Tomato Plant Disease Detection and Classification Using Convolutional Neural Network.

CNNs are a type of artificial neural network that are particularly well-suited for image classification tasks, so they could potentially be a good choice for this problem. To use a CNN for tomato plant disease classification, you will need to collect a dataset of images of healthy and diseased tomato plants. You will then need to label the images with the correct disease class, if applicable. Once you have a dataset, you can then use it to train a CNN to classify the diseases in the images.

There are several steps involved in this process:

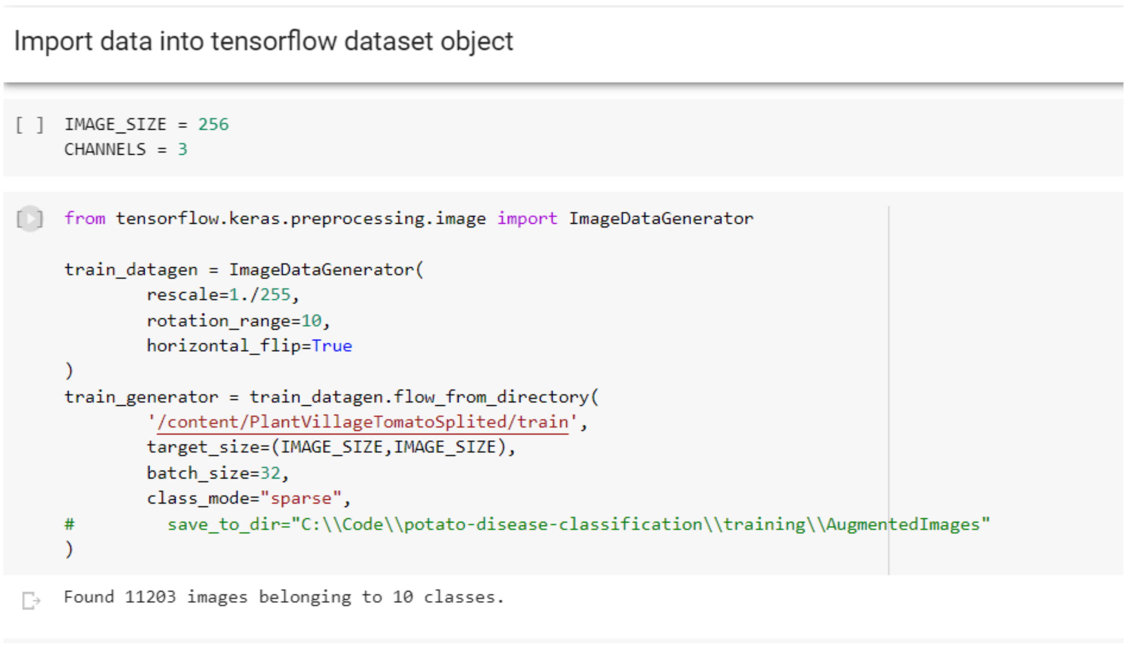
- Preprocessing: You will need to preprocess the images in your dataset by resizing them to a consistent size and possibly also applying some basic image transformations (e.g., cropping, rotating).
- Splitting the dataset: You will need to split your dataset into a training set, a validation set, and a test set. The training set is used to train the CNN, the validation set is used to tune the model's hyperparameters, and the test set is used to evaluate the model's performance.
- Building the CNN: You will need to design and build the CNN architecture. This will involve choosing the number and size of the convolutional and fully connected layers, as well as the type of activation functions to use.
- Training the CNN: Once the CNN is built, you will need to train it using the training set. During training, the model will learn to map the input images to the correct disease labels.
- Evaluating the CNN: After training, you can evaluate the CNN's performance on the test set to see how well it generalizes to unseen data.

ABOUT DATA SET:

The dataset used for this project consists of 11203 images, with 10 classes. There are 10 classes named:

- 'Tomato_Bacterial_spot',
- 'Tomato_Early_blight',
- 'Tomato_Late_blight',
- 'Tomato_Leaf_Mold',
- 'Tomato_Septoria_leaf_spot',
- 'Tomato_Spider_mites_Two_spotted_spider_mite',
- 'Tomato__Target_Spot',
- 'Tomato__Tomato_YellowLeaf__Curl_Virus',
- 'Tomato__Tomato_mosaic_virus',
- 'Tomato_healthy'

This code snippet uses the “ImageDataGenerator” class from the TensorFlow's “tf.keras.preprocessing.image” module to create a generator for training data. The generator applies a series of random transformations to the training images, such as rescaling, rotation, and horizontal flipping. These transformations help to augment the training data and can improve the generalization of the model. The generator is created using the “flow_from_directory” method, which reads the training images from a specified directory and automatically labels them based on their subdirectory names. The “target_size” parameter specifies the size to which the images should be resized, and the “batch_size” parameter specifies the number of images to include in each batch. The “class_mode” parameter specifies the format in which the labels should be returned. In this case, it is set to "sparse", which means that the labels will be returned as integer labels rather than one-hot vectors.



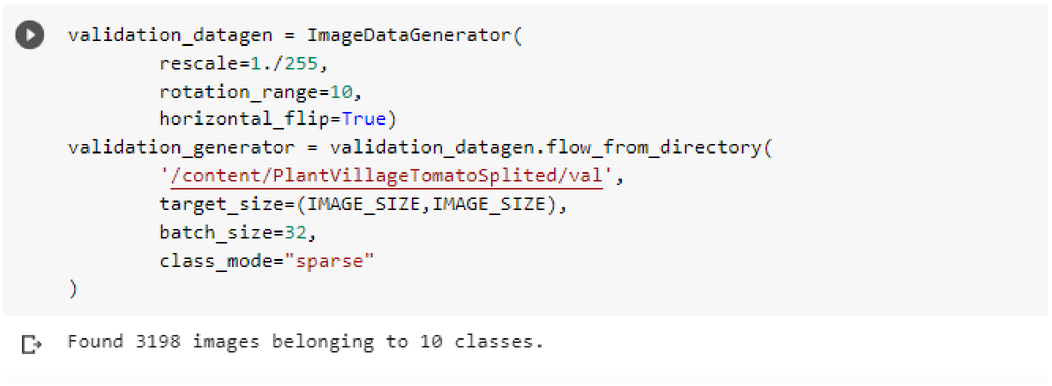
Import data into tensorflow dataset object

```
IMAGE_SIZE = 256
CHANNELS = 3
```

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=10,
        horizontal_flip=True
)
train_generator = train_datagen.flow_from_directory(
        '/content/PlantVillageTomatoSplited/train',
        target_size=(IMAGE_SIZE,IMAGE_SIZE),
        batch_size=32,
        class_mode="sparse",
#         save_to_dir="C:\\Code\\potato-disease-classification\\training\\AugmentedImages"
)
```

```
Found 11203 images belonging to 10 classes.
```

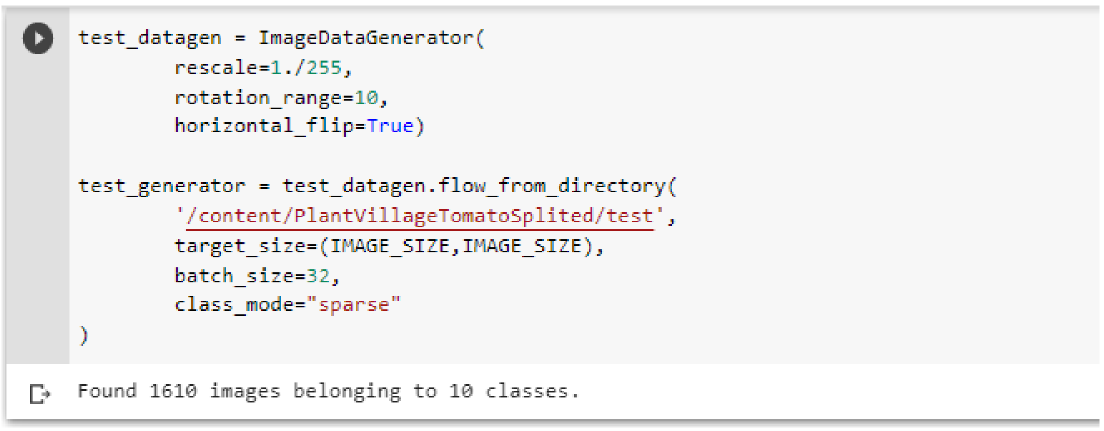
VALIDATIION SET:



```
validation_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=10,
        horizontal_flip=True)
validation_generator = validation_datagen.flow_from_directory(
        '/content/PlantVillageTomatoSplited/val',
        target_size=(IMAGE_SIZE,IMAGE_SIZE),
        batch_size=32,
        class_mode="sparse"
)
```

```
Found 3198 images belonging to 10 classes.
```

TEST SET:



```
test_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=10,
        horizontal_flip=True)

test_generator = test_datagen.flow_from_directory(
        '/content/PlantVillageTomatoSplited/test',
        target_size=(IMAGE_SIZE,IMAGE_SIZE),
        batch_size=32,
        class_mode="sparse"
)
```

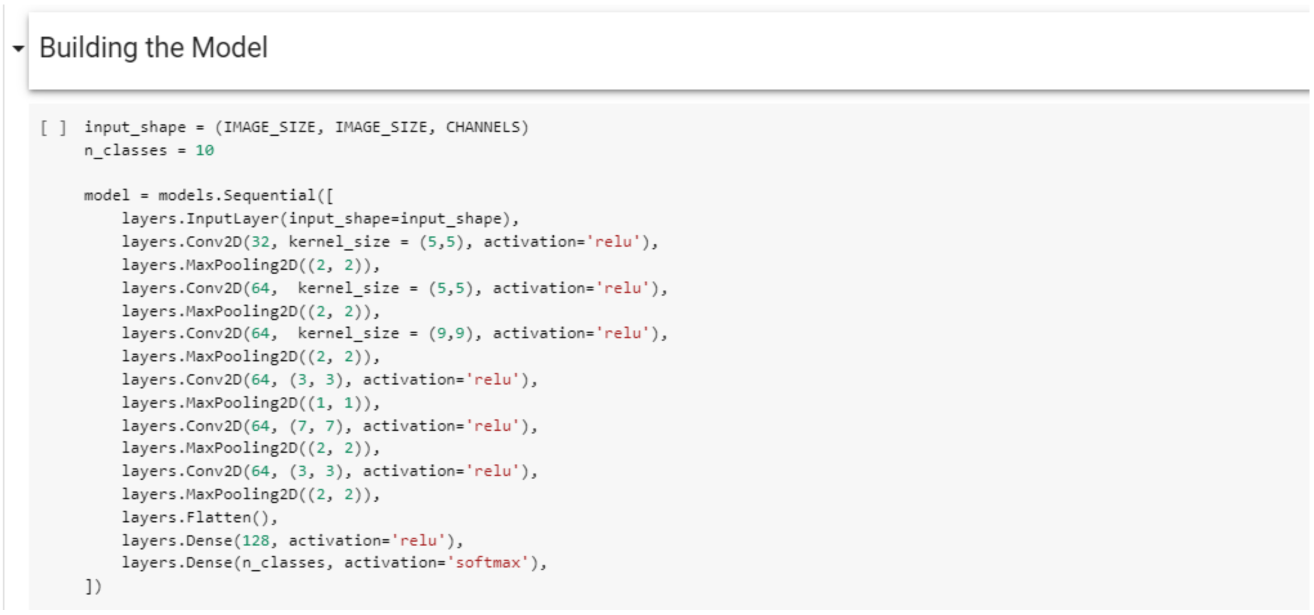
```
Found 1610 images belonging to 10 classes.
```

BUILDING MODEL:

The objective was to use you registration number in inverse order like (MINE Registration number is= 363944 and the inverse order is=449363). The model has convolutional layers with different kernel sizes of my registration number (if even add +1) or (for odd, use as it is), which are used to extract features from the input images. The kernel size specifies the size of the convolutional window that slides over the input image. Each convolutional layer also has a activation function, which is used to introduce non-linearity into the model. In this case, the activation function used is the ReLU (Rectified Linear Unit) function.

The model also has max pooling layers which are used to down-sample the input image and reduce its dimensionality. This helps to reduce the computational complexity of the model and also helps to reduce overfitting.

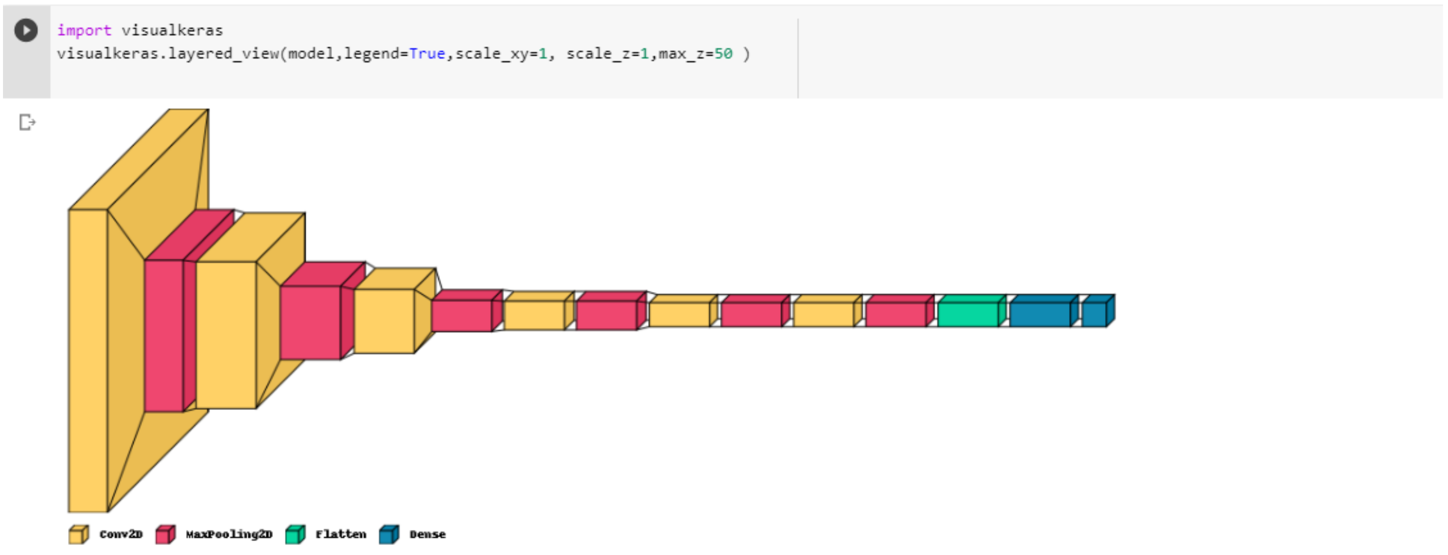
After the convolutional and max pooling layers, the model has a flatten layer, which flattens the output of the previous layer into a 1D vector. This is followed by two fully-connected (Dense) layers, which perform the final classification of the input image. The final layer has a softmax activation function, which outputs a probability distribution over the possible classes.



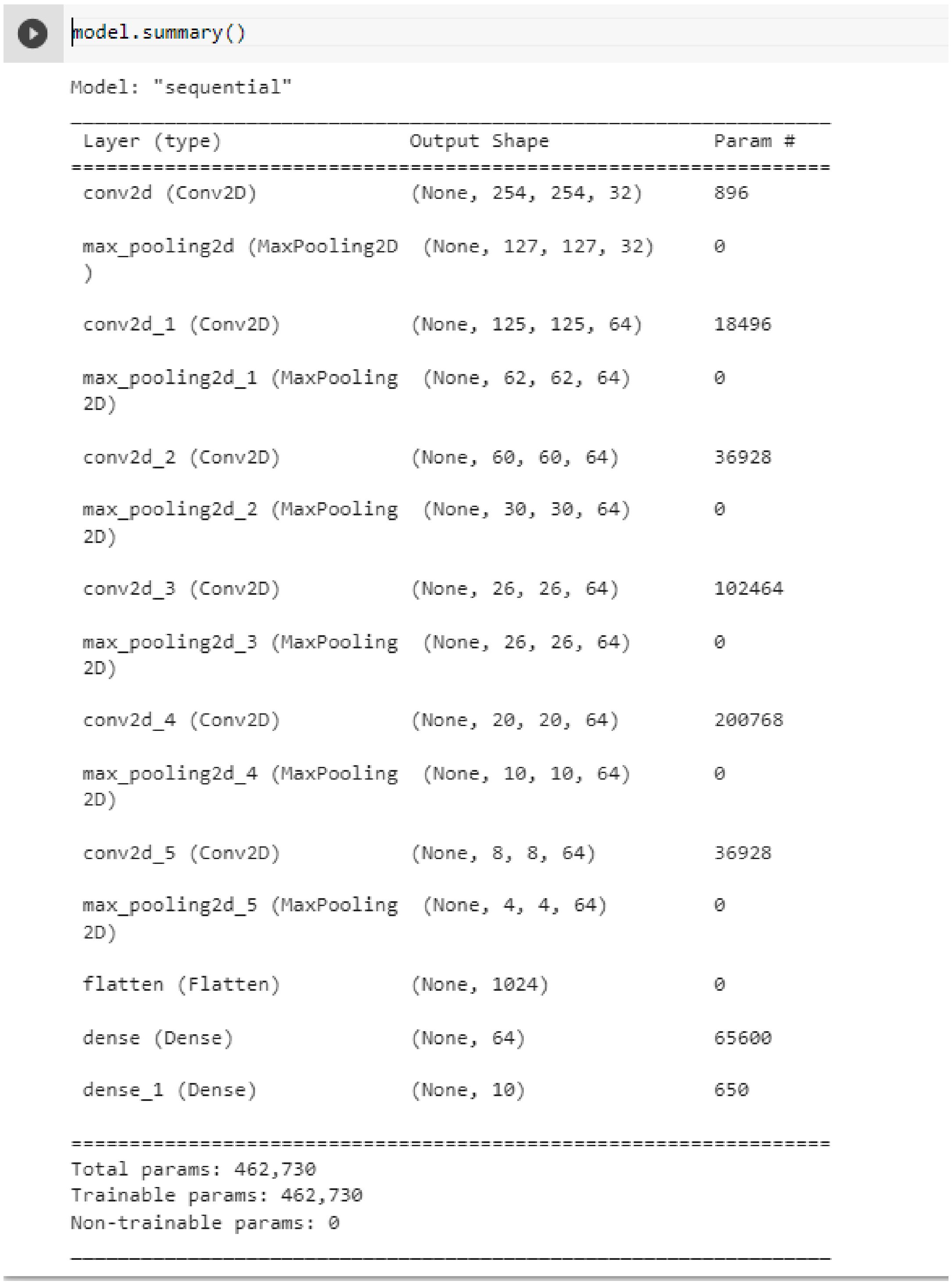
Building the Model

```
input_shape = (IMAGE_SIZE, IMAGE_SIZE, CHANNELS)
n_classes = 10

model = models.Sequential([
    layers.InputLayer(input_shape=input_shape),
    layers.Conv2D(32, kernel_size = (5,5), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64,  kernel_size = (5,5), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64,  kernel_size = (9,9), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((1, 1)),
    layers.Conv2D(64, (7, 7), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(n_classes, activation='softmax'),
])
```

3D plot of the model architecture, showing the names of each layer. The x and y axes will be scaled by a factor of 1, and the z axis will be scaled by a factor of 1 and will have a maximum value of 50.



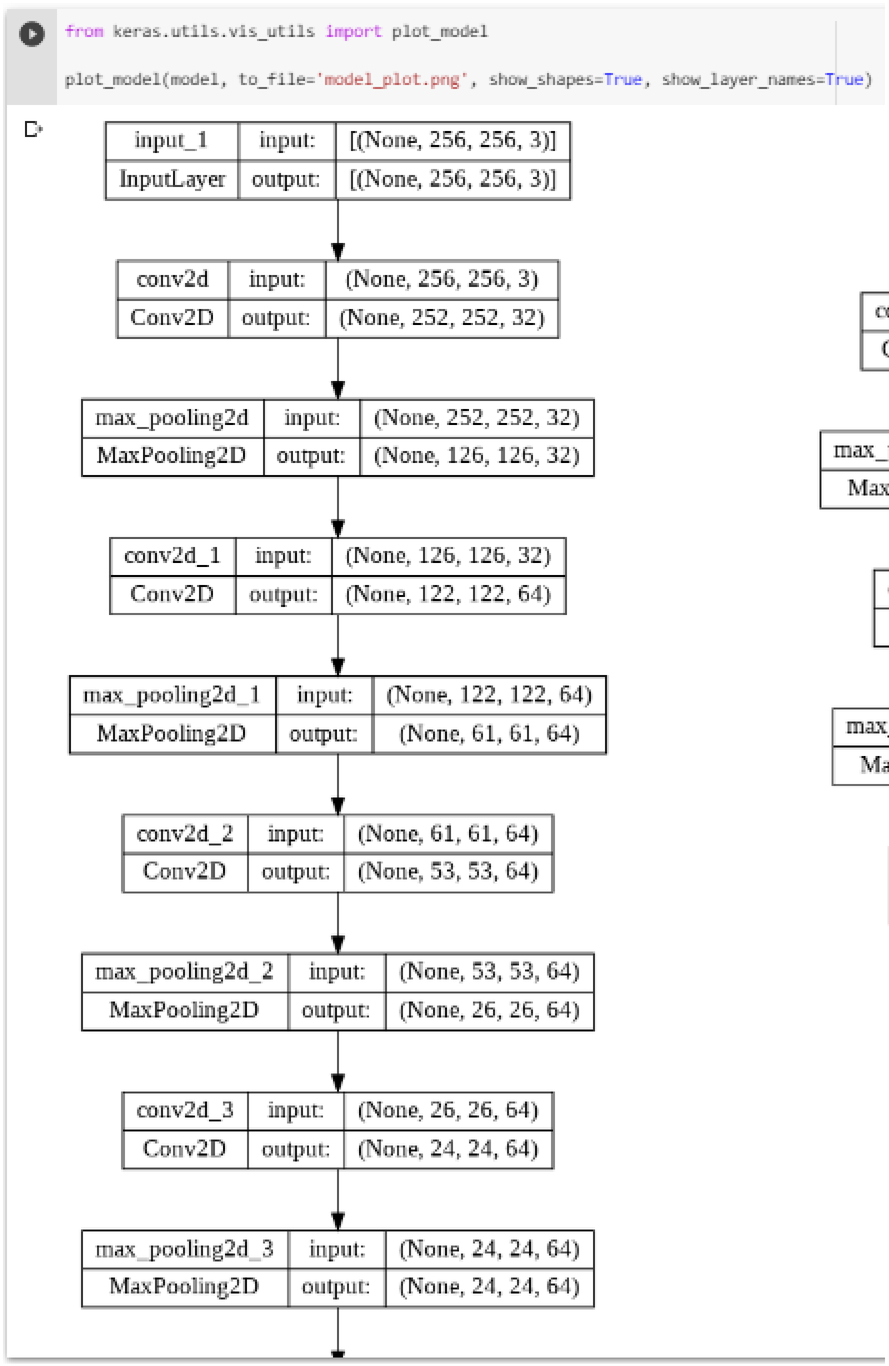
```
import visualkeras
visualkeras.layered_view(model,legend=True,scale_xy=1, scale_z=1,max_z=50 )
```



```
model.summary()
```

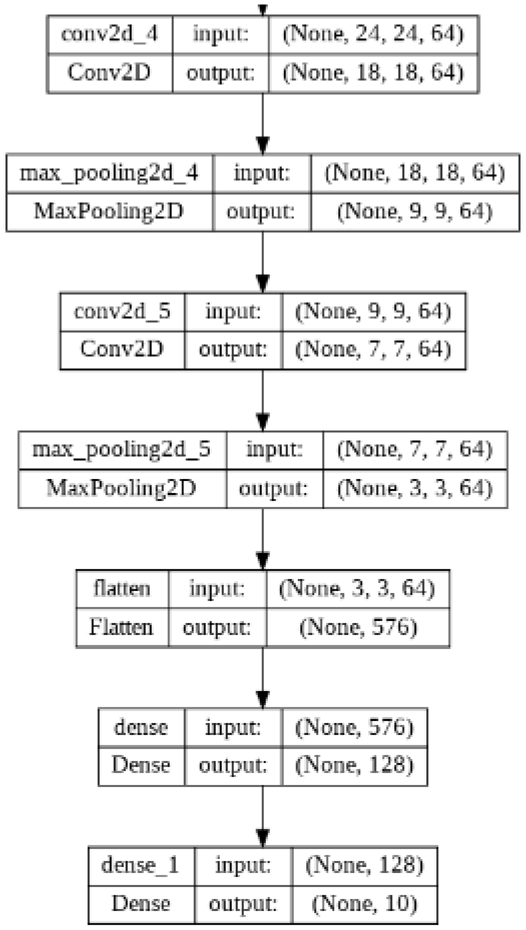
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 254, 254, 32)      896

 max_pooling2d (MaxPooling2D  (None, 127, 127, 32)     0
 )

 conv2d_1 (Conv2D)           (None, 125, 125, 64)      18496

 max_pooling2d_1 (MaxPooling  (None, 62, 62, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 60, 60, 64)        36928

 max_pooling2d_2 (MaxPooling  (None, 30, 30, 64)       0
 2D)

 conv2d_3 (Conv2D)           (None, 26, 26, 64)        102464

 max_pooling2d_3 (MaxPooling  (None, 26, 26, 64)       0
 2D)

 conv2d_4 (Conv2D)           (None, 20, 20, 64)        200768

 max_pooling2d_4 (MaxPooling  (None, 10, 10, 64)       0
 2D)

 conv2d_5 (Conv2D)           (None, 8, 8, 64)          36928

 max_pooling2d_5 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 64)                65600

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 462,730
Trainable params: 462,730
Non-trainable params: 0
_________________________________________________________________
```

This plot is the model architecture The plot shows the output shapes of each layer and the names of each layer.



```
from keras.utils.vis_utils import plot_model

plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=True)
```

input_1 | InputLayer | input: [(None, 256, 256, 3)] | output: [(None, 256, 256, 3)]
conv2d | Conv2D | input: (None, 256, 256, 3) | output: (None, 252, 252, 32)
max_pooling2d | MaxPooling2D | input: (None, 252, 252, 32) | output: (None, 126, 126, 32)
conv2d_1 | Conv2D | input: (None, 126, 126, 32) | output: (None, 122, 122, 64)
max_pooling2d_1 | MaxPooling2D | input: (None, 122, 122, 64) | output: (None, 61, 61, 64)
conv2d_2 | Conv2D | input: (None, 61, 61, 64) | output: (None, 53, 53, 64)
max_pooling2d_2 | MaxPooling2D | input: (None, 53, 53, 64) | output: (None, 26, 26, 64)
conv2d_3 | Conv2D | input: (None, 26, 26, 64) | output: (None, 24, 24, 64)
max_pooling2d_3 | MaxPooling2D | input: (None, 24, 24, 64) | output: (None, 24, 24, 64)



conv2d_4 | Conv2D | input: (None, 24, 24, 64) | output: (None, 18, 18, 64)
max_pooling2d_4 | MaxPooling2D | input: (None, 18, 18, 64) | output: (None, 9, 9, 64)
conv2d_5 | Conv2D | input: (None, 9, 9, 64) | output: (None, 7, 7, 64)
max_pooling2d_5 | MaxPooling2D | input: (None, 7, 7, 64) | output: (None, 3, 3, 64)
flatten | Flatten | input: (None, 3, 3, 64) | output: (None, 576)
dense | Dense | input: (None, 576) | output: (None, 128)
dense_1 | Dense | input: (None, 128) | output: (None, 10)

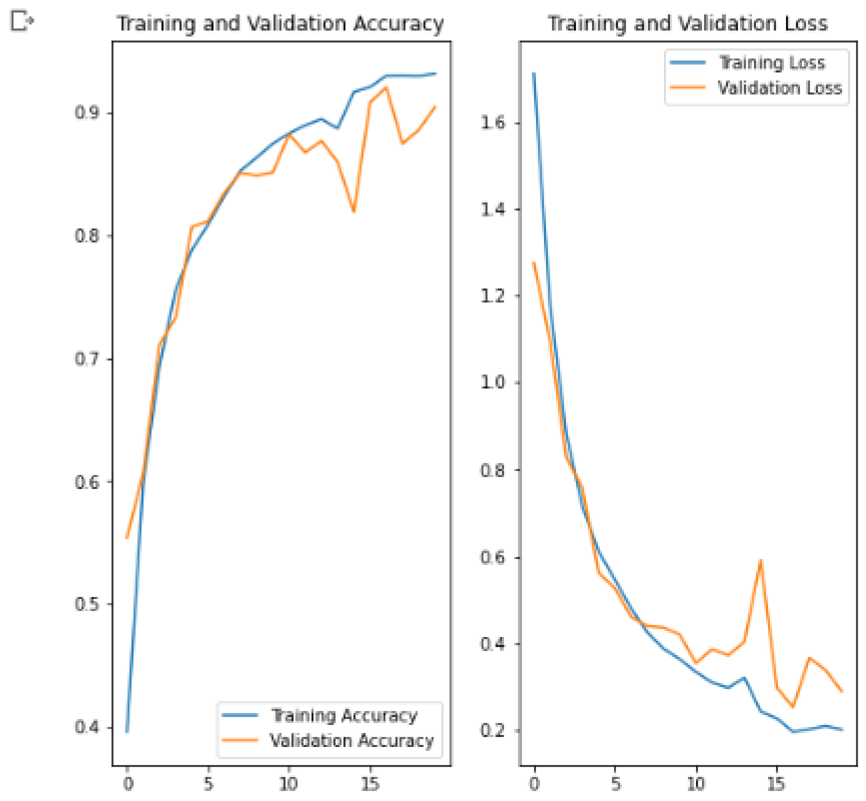
Compiling the Model:

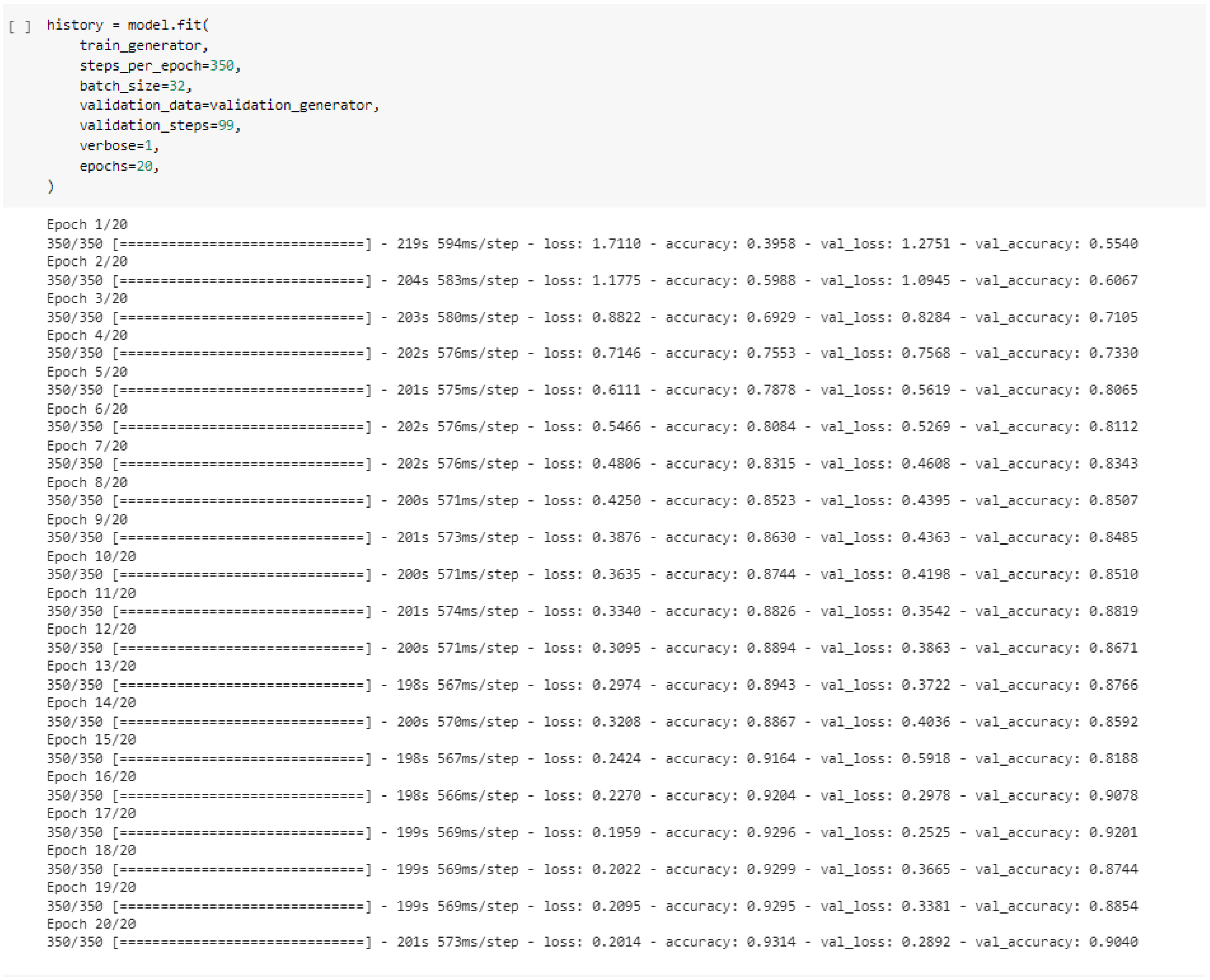
In machine learning, the optimizer is a crucial component of the training process. It determines how the model's parameters will be updated based on the loss function and the input data. I use Adam optimizer which is a popular choice for training deep learning models because it is computationally efficient and has been shown to work well in a wide range of applications. The loss function is used to measure how well the model is performing. The SparseCategoricalCrossentropy loss is a loss function that is often used for classification tasks with a single label per sample. It is similar to the categorical cross-entropy loss, but it is used when the classes are encoded as integers rather than one-hot vectors. The metric is a function that is used to evaluate the model's performance. The accuracy metric is a simple and widely used metric for classification tasks, which measures the fraction of correctly classified samples. In summary, the Adam optimizer will be used to update the model's parameters based on the SparseCategoricalCrossentropy loss and the model's performance will be evaluated using the accuracy metric.

The reason why I use this for tomato plant disease detection and classification, because Adam optimizer is to minimize the loss function, which measures the difference between the model's predicted labels and the true labels. By minimizing the loss, the model will learn to classify the diseases correctly.

RESULTS:

The results show 92% accuracy. A decrease in the training loss and an increase in the training accuracy indicate that the model is learning and improving. Similarly, a decrease in the validation loss and an increase in the validation accuracy indicate that the model is generalizing well to unseen data. As shown in Graphs.





```
[ ] history = model.fit(
        train_generator,
        steps_per_epoch=350,
        batch_size=32,
        validation_data=validation_generator,
        validation_steps=99,
        verbose=1,
        epochs=20,
    )
```

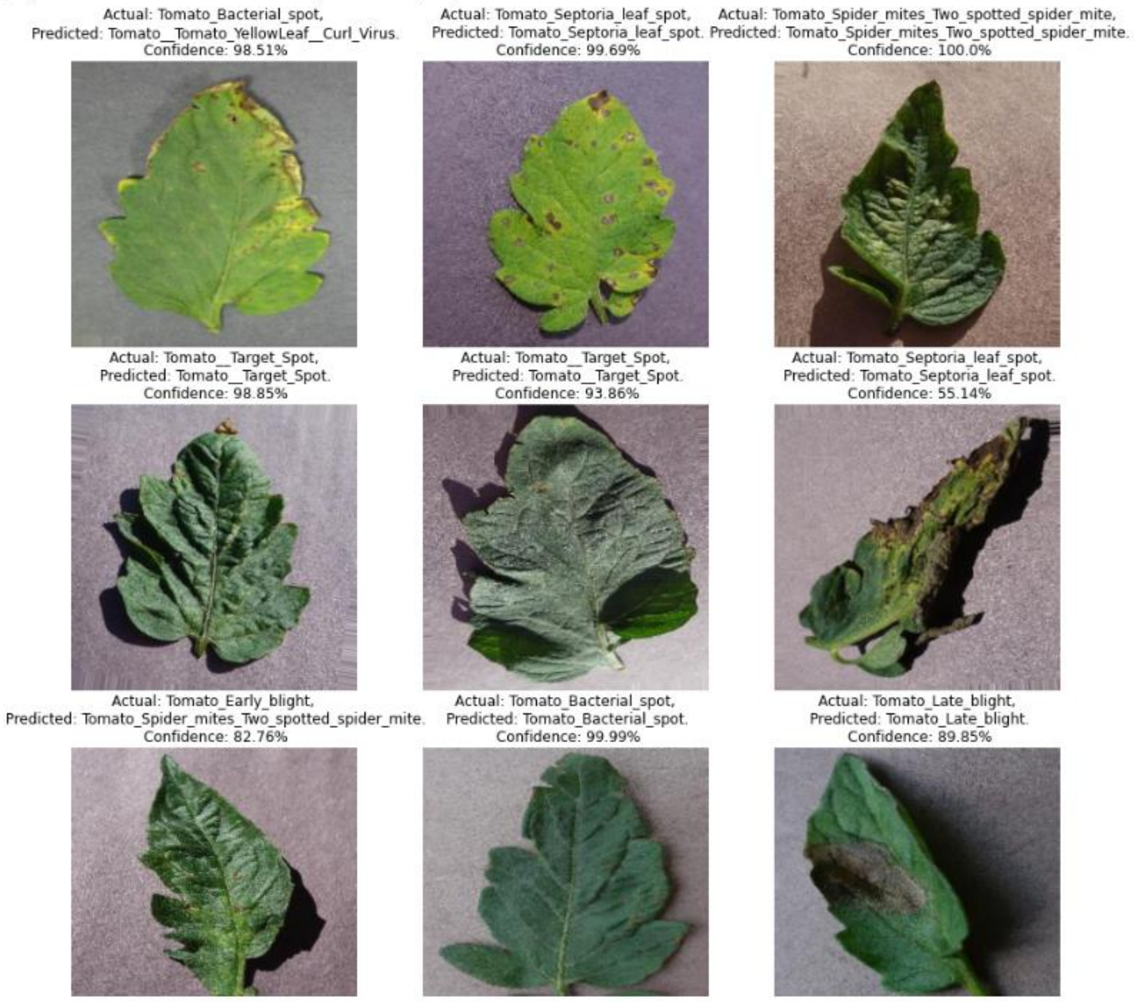
```
Epoch 1/20
350/350 [==============================] - 219s 594ms/step - loss: 1.7110 - accuracy: 0.3958 - val_loss: 1.2751 - val_accuracy: 0.5540
Epoch 2/20
350/350 [==============================] - 204s 583ms/step - loss: 1.1775 - accuracy: 0.5988 - val_loss: 1.0945 - val_accuracy: 0.6067
Epoch 3/20
350/350 [==============================] - 203s 580ms/step - loss: 0.8822 - accuracy: 0.6929 - val_loss: 0.8284 - val_accuracy: 0.7105
Epoch 4/20
350/350 [==============================] - 202s 576ms/step - loss: 0.7146 - accuracy: 0.7553 - val_loss: 0.7568 - val_accuracy: 0.7330
Epoch 5/20
350/350 [==============================] - 201s 575ms/step - loss: 0.6111 - accuracy: 0.7878 - val_loss: 0.5619 - val_accuracy: 0.8065
Epoch 6/20
350/350 [==============================] - 202s 576ms/step - loss: 0.5466 - accuracy: 0.8084 - val_loss: 0.5269 - val_accuracy: 0.8112
Epoch 7/20
350/350 [==============================] - 202s 576ms/step - loss: 0.4806 - accuracy: 0.8315 - val_loss: 0.4608 - val_accuracy: 0.8343
Epoch 8/20
350/350 [==============================] - 200s 571ms/step - loss: 0.4250 - accuracy: 0.8523 - val_loss: 0.4395 - val_accuracy: 0.8507
Epoch 9/20
350/350 [==============================] - 201s 573ms/step - loss: 0.3876 - accuracy: 0.8630 - val_loss: 0.4363 - val_accuracy: 0.8485
Epoch 10/20
350/350 [==============================] - 200s 571ms/step - loss: 0.3635 - accuracy: 0.8744 - val_loss: 0.4198 - val_accuracy: 0.8510
Epoch 11/20
350/350 [==============================] - 201s 574ms/step - loss: 0.3340 - accuracy: 0.8826 - val_loss: 0.3542 - val_accuracy: 0.8819
Epoch 12/20
350/350 [==============================] - 200s 571ms/step - loss: 0.3095 - accuracy: 0.8894 - val_loss: 0.3863 - val_accuracy: 0.8671
Epoch 13/20
350/350 [==============================] - 198s 567ms/step - loss: 0.2974 - accuracy: 0.8943 - val_loss: 0.3722 - val_accuracy: 0.8766
Epoch 14/20
350/350 [==============================] - 200s 570ms/step - loss: 0.3208 - accuracy: 0.8867 - val_loss: 0.4036 - val_accuracy: 0.8592
Epoch 15/20
350/350 [==============================] - 198s 567ms/step - loss: 0.2424 - accuracy: 0.9164 - val_loss: 0.5918 - val_accuracy: 0.8188
Epoch 16/20
350/350 [==============================] - 198s 566ms/step - loss: 0.2270 - accuracy: 0.9204 - val_loss: 0.2978 - val_accuracy: 0.9078
Epoch 17/20
350/350 [==============================] - 199s 569ms/step - loss: 0.1959 - accuracy: 0.9296 - val_loss: 0.2525 - val_accuracy: 0.9201
Epoch 18/20
350/350 [==============================] - 199s 569ms/step - loss: 0.2022 - accuracy: 0.9299 - val_loss: 0.3665 - val_accuracy: 0.8744
Epoch 19/20
350/350 [==============================] - 199s 569ms/step - loss: 0.2095 - accuracy: 0.9295 - val_loss: 0.3381 - val_accuracy: 0.8854
Epoch 20/20
350/350 [==============================] - 201s 573ms/step - loss: 0.2014 - accuracy: 0.9314 - val_loss: 0.2892 - val_accuracy: 0.9040
```

Inference on few samples images:

The predict function takes a model and an image as input and returns the model's predicted class for the image and the confidence of the prediction. The function first converts the image to a NumPy array using the img_to_array function from the "tf.keras.preprocessing.image" module. It then adds an extra batch dimension to the array using tf.expand_dims. This is necessary because the model expects a batch of images as input and not a single image.

The model's prediction is obtained using the predict method, which takes the image array as input and returns a NumPy array of predictions, one for each class. The predicted class is obtained by finding the index of the maximum element in the predictions array using "np.argmax." The confidence of the prediction is calculated as the maximum element of the predictions array, multiplied by 100 and rounded to 2 decimal places.

In code I defines a for loop that iterates over the test generator and plots the first nine images shown below in the test set, along with their actual and predicted classes and confidence. The break statement is used to exit the loop after the first batch of images is processed, so that only the first nine images are plotted. This can be useful for quickly visualizing the model's performance on a small subset of the test set.


Actual: Tomato_Bacterial_spot,
Predicted: Tomato__Tomato_YellowLeaf__Curl_Virus.
Confidence: 98.51%
Actual: Tomato_Septoria_leaf_spot,
Predicted: Tomato_Septoria_leaf_spot.
Confidence: 99.69%
Actual: Tomato_Spider_mites_Two_spotted_spider_mite,
Predicted: Tomato_Spider_mites_Two_spotted_spider_mite.
Confidence: 100.0%
Actual: Tomato__Target_Spot,
Predicted: Tomato__Target_Spot.
Confidence: 98.85%
Actual: Tomato__Target_Spot,
Predicted: Tomato__Target_Spot.
Confidence: 93.86%
Actual: Tomato_Septoria_leaf_spot,
Predicted: Tomato_Septoria_leaf_spot.
Confidence: 55.14%
Actual: Tomato_Early_blight,
Predicted: Tomato_Spider_mites_Two_spotted_spider_mite.
Confidence: 82.76%
Actual: Tomato_Bacterial_spot,
Predicted: Tomato_Bacterial_spot.
Confidence: 99.99%
Actual: Tomato_Late_blight,
Predicted: Tomato_Late_blight.
Confidence: 89.85%

COMPLETE CODE:

```
!unzip drive/My\ Drive/PlantVillageTomatoSplited.zip


// IMPORT ALL THE DEPENDENCIES

import tensorflow as tf
from tensorflow.keras import models, layers
import matplotlib.pyplot as plt


//IMPORT DATA INTO TENSORFLOW DATASET OBJECT



IMAGE_SIZE = 256
CHANNELS = 3


from tensorflow.keras.preprocessing.image import ImageDataGenerator

train_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=10,
        horizontal_flip=True
)
train_generator = train_datagen.flow_from_directory(
        '/content/PlantVillageTomatoSplited/train',
        target_size=(IMAGE_SIZE,IMAGE_SIZE),
        batch_size=32,
        class_mode="sparse",
#          save_to_dir="C:\\Code\\potato-disease-
classification\\training\\AugmentedImages"
)


train_generator.class_indices


class_names = list(train_generator.class_indices.keys())
class_names


count=0
for image_batch, label_batch in train_generator:
    print(image_batch[0])
    break
```

```
validation_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=10,
        horizontal_flip=True)
validation_generator = validation_datagen.flow_from_directory(
        '/content/PlantVillageTomatoSplited/val',
        target_size=(IMAGE_SIZE,IMAGE_SIZE),
        batch_size=32,
        class_mode="sparse"
)


test_datagen = ImageDataGenerator(
        rescale=1./255,
        rotation_range=10,
        horizontal_flip=True)

test_generator = test_datagen.flow_from_directory(
        '/content/PlantVillageTomatoSplited/test',
        target_size=(IMAGE_SIZE,IMAGE_SIZE),
        batch_size=32,
        class_mode="sparse"
)
for image_batch, label_batch in test_generator:
    print(image_batch[0])
    break
```

// BUILDING THE MODEL

```
input_shape = (IMAGE_SIZE, IMAGE_SIZE, CHANNELS)
n_classes = 10

model = models.Sequential([
    layers.InputLayer(input_shape=input_shape),
    layers.Conv2D(32, kernel_size = (5,5), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64,  kernel_size = (5,5), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64,  kernel_size = (9,9), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((1, 1)),
    layers.Conv2D(64, (7, 7), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(n_classes, activation='softmax'),
])
```

// COMPILING MODEL

```
model.compile(
    optimizer='adam',
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=False),
    metrics=['accuracy']
)


11203/32


3198/32


history = model.fit(
    train_generator,
    steps_per_epoch=350,
    batch_size=32,
    validation_data=validation_generator,
    validation_steps=99,
    verbose=1,
    epochs=20,
)


scores = model.evaluate(test_generator)


scores
```

// PLOTTING THE ACCURACY AND LOSS CURVES

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

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

```
EPOCHS = 20

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')

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')
plt.show()
```

// RUN PREDICTION ON SAMPLE IMAGE

```
import numpy as np


for image_batch, label_batch in test_generator:
    first_image = image_batch[0]
    first_label = int(label_batch[0])

    print("first image to predict")
    plt.imshow(first_image)
    print("actual label:",class_names[first_label])

    batch_prediction = model.predict(image_batch)
    print("predicted label:",class_names[np.argmax(batch_prediction[0])])

    break
```

// WRITE A FUNCTION OF INFERENCE

```
def predict(model, img):
    img_array = tf.keras.preprocessing.image.img_to_array(images[i])
    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
```

```
// NOW RUN INFERENCE ON FEW SAMPLE IMAGES


plt.figure(figsize=(15, 15))
for images, labels in test_generator:
    for i in range(9):
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(images[i])

        predicted_class, confidence = predict(model, images[i])
        actual_class = class_names[int(labels[i])]

        plt.title(f"Actual: {actual_class},\n Predicted: {predicted_class}.\n Confi
dence: {confidence}%")

        plt.axis("off")
    break


from keras.utils.vis_utils import plot_model

plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=True
)
```

// This will create a plot of the model architecture and save it to a file called model.png. The plot will show the output shapes of each layer and the names of each layer.

```
import visualkeras
visualkeras.layered_view(model,legend=True,scale_xy=1, scale_z=1,max_z=50 )
```

// 3D plot of the model architecture, with a legend showing the names of each layer. The x and y axes will be scaled by a factor of 1, and the z axis will be scaled by a factor of 1 and will have a maximum value of 50.