Model Building

```
from keras.models import Sequential

input_shape = (IMAGE_SIZE, IMAGE_SIZE, CHANNELS)
n_classes = 3


model= Sequential([
    resize_and_rescale,
    data_augmentation,
    #layers.Input(shape=input_shape),
    layers.Conv2D(32, (3,3), activation='relu'),
    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')
])
```

- The numbers 32 & 64 are on trial & error basis

💡 `layers.Input(shape=input_shape)` was giving error. so it was removed.

you can try using `layers.InputLayer(input_shape=input_shape)`

OR

`input_shape=input_shape` in the 1st `COV2` layer

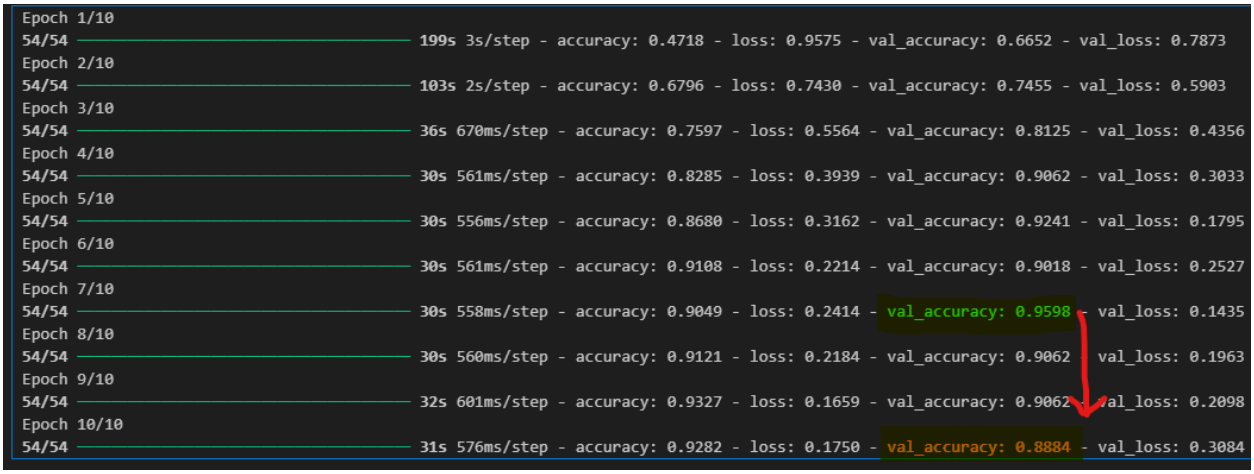
Compile

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

Fit:

```
history= model.fit(train_ds, epochs=10, verbose=1,validation_data=val_ds)
```

💡 👆 **Took 10 mIns to run on CPU.**



- To avoid this issue, use `ModelCheckpoint`

Accuracy on test dataset:

```
scores = model.evaluate(test_ds)
```

```
7/7 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 18s 205ms/step - accuracy: 0.8970 - loss: 0.2847
```

Plotting the Accuracy and Loss Curves

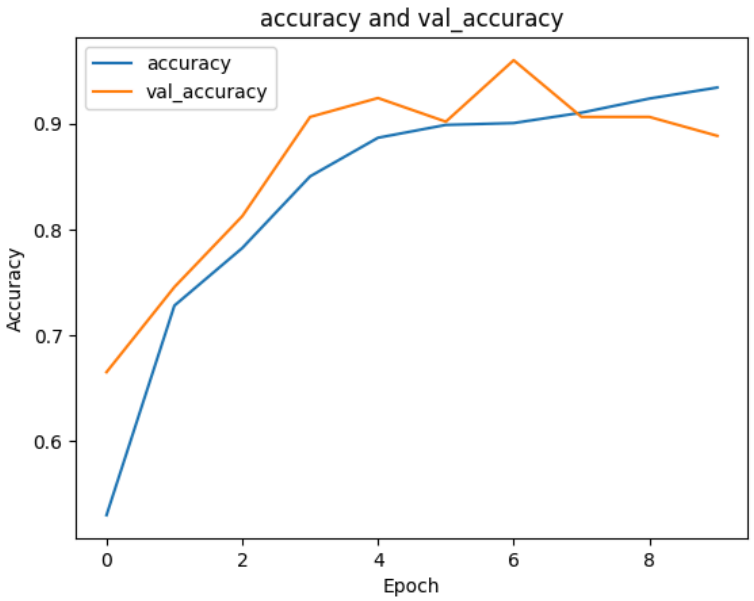
```
history.history.keys()
```

```
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```

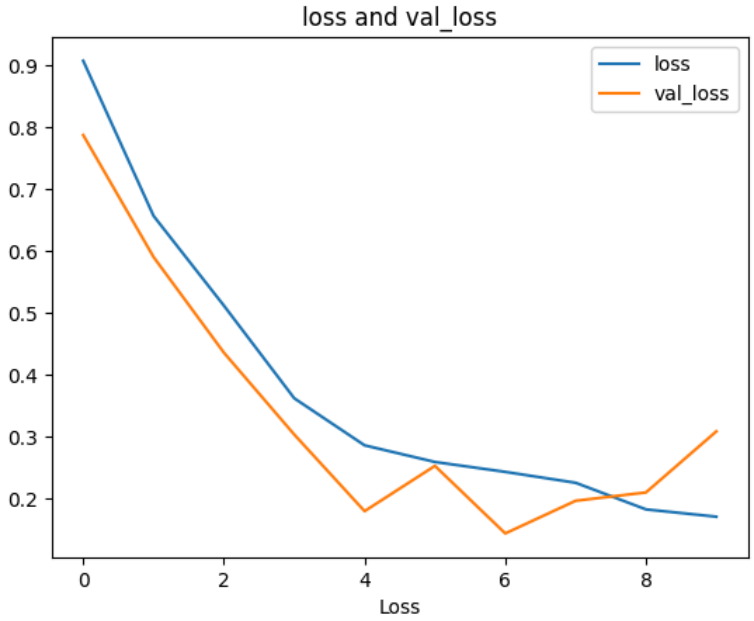
loss, accuracy, val loss etc are a python list containing values of loss, accuracy etc at the end of each epoch

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

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

```
plt.plot(acc, label='accuracy')
plt.plot(val_acc, label='val_accuracy')
plt.title('accuracy and val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
```

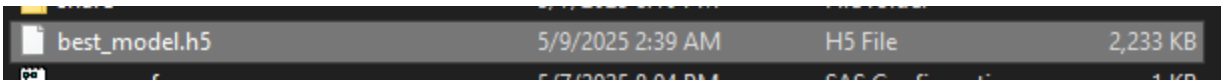


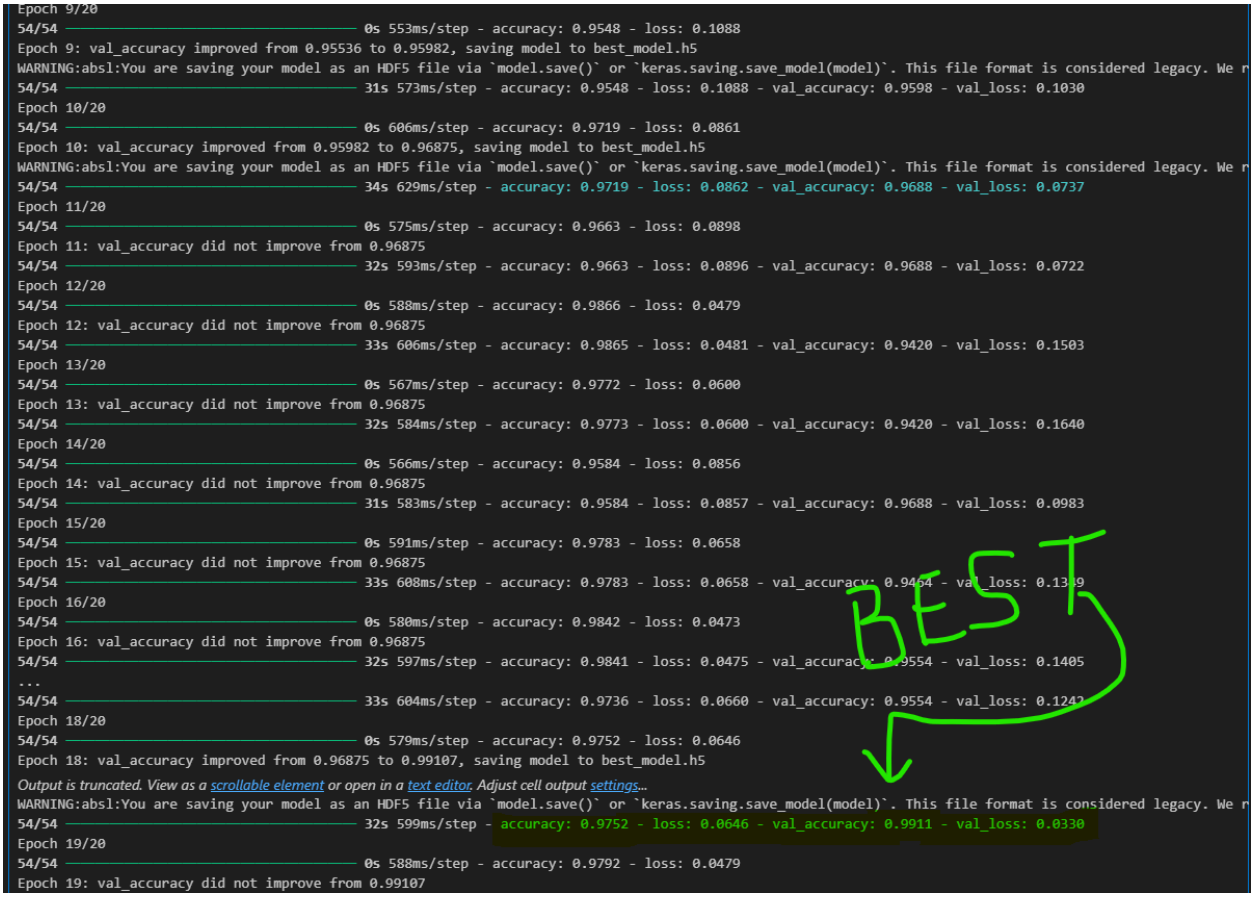

```
plt.plot(loss, label='loss')
plt.plot(val_loss, label='val_loss')
plt.title('loss and val_loss')
plt.xlabel('Loss')
plt.legend()
```



Use `ModelCheckpoint` to Save the Best Model

```
from tensorflow.keras.callbacks import ModelCheckpoint

# Save the model with highest validation accuracy
checkpoint = ModelCheckpoint(
    'best_model.h5',            # File to save weights
    monitor='val_accuracy',     # Metric to track
    mode='max',                 # Save max val_accuracy
    save_best_only=True,        # Only save if better than previous
    verbose=1                   # Print when saving
)

history = model.fit(
    train_ds,
    epochs=20,
    validation_data=val_ds,
    callbacks=[checkpoint]      # Add this callback
)
```





Line	Meaning
`'best_model.h5'`	The file where the best model is saved. This is your backup.
`monitor='val_accuracy'`	The model watches the **validation accuracy** at the end of every epoch.
`mode='max'`	Since higher accuracy is better, save the model when accuracy **increases**.
`save_best_only=True`	Only save the model if it's **better than all previous epochs**.
`verbose=1`	Prints a message like `Epoch 5: val_accuracy improved...` when it saves.

`callbacks=[checkpoint]` → Uses the callback above to **track and save the best model**

💡 Recommended format → **`'best_model.keras'`**

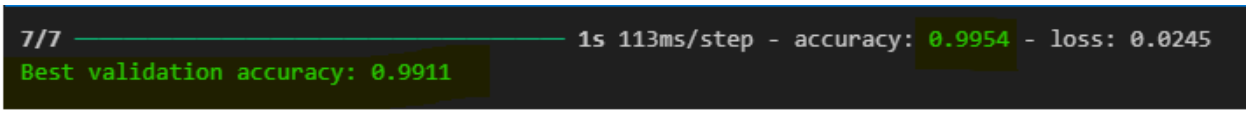
Use the Best Model:

```
model.load_weights("best_model.h5")
```

- Now you're using the version of the model that performed **best on validation data**, even if later epochs got worse (overfitting).
- If you didn't use checkpoints, the model object currently has weights from epoch 10 (val_accuracy: 88.84%). You can't restore epoch 7 weights without a checkpoint, but you can evaluate the final model or make small improvements.

Evaluate the Model

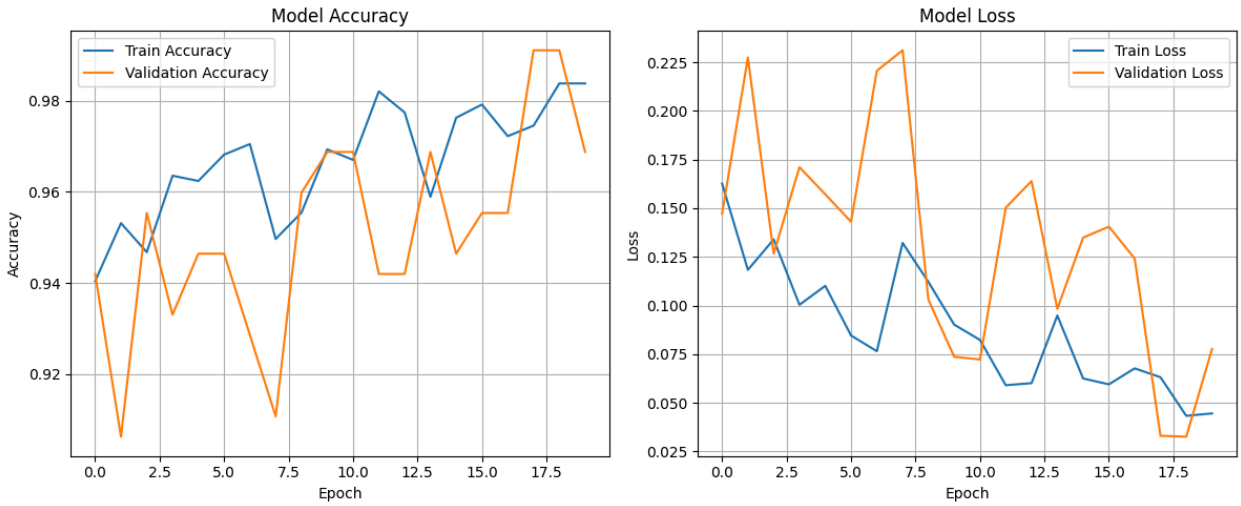
```
# Load the best model (if not already loaded)
model.load_weights('best_model.h5')

# Evaluate on validation data
val_loss, val_accuracy = model.evaluate(val_ds)
print(f"Best validation accuracy: {val_accuracy:.4f}")
```



Plotting the Accuracy and Loss Curves

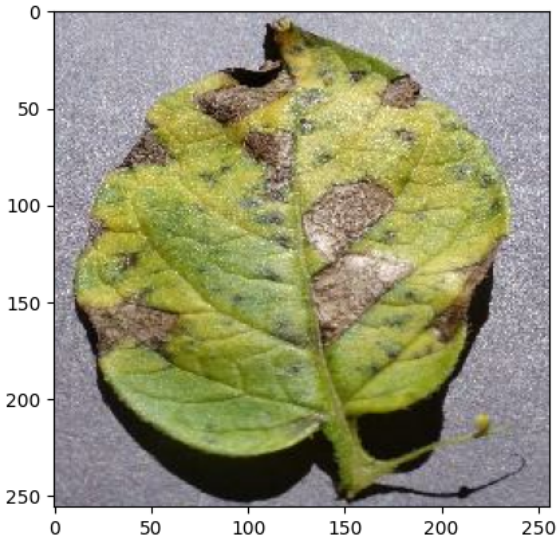
```
import matplotlib.pyplot as plt

# Plot Accuracy
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
```

```
plt.legend()
plt.grid(True)

# Plot Loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```



Run prediction on a sample image

```
for image_batch, labels_batch in test_ds.take(1):
    plt.imshow(image_batch[0].numpy().astype('uint8'))
```



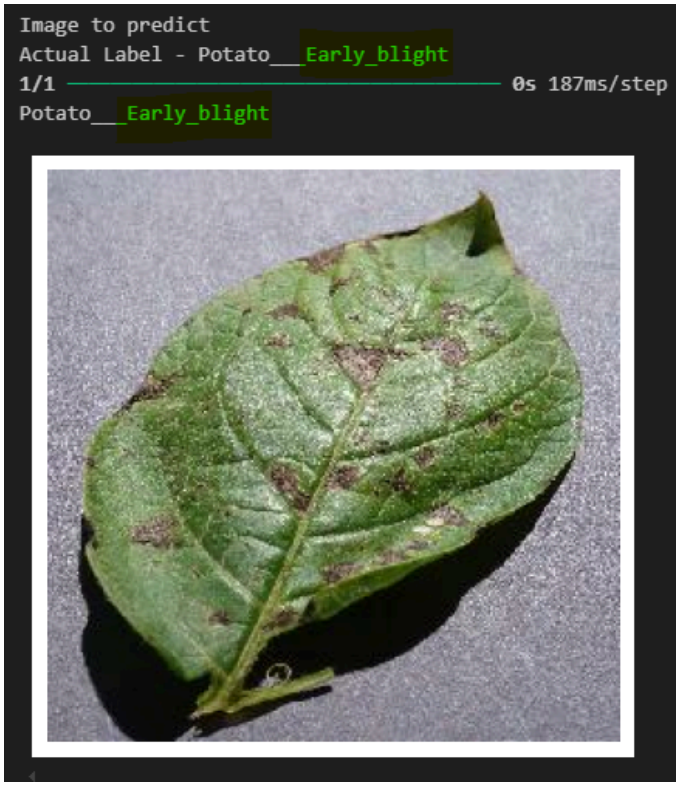
- This will display the image

Remember:

```
class_names

['Potato___Early_blight', 'Potato___Late_blight', 'Potato___healthy']
```

```
import numpy as np
for image_batch, labels_batch in test_ds.take(1):


    image= image_batch[0].numpy().astype('uint8')
    label= labels_batch[0].numpy()

    print("Image to predict")
    plt.imshow(image_batch[0].numpy().astype('uint8'))
    print(f"Actual Label - {class_names[label]}")

    #Predicted Label
```

```
batch_prediction= model.predict(image_batch)
print(class_names[np.argmax(batch_prediction[0])])

plt.axis("off")
```



- `batch_prediction= model.predict(image_batch)` → **Prediction for 32 images.**
  - Prediction for 1st image → `batch_prediction[0]`
  - This will give result like 👇

```
[9.9999988e-01 7.3204845e-08 2.5817703e-14]
```

  - `np.argmax` `(batch_prediction[0])` will give the number
  - `class_names` `[` `np.argmax` `(batch_prediction[0])]` will give the label

Write a function for inference

```
def predict(model, img, class_names=['Early Blight', 'Late Blight', 'Healthy']):
    img_array = tf.keras.preprocessing.image.img_to_array(images[i])
    img_array = tf.expand_dims(img_array, axis=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
```

`img_array = tf.keras.preprocessing.image.img_to_array(images[i]):`

- Converts `images[i]` (a tensor or PIL image) to a numpy array with shape `(height, width, channels)` (e.g., `(256, 256, 3)`).

📦 When do you use `img_to_array()`?

You use it when you're working with:

- A **single image** (e.g. during prediction)
- A **PIL image or OpenCV image** that isn't already a NumPy array
- A **custom image loading pipeline**, outside of `image_dataset_from_directory`

`img_array = tf.expand_dims(img_array, 0):`

- Adds a batch dimension, changing the shape from `(256, 256, 3)` → `(1, 256, 256, 3)`
- Required because the model expects batched inputs (e.g., `(batch_size, 256, 256, 3)`)

What is `images[i]`?

This code assumes `images` is a **list or array of images**, and `i` is an index.

`predictions = model.predict(img_array):`

- Runs the image through the model, returning a tensor of probabilities (shape: `(1, 3)` for 3 classes).

- Example: predictions[0] = `[0.05, 0.02, 0.93]` (probabilities for **Early Blight, Late Blight, Healthy**).

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

- Takes the highest probability `(np.max([0.05, 0.02, 0.93]) = 0.93)`.
- Converts to percentage `(0.93 * 100 = 93.0)` and rounds to 2 decimals (`93.0`).

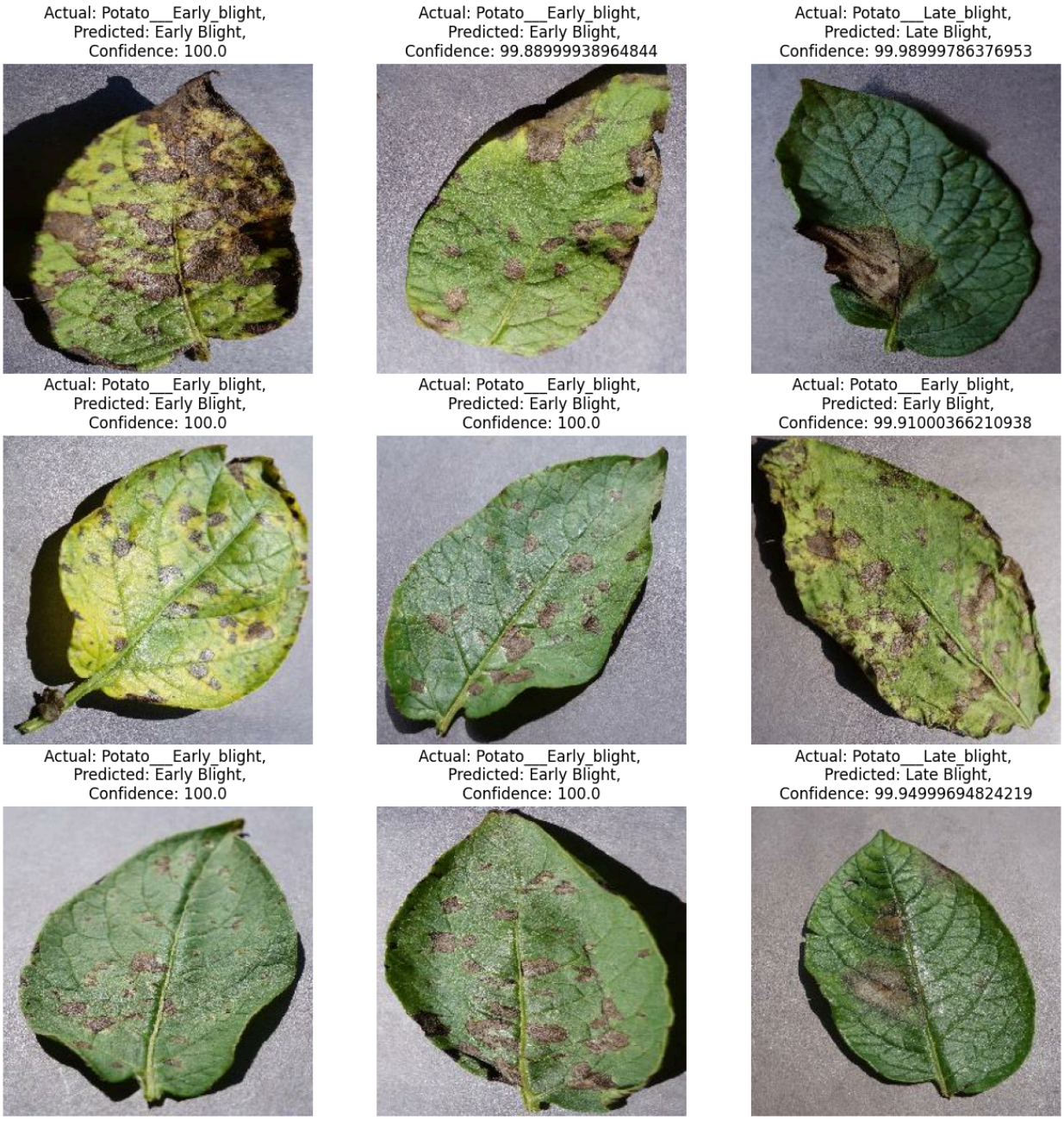
Now run inference on few sample images:

Show Image:

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

for images, labels in test_ds.take(1):
    for i in range(9):
        ax= plt.subplot(3,3,i+1)
        plt.imshow(images[i]/255)
        plt.axis("off")
```

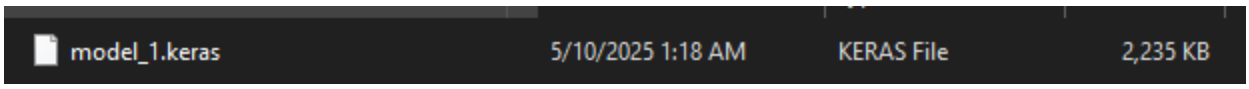
Predict:

```
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]/255)
        plt.axis("off")
        predicted_class, confidence = predict(model, images[i].numpy())
        actual_class= class_names[labels[i]]

        plt.title(f"Actual: {actual_class}, \n Predicted: {predicted_class},\nConfide
nce: {confidence}")
```



Save the model:

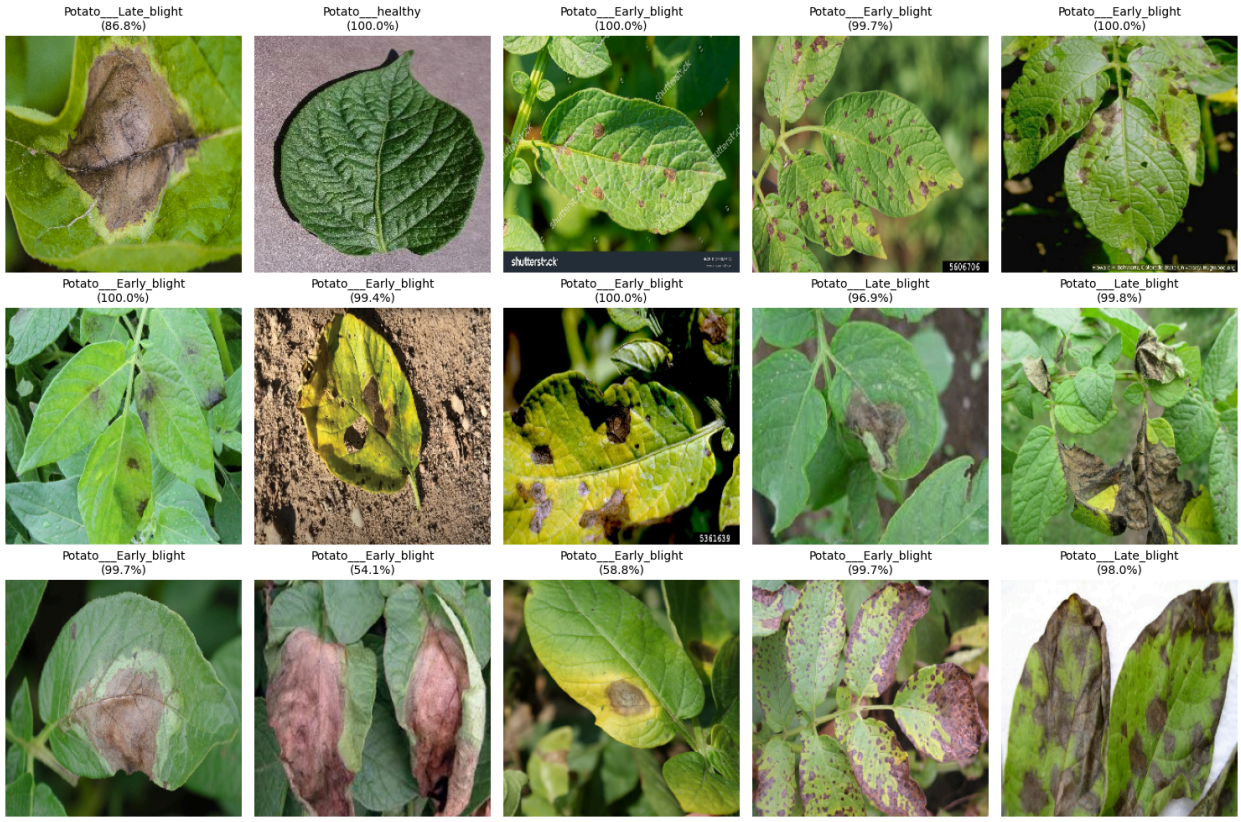
```
model.save(r'CNN Project\Models\model_1.keras')
```



Predict Unseen Images from Google:

```
import numpy as np
import matplotlib.pyplot as plt



def predict_image(model, image, class_names):

    image = tf.expand_dims(image, axis=0)  # Add batch dimension

    # Predict
    predictions = model.predict(image)
    predicted_class = class_names[np.argmax(predictions)]
    confidence = 100 * np.max(predictions)
    return predicted_class, confidence

plt.figure(figsize=(15, 10))

# Process each image individually
for i, image in enumerate(dataset_unseen.unbatch().take(15)):  # Explicitly tak
e 15 images
    # Predict
    predicted_class, confidence = predict_image(model, image, class_names)

    # Display
    ax = plt.subplot(3, 5, i+1)
    plt.imshow(image.numpy().astype('uint8'))  # Display original pixel values
    plt.title(f"{predicted_class}\n({confidence:.1f}%)", fontsize=10)
    plt.axis('off')
```

```
plt.tight_layout()
plt.show()
```



✅ .unbatch()

- This **converts batched data into individual elements**.
- If your dataset has 5 batches of 32 images (160 total), unbatching gives you 160 single images.

Why?

- By default, `image_dataset_from_directory` loads images in batches (e.g., `(batch_size, height, width, channels)`).
- `unbatch()` splits these batches into individual samples for easier processing.

```
# Before unbatch (batched)
for batch in dataset_unseen.take(1):
    print(batch[0].shape)  # Output: (32, 256, 256, 3) (batch of 32 images)

# After unbatch (single images)
for image in dataset_unseen.unbatch().take(1):
    print(image[0].shape)  # Output: (256, 256, 3) (single image)
```

✅ `enumerate(...)`

- Adds a **counter** `i` along with each image.
- So on the first iteration, `i = 0`, second `i = 1`, and so on.

```
for i, image in enumerate(dataset_unseen.unbatch().take(3)):
    print(f"Index: {i}, Image Shape: {image[0].shape}")

# Output:
# Index: 0, Image Shape: (256, 256, 3)
# Index: 1, Image Shape: (256, 256, 3)
# Index: 2, Image Shape: (256, 256, 3)
```

Accuracy

```
# Evaluate on validation dataset
val_loss, val_accuracy = model.evaluate(val_ds, verbose=1)
print(f"Validation Accuracy: {val_accuracy * 100:.2f}%")

# Evaluate on test dataset
test_loss, test_accuracy = model.evaluate(test_ds, verbose=1)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

```
7/7 ──────────────────────────────── 10s 1s/step - accuracy: 0.9921 - loss: 0.0235
Validation Accuracy: 99.11%
7/7 ──────────────────────────────── 9s 1s/step - accuracy: 0.9952 - loss: 0.0161
Test Accuracy: 99.55%
```

💡 **Normalization & Scaling are handled in initial layers of the model. Therefore, we don't need these steps again.**

💡 **Unseen images are treated the same as training images. Therefore, they don't need Normalization & Scaling.**

Classification Report

```
from sklearn.metrics import classification_report
import numpy as np

# Get true labels and predictions
y_true = []
y_pred = []

for images, labels in test_ds.unbatch():  # Iterate through all test data
    y_true.append(labels.numpy())
    pred = model.predict(tf.expand_dims(images, axis=0), verbose=0)
    y_pred.append(np.argmax(pred))

# Generate report
print(classification_report(
    y_true,
    y_pred,
```

```
    target_names=class_names  # ['Early Blight', 'Late Blight', 'Healthy']
))
```

```
                    precision    recall  f1-score   support

Potato___Early_blight       0.99      1.00      1.00       110
 Potato___Late_blight       1.00      0.99      1.00       101
     Potato___healthy       1.00      1.00      1.00        13

             accuracy                           1.00       224
            macro avg       1.00      1.00      1.00       224
         weighted avg       1.00      1.00      1.00       224
```