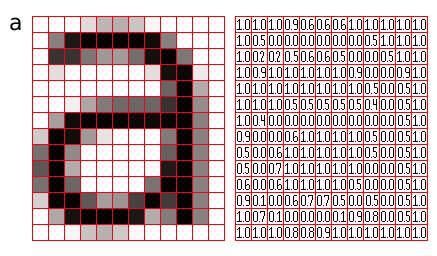
**Image Classification - Distinguishing Healthy and Diseases leaves**

A Convolutional Neural Network (CNN), which is also called ConvNet, is a type of neural network designed to process data that has a grid-like structure, such as digital images. Images are typically represented as grids of pixels, each of which has a value indicating its brightness and color. CNNs are particularly well-suited to handling this type of data.



*Representation of image as a grid of pixels*

The human brain can process a vast amount of visual information within a split second. Neurons in the brain are arranged in a way that enables them to collectively cover the entire visual field, with each neuron processing information within its own receptive field. Similarly, in a Convolutional Neural Network (CNN), each neuron processes data only within its receptive field. The layers in a CNN are designed to detect simpler visual patterns first, such as lines and curves, and then progressively identify more complex patterns like faces and objects. By utilizing CNNs, computers can also be trained to interpret visual information in a similar manner to humans.

Diagram, engineering drawing

Description automatically generated

*Architecture of a CNN*

## **Code Development:**

## Importing Libraries:

First step of any ML project is importing all the required libraries used for building our model, in our case we are building image classifier so we will import libraries like numpy, tensorflow and etc.

import tensorflow as tf

from tensorflow.keras import models, layers

import matplotlib.pyplot as plt

import numpy as np

Dataset:

For this model we are using a dataset from Kaggle named Mango leaves disease (https://www.kaggle.com/datasets/aryashah2k/mango-leaf-disease-dataset). This contains a total of 4000 images from 8 different types of diseases. Our goal is to use this dataset to train our model and test its accuracy on test dataset.

images\_dataset=tf.keras.preprocessing.image\_dataset\_from\_directory(

'archive',

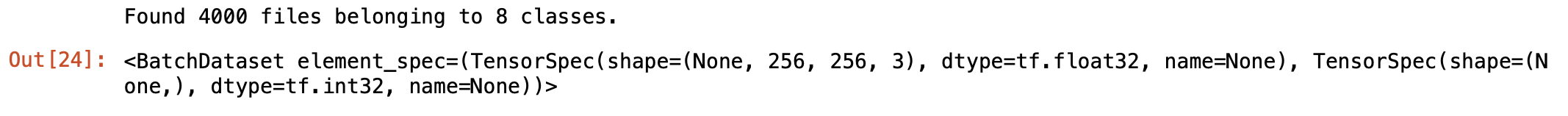
shuffle=True,

image\_size=(IMAGE\_SIZE,IMAGE\_SIZE),

batch\_size=BATCH\_SIZE,

)

images\_dataset



class\_names=images\_dataset.class\_names

class\_names

Background pattern

Description automatically generated with low confidence

#visualize the first image in that batch

plt.figure(figsize=(18,18))

for image\_batch, label\_batch in images\_dataset.take(1):

for i in range (12):

ax=plt.subplot(3,4, i+1)

plt.imshow(image\_batch[i].numpy().astype('uint8'))

plt.title(class\_names[label\_batch[i]])

plt.axis('off')

Storing the dataset into our jupyter notebook and visualising the dataset.

PowerPoint

Description automatically generated with low confidence

Splitting of Dataset:

Best way to split our data is to make our dataset into three parts i.e, train, validation and test datasets. This will help us to check our models real time performance. In our model we are going to make the split in 80:10:10 ratio.

I created a function to partition a TensorFlow dataset object (**ds**) into training, validation, and test sets based on the desired split percentages (**train\_split**, **val\_split**, **test\_split**). I can also specify whether to shuffle the dataset (**shuffle**) and the number of elements to use for shuffling (**shuffle\_size**).

To start, I calculate the size of the dataset (**ds\_size**). If the **shuffle** parameter is set to **True**, I shuffle the dataset using the **shuffle** method and a specified random seed (**seed=12**).

Next, I calculate the sizes of the training, validation, and test sets based on the provided split percentages. I create the training set by taking the first **train\_size** elements of the shuffled or original dataset (**ds**). To create the validation set, I skip the first **train\_size** elements and take the next **val\_size** elements. Finally, I create the test set by skipping the first **train\_size** and **val\_size** elements.

Lastly, I return the three sets as separate TensorFlow dataset objects: **train\_ds**, **val\_ds**, and **test\_ds**.

def get\_dataset\_partitions\_tf(ds,train\_split=0.8, val\_split=0.1, test\_split=0.1,shuffle=True, shuffle\_size=10000):

ds\_size=len(ds)

if shuffle:

ds=ds.shuffle(shuffle\_size,seed=12)

train\_size= int(train\_split\* ds\_size)

val\_size=int(val\_split\* ds\_size)

train\_ds=ds.take(train\_size)

val\_ds=ds.skip(train\_size).take(val\_size)

test\_ds=ds.skip(train\_size).skip(val\_size)

return train\_ds, val\_ds, test\_ds

Pre-processing of Data:

I tried to improve the performance of TensorFlow datasets during training by using the **cache()**, **shuffle()**, and **prefetch()** methods.

To start, I use the **cache()** method on the **train\_ds** dataset to keep the elements in memory after the first load. This helps to speed up training time. Then, I shuffle the cached dataset with a buffer size of 1000 using the **shuffle()** method. Finally, I use the **prefetch()** method to prefetch elements from the dataset to improve training performance. By using **tf.data.experimental.AUTOTUNE** as the buffer size parameter, the system automatically tunes the buffer size based on available memory and other factors.

The same operations are performed on **val\_ds** and **test\_ds** to improve their performance during validation and testing.

train\_ds=train\_ds.cache().shuffle(1000).prefetch(buffer\_size=tf.data.experimental.AUTOTUNE)

val\_ds=val\_ds.cache().shuffle(1000).prefetch(buffer\_size=tf.data.experimental.AUTOTUNE)

test\_ds=test\_ds.cache().shuffle(1000).prefetch(buffer\_size=tf.data.experimental.AUTOTUNE)

By defining a Keras Sequential model named **resize\_and\_rescale**, I tried to pre-process images by resizing and rescaling them to a specified size.

This Sequential model consists of two layers of pre-processing operations that are applied sequentially. The first layer is a **Resizing** layer from the **layers.experimental.preprocessing** module, which can resize the input image to the dimensions specified by the **IMAGE\_SIZE** variable in both height and width.

The second layer is a **Rescaling** layer, also from the **layers.experimental.preprocessing** module, which can scale down the pixel values of the input image by a factor of 1/255. This is done to ensure that the pixel values of the image are in the range of [0, 1], which is a commonly used range in machine learning applications.

By using these two layers in the **resize\_and\_rescale** model, we can first resize the input image to the desired dimensions and then rescale its pixel values to an appropriate range for training a machine learning model.

resize\_and\_rescale=tf.keras.Sequential([

layers.experimental.preprocessing.Resizing(IMAGE\_SIZE,IMAGE\_SIZE),

layers.experimental.preprocessing.Rescaling(1.0/255)

])

Then I one more feature from Keras Sequential model called data\_augmentation, which applies random transformations to images during training to increase the diversity of the data and improve model performance.

The Sequential model consists of two layers of data augmentation operations applied in sequence. The first layer is a RandomFlip layer from the layers.experimental.preprocessing module. This layer randomly flips the input image horizontally and vertically, which helps the model learn to recognize objects from different viewpoints and angles.

The second layer is a RandomRotation layer, which randomly rotates the input image by a degree between -0.2 and 0.2. This helps the model learn to recognize objects in different orientations and angles.

Together, these two layers in the data\_augmentation model introduce randomness and variability to the input images during training, which can help prevent overfitting and improve the generalization performance of the model.

data\_augmentation=tf.keras.Sequential([

layers.experimental.preprocessing.RandomFlip("horizontal\_and\_vertical"),

layers.experimental.preprocessing.RandomRotation(0.2)

])

## Building the Model:

First, I set the input shape of the images to be fed into the model as a tuple containing the batch size, image size, and number of channels.

Then, I specified the number of classes for the model to classify the input images into.

Next, I created a Sequential model using Keras, which allows me to build a deep learning model layer by layer.

I added two layers to preprocess the input images by resizing and rescaling them, followed by data augmentation to artificially increase the number of training images.

Then, I added six layers of 2D convolutional neural networks with various kernel sizes and activation functions, each followed by a max pooling layer to downsample the feature maps.

Afterward, I added a flatten layer to convert the feature maps to a one-dimensional vector, followed by two dense layers, each with 64 units and a ReLU activation function, and the final dense layer with a softmax activation function to output the class probabilities.

Finally, I built the model with the specified input shape.

#bulding the model

input\_shape=(BATCH\_SIZE, IMAGE\_SIZE, IMAGE\_SIZE, CHANNELS)

n\_classes=8

model=models.Sequential ( [

resize\_and\_rescale,

data\_augmentation,

layers. Conv2D(32, (3, 3), activation='relu',input\_shape=input\_shape),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, kernel\_size=(3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, kernel\_size=(3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers. Conv2D(64, (3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers. Conv2D(64, (3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers. Conv2D(64, (3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers. Flatten(),

layers.Dense (64, activation='relu'),

layers. Dense(n\_classes, activation='softmax'),

])

model.build(input\_shape=input\_shape)

Summary of our model

Table

Description automatically generated

A picture containing table

Description automatically generated

To compile the model, I used the compile() method to specify the optimizer, loss function, and metrics for training. I chose to use the adam optimizer, SparseCategoricalCrossentropy loss function, and track accuracy during training by setting the metrics parameter to ['accuracy'].

#compiling the model

model.compile(

optimizer='adam',

loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=False),

metrics=['accuracy']

)

To train the model, I used the fit() method which takes in the training and validation data, as well as parameters such as the number of epochs, batch size, and verbosity level.

In this case, I trained the model on the train\_ds dataset for EPOCHS number of epochs, with a batch size of BATCH\_SIZE, and set the verbosity level to 1 to see the progress during training. The validation data val\_ds was also passed to monitor the performance of the model on unseen data during training.

#train the model

history=model.fit(

train\_ds,

epochs=EPOCHS,

batch\_size=BATCH\_SIZE,

verbose=1,

validation\_data=val\_ds

)

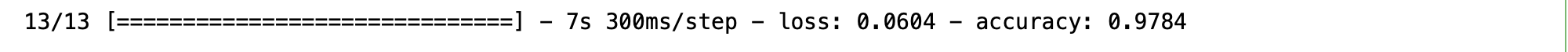
A picture containing text

Description automatically generated

Checking our model accuracy with test dataset.

#evaluate the model

scores=model.evaluate(test\_ds)



I extracted the training and validation accuracy as well as the training and validation loss from the history object returned by the fit() method of the model.

I stored the training accuracy in the variable acc, validation accuracy in val\_acc, training loss in loss, and validation loss in val\_loss.

acc=history.history['accuracy']

val\_acc=history.history['val\_accuracy']

loss=history.history['loss']

val\_loss=history.history['val\_loss']

I plotted the training and validation accuracy and loss graphs for the trained model using Matplotlib with the history callback.

To plot the training and validation accuracy, I created a figure with a size of 8 by 8 and added a subplot of 1 row and 2 columns. Then, I plotted the training and validation accuracy values over the number of epochs. I added a legend to the graph and titled it "Training and Validation Accuracy".

Next, I plotted the training and validation loss values over the number of epochs by creating a second subplot in the same figure. I also added a legend to this graph and titled it "Training and Validation Loss".

#Plotting Accuracy and Loss Graph for Trained Model using Matplotlib with History Callback

#ploting training and validation accuracy

plt.figure(figsize=(8,8))

plt.subplot (1,2,1)

plt.plot(range (EPOCHS), acc, label='Training Accuracy')

plt.plot(range (EPOCHS), val\_acc, label='Validation Accuracy')

plt.legend (loc='lower right')

plt.title('Training and Validation Accuracy')

#ploting training and validation Loss.

plt.subplot (1,2,2)

plt.plot(range (EPOCHS), loss, label='Training loss')

plt.plot(range (EPOCHS), val\_loss, label='Validation loss')

plt.legend (loc='upper right')

plt.title('Training and Validation Loss')

Chart, histogram

Description automatically generated

To make predictions on a batch of images using the trained model, we can iterate over the batches in the test dataset. In this example, we take the first batch and select the first image and its label. We then display the image and print its true label. Next, we use the model to predict the labels for all the images in the batch and select the prediction for the first image. We print this predicted label.

#making predictions

for images\_batch, label\_batch in test\_ds.take (1):

image1=images\_batch[0].numpy().astype('uint8')

label1=label\_batch[0].numpy()

print("predicting the first image")

plt.imshow(image1)

print('image1 True Label: ',class\_names[label1])

batch\_prediction= model.predict(images\_batch)

print("Image1's predicted label: ", class\_names [np.argmax (batch\_prediction [0])])

Chart

Description automatically generated

I created a function called **predict()** that takes in a trained model and an image as inputs. Inside the function, I converted the image to a numpy array and then added an extra dimension to it using **tf.expand\_dims()** so that it can be passed to the model. I used the model to make a prediction on the image and then used **np.argmax()** to get the index of the predicted class with the highest probability. I looked up the name of the predicted class from the **class\_names** list and rounded off the confidence to two decimal places. Finally, I returned the predicted class and the confidence value.

#function to predict with confidence

def predict (model, img):

img\_array=tf.keras.preprocessing.image.img\_to\_array(images [i].numpy())

img\_array=tf.expand\_dims (img\_array,0)

predictions=model.predict (img\_array)

predicted\_class=class\_names [np.argmax(predictions[0])]

confidence=round (100\* (np.max (predictions [0])), 2)

return predicted\_class, confidence

I am creating a figure using the plt.figure() function with a size of 15x15. Then I am looping through the test dataset and displaying the first 9 images and their corresponding actual and predicted labels using the plt.subplot() function. For each image, I am calling the predict() function to get the predicted class and confidence score. Finally, I am setting the title of each subplot with the actual and predicted label and confidence score and turning off the axis using the plt.title() and plt.axis() functions.

plt.figure(figsize=(15,15))

for images, labels in test\_ds.take (1):

for i in range (9):

ax=plt.subplot (3,3,i+1)

plt.imshow(images [i].numpy().astype("uint8"))

predicted\_class, confidence=predict (model, images[i].numpy())

actual\_class=class\_names[labels[i]]

plt.title (f"Actual: {actual\_class}, \n Predicted: {predicted\_class}.\n Confidence: {confidence}%")

plt.axis("off")

A picture containing diagram

Description automatically generated

## Improving the Model:

In the first model, I included six convolutional layers, while the second model only had four. I also used fewer filters in each of the convolutional layers in the second model. To prevent overfitting, I included a dropout layer in the second model, which was not present in the first model.

For the dense layers, the first model had two layers, one with 64 units and the second with n\_classes units. The second model only had one dense layer, which had 256 units, along with a dropout layer.

Overall, I believe both models are capable of performing image classification tasks. However, I believe that the second model might perform slightly better due to the added dropout layer, which helps prevent overfitting.

# Define the input shape for the model

input\_shape = (BATCH\_SIZE, IMAGE\_SIZE, IMAGE\_SIZE, CHANNELS)

# Define the number of classes in the dataset

n\_classes = 8

# Define the Sequential model and add layers to it

model = models.Sequential([

# Preprocessing layers

resize\_and\_rescale,

data\_augmentation,

# Convolutional layers

layers.Conv2D(32, kernel\_size=(3, 3), activation='relu', input\_shape=input\_shape),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(64, kernel\_size=(3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(128, kernel\_size=(3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

layers.Conv2D(256, kernel\_size=(3, 3), activation='relu'),

layers.MaxPooling2D((2, 2)),

# Flatten layer

layers.Flatten(),

# Dense layers

layers.Dense(256, activation='relu'),

layers.Dropout(0.5),

layers.Dense(n\_classes, activation='softmax'),

])

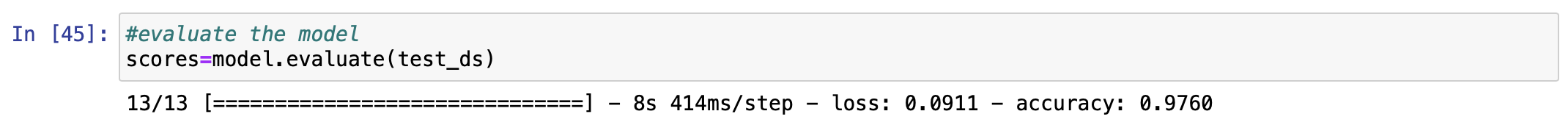
# Build the model with the input shape

model.build(input\_shape)

After updating layers and using early stopping accuracy went down form 97.8% to 97.6%

Chart, line chart

Description automatically generated



## Building the Model with Decision tree to compare accuracy:

The decision tree is a versatile supervised learning algorithm used for classification and regression tasks. It creates a tree structure by recursively partitioning data based on the most significant feature until the subsets are pure or the maximum tree depth is reached. In classification problems, subsets contain data points of the same class, while in regression problems, subsets minimize variance. New data points are predicted by traversing the tree based on feature values. Although easy to understand and handle both categorical and numerical data, decision trees are prone to overfitting, sensitive to small data changes, and can be computationally expensive for large datasets.

Looping through the classes of a dataset and retrieve the images for each class. First, I used the **os.path.join()** method to join the class name with the data path to create a class path. I checked if the class path is a directory and if not, I skipped it.

Next, I looped through the images in each class path using **os.listdir()**. I added a condition to skip hidden files such as **.DS\_Store**. I then joined the image name with the class path to get the image path and used the **cv2.imread()** method to read the image in grayscale format. I resized each image to 64x64 pixels using **cv2.resize()** method and appended them to the **images** list. I also appended the corresponding label of each image to the **labels** list.

for label, class\_name in enumerate(classes):

class\_path = os.path.join(data\_path, class\_name)

if not os.path.isdir(class\_path):

continue

for image\_name in os.listdir(class\_path):

if image\_name == '.DS\_Store': # skip hidden file

continue

image\_path = os.path.join(class\_path, image\_name)

image = cv2.imread(image\_path, cv2.IMREAD\_GRAYSCALE)

image = cv2.resize(image, (64, 64)) # Resize the images

images.append(image)

labels.append(label)

Building decision tree.

# Train the decision tree

clf = DecisionTreeClassifier()

clf.fit(train\_features, train\_labels)



Testing the accuracy.

# Test the model

predictions = clf.predict(test\_features)

accuracy = accuracy\_score(test\_labels, predictions)

print("Accuracy:", accuracy)



Accuracy of Decision tree is 95% where as accuracy using CNN is 97%.

**Contribution:**

* Splitting the dataset into train, test and validation.
* Tried to improve the accuracy by making changes to convolution and dense layers.
* Using early stopping to improve a model's generalization performance and prevent overfitting
* Comparing different models and checking there accuracies.

**Challenges:**

* Epochs taking too long to run consumed lot of time to compare different combination of layers for better accuracy.
* Comparing different models caused lot of modifications to be done to the format of the data and figuring out the approach was a bit challenging and time taking.

## **References:**

* <https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939>
* <https://www.tutorialspoint.com/how-can-tensorflow-be-used-to-configure-the-dataset-for-performance>
* <https://www.tensorflow.org/api_docs/python/tf/keras/layers/Rescaling>
* <https://www.tensorflow.org/tutorials/images/data_augmentation>
* <https://github.com/rehema-HM/yeesi104-Machine-Vision-in-Agriculture/blob/main/yeesi104/Tomato%20leaf%20classification.ipynb>
* <https://www.youtube.com/watch?v=kAV_F6uIVIk>
* <https://www.youtube.com/watch?v=ZITbTgwTDro>
* <https://www.youtube.com/watch?v=NJdz2uEMwf0>
* https://towardsdatascience.com/decision-tree-in-machine-learning-e380942a4c96