keras.layers.Flatten(),
          tf.keras.layers.Dense(512, activation='relu'),
          tf.keras.layers.Dense(number_of_classes,
activation='sigmoid'),
      ])
  vanilla_cnn.build(input_shape)
  vanilla_cnn.compile(optimizer =
tf.keras.optimizers.Adam(learning_rate = 0.0001),
                                  loss =
tf.keras.losses.SparseCategoricalCrossentropy(),
                                  metrics = 'categorical_accuracy')
  callbacks = create_callbacks(metric = metric, filename =
"vanilla_cnn_256.h5", patience = 10)
  start_time = time.time()
  base_history = vanilla_cnn.fit(train_ds,
                                  epochs = EPOCHS,
                                  validation_data = val_ds,
                                  callbacks = callbacks,
                                  batch_size = BATCH_SIZE,
                                  verbose = 1
                                  )

  training_times['Vanilla_CNN'] = time.time() - start_time

report["Vanilla_CNN"] =
```

```
max(base_history.history['categorical_accuracy'])
vanilla_cnn.summary()
```

This code defines a convolutional neural network (CNN) model using the TensorFlow library to perform image classification. The model consists of several convolutional and max pooling layers, followed by a dense layer and a final output layer with sigmoid activation. The code also applies data augmentation to increase the variety of images seen by the model during training. The model is trained on a dataset using the Adam optimizer, sparse categorical cross-entropy loss function, and a specified number of epochs. The accuracy of the trained model is then evaluated and stored for later analysis.
Code for Efficient Net:

```
tf.keras.backend.clear_session()
with tf.device('/device:GPU:0'):
  efficient_model = tf.keras.Sequential([
      # resize_rescale,
      # data_augmentation,

tf.keras.applications.efficientnet_v2.EfficientNetV2B0(include_top =
False, weights='imagenet', input_shape = input_shape[1:]),
      tf.keras.layers.GlobalAveragePooling2D(),
      tf.keras.layers.Dense(512, activation =
tf.keras.layers.LeakyReLU(alpha=0.2)),
      tf.keras.layers.BatchNormalization(),
      tf.keras.layers.Dense(128, activation =
tf.keras.layers.LeakyReLU(alpha=0.2)),
      tf.keras.layers.Dense(number_of_classes, activation="sigmoid",
dtype='float32')

  ])
  efficient_model.build(input_shape)
  efficient_model.compile(optimizer =
tf.keras.optimizers.Adam(learning_rate = 0.001),
                          loss =
tf.keras.losses.SparseCategoricalCrossentropy(),
                          metrics = 'categorical_accuracy'
                          )
  callbacks =
create_callbacks(filename="efficient_net_v2_model_256.h5")
  start_time = time.time()
  effn_history = efficient_model.fit(train_ds,
```

```
                                    epochs = EPOCHS,
                                    validation_data = val_ds,
                                    callbacks = callbacks,
                                    batch_size = BATCH_SIZE
                                    )
    training_times['EfficientNetV2'] = time.time() - start_time
report["Efficient_Net_V2"] =
max(effn_history.history['categorical_accuracy'])
```

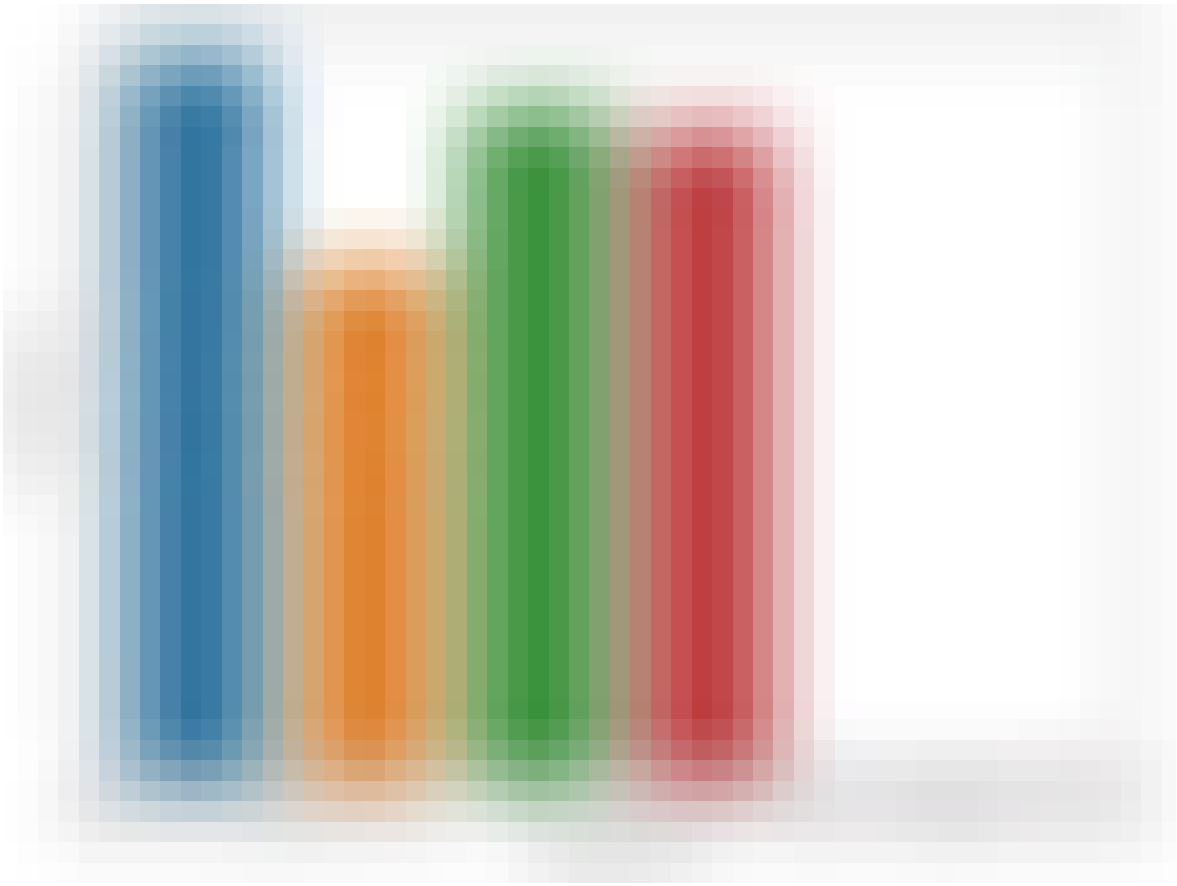
This code builds a deep learning model using the EfficientNetV2 architecture for image classification. The model is trained on a dataset using the fit() function in Keras. The training process is optimized using the Adam optimizer and the Sparse Categorical Crossentropy loss function, with accuracy as the metric. The model is saved using a callback function. Finally, the maximum accuracy achieved during the training process is recorded in a dictionary. Lets define what is Adam optimizer and the sparse categorical crossentropy loss functions.

Adam is a popular optimization algorithm used in deep learning that is designed to update the weights of a neural network in an efficient and adaptive way. It adapts the learning rate based on the gradient of the loss function and can speed up convergence and improve the performance of the model.

Sparse categorical cross-entropy loss function is a commonly used loss function in multi-class classification problems where the labels are integers. It is used to calculate the difference between the predicted class probabilities and the true class probabilities. The function works by computing the negative logarithm of the predicted probability for the true class label. It is called "sparse" because the labels are integers rather than one-hot encoded vectors.

Experimentation & Results:

All the above models are trained and tested against two images sizes and also compared against each other using visualization we have shown the results.



This chart shows use the algorithms on x-axis and their respective accuracies after resizing and reshaping them.
2)After developing the classifier we used categorical_accuracy in it. Which is a metric used to evaluate the performance of a multi-class classification model. It calculates the percentage of correctly predicted class labels out of all predictions made by the model.
In the case of a multi-class classification problem, where each input can only belong to one class, categorical_accuracy will return the percentage of instances where the predicted class label matches the true class label. So here are their results in plot graphs.
1) It is plotted Accuracies against Epochs.

We can observe that as the epochs increase the accuracy is slightly fluctuating.
2)It is plotted Loss against Epochs.

Here , as the epochs increasing the loss is reducing.

3)For hyper_parameter tuning time taken for few algorithms are ['Logistic Regression', 'Naive Bayes', 'KNN', 'Decision Tree']=[0, 0, 166.43, 3620.35].
4)For image sizes of 128 and 256, some algorithms may perform better than others. For example, decision trees and naive Bayes may perform better on smaller image sizes because they are less complex models. On the other hand, more complex models like SVM, logistic regression, and random forests may perform better on larger image sizes because they are better able to capture complex relationships between features.
When it comes to dataset size, again, some algorithms may perform better than others. For smaller datasets, simple models like decision trees and naive Bayes may perform better because they are less prone to overfitting. However, for larger datasets, more complex models like SVM, logistic regression, and random forests may perform better because they are better able to capture the underlying patterns in the data.

5)In our case , when dealing with small datasets, simpler algorithms such as Logistic Regression and Naive Bayes tend to perform better than more complex algorithms such as Random Forest and SVM. This is because simpler algorithms have fewer parameters to learn, making them less prone to overfitting on a small dataset.

Challenges and Solutions:
Challenges:
There are some challenges we've faced regarding accuracies and time as discussed below:
1. First of all, while training machine learning algorithm we aren't satisfied with the accuracies models were giving us.
2. We felt that there is some loss of relation between pixels while we're loading it for machine learning models and then using the same for deep learning model.
3. We were not satisfied with the speed of training deep learning models.
4. We were not satisfied with the accuracy that models were giving us.
5. Sometimes because of large data my googlecolab gpu has crashed for 256 images sizes not able to fix that as it took too long .
Solutions:
Here are some solutions that we tried to implement for the above challenges:
1. To deal with the first challenge we used a hyperparameter tuning technique known as GRIDSearch CV and eventually our accuracy increases.
2. For dealing with second challenge, we've used tensorflow to create dataset for deep learning architectures.
3. To deal with third challenge we've used a resize_rescale layer in the deep learning models.
4. And at last to deal with the forth problem we added a data augmentation layer and also there is callback which will change learning rate if there is no change or improvement in accuracy.

REFERENCES:

1. https://www.kaggle.com/datasets/aryashah2k/mango-leaf-disease-dataset
2. https://akshithasingareddy.wixsite.com/2022/post/image-classification-wild-animal-dataset
3. https://medium.com/@mohtedibf/indepth-parameter-tuning-for-decision-tree-6753118a03c3
4. https://medium.com/@mohtedibf/in-depth-parameter-tuning-for-knn-4c0de485baf6

5. https://medium.com/all-things-ai/in-depth-parameter-tuning-for-random-forest-d67bb7e920d
6. https://medium.com/all-things-ai/in-depth-parameter-tuning-for-svc-758215394769
7. https://www.tensorflow.org/api_docs/python/tf
8. https://scikit-learn.org/stable/modules/classes.html
9. https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-early-stopping/
10. https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/ReduceLROnPlateau
11. https://www.kaggle.com/code/michaelbryantds/titanic-survivor-classification-top-8-accuracy#Imputation,-Create-Dummies,-and-Scale
12. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

For Code link: https://github.com/saicharan1922/Image-Classification-Mango-Disease-Image-Dataset.git
Video description: will be added soon.