Trash Classification

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Import all the Dependencies

```
import tensorflow as tf
from tensorflow.keras import models, layers
import matplotlib.pyplot as plt
from IPython.display import HTML
```

Set all the Constants

```
BATCH_SIZE = 32
IMAGE_SIZE = 256
CHANNELS=3
EPOCHS=50
```

Import data into tensorflow dataset object

```
import os
import tensorflow as tf

# Step 1: Unzip the file from Google Drive to a local directory
zip_path = '/content/drive/MyDrive/DL_Project/Trash_Dataset'


dataset = tf.keras.preprocessing.image_dataset_from_directory(
    zip_path,
    seed=123,
    shuffle=True,
    image_size=(IMAGE_SIZE, IMAGE_SIZE),
    batch_size=BATCH_SIZE
)
```

```
Found 2132 files belonging to 4 classes.
```

```
class_names = dataset.class_names
class_names
```

```
['e-waste', 'food', 'paper', 'plastic']
```

```
for image_batch, labels_batch in dataset.take(1):
    print(image_batch.shape)
    print(labels_batch.numpy())
```

```
(32, 256, 256, 3)
[0 3 2 0 2 3 0 1 3 2 3 3 3 2 0 3 3 2 0 0 1 0 2 0 2 2 3 0 3 0 2 1]
```

As you can see above, each element in the dataset is a tuple. First element is a batch of 32 elements of images. Second element is a batch of 32 elements of class labels

Visualize some of the images from our dataset

```
plt.figure(figsize=(10, 10))
for image_batch, labels_batch in 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[labels_batch[i]])
        plt.axis("off")
```

Function to Split Dataset

Dataset should be bifurcated into 3 subsets, namely:

1. Training: Dataset to be used while training
2. Validation: Dataset to be tested against while training
3. Test: Dataset to be tested against after we trained a model

```
len(dataset)
```

```
67
```

```
train_size = 0.8
len(dataset)*train_size
```

```
53.6
```

```
train_ds = dataset.take(53)
len(train_ds)
```

```
53
```

```
test_ds = dataset.skip(53)
len(test_ds)
```

```
14
```

```
val_size=0.1
len(dataset)*val_size
```

```
6.7
```

```
val_ds = test_ds.take(6)
len(val_ds)
```

```
6
```

```
test_ds = test_ds.skip(6)
len(test_ds)
```

```
8
```

```
def get_dataset_partitions_tf(ds, train_split=0.8, val_split=0.1, test_split=0.1, shuffle=True, shuffle_size=10000):
    assert (train_split + test_split + val_split) == 1

    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
```

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

```
len(train_ds)
```

```
53
```

```
len(val_ds)
```

```
6
```

```
len(test_ds)
```

```
8
```

Cache, Shuffle, and Prefetch the Dataset

```
train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
val_ds = val_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
test_ds = test_ds.cache().shuffle(1000).prefetch(buffer_size=tf.data.AUTOTUNE)
```

Building the Model

Creating a Layer for Resizing and Normalization

Before we feed our images to network, we should be resizing it to the desired size. Moreover, to improve model performance, we should normalize the image pixel value (keeping them in range 0 and 1 by dividing by 256). This should happen while training as well as inference. Hence we can add that as a layer in our Sequential Model.

You might be thinking why do we need to resize (256,256) image to again (256,256). You are right we don't need to but this will be useful when we are done with the training and start using the model for predictions. At that time somone can supply an image that is not (256,256) and this layer will resize it

```
resize_and_rescale = tf.keras.Sequential([
  layers.Resizing(IMAGE_SIZE, IMAGE_SIZE),
  layers.Rescaling(1./255),
])
```

Data Augmentation

Data Augmentation is needed when we have less data, this boosts the accuracy of our model by augmenting the data.

```
data_augmentation = tf.keras.Sequential([
  layers.RandomFlip("horizontal_and_vertical"),
  layers.RandomRotation(0.2),
  layers.RandomZoom(0.2),
  layers.RandomContrast(0.2),
  layers.RandomBrightness(0.2),
])
```

Applying Data Augmentation to Train Dataset

```
train_ds = train_ds.map(
    lambda x, y: (data_augmentation(x, training=True), y)
).prefetch(buffer_size=tf.data.AUTOTUNE)
```

Model Architecture

We use a CNN coupled with a Softmax activation in the output layer. We also add the initial layers for resizing, normalization and Data Augmentation.

```
input_shape = (BATCH_SIZE, IMAGE_SIZE, IMAGE_SIZE, CHANNELS)
n_classes = len(class_names)

model = models.Sequential([
    resize_and_rescale,
    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(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)
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/convolutional/base_conv.py:113: UserWarning: Do not pass an `input_shap
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
model.summary()
```

Model: "sequential_2"

Layer (type)	Output Shape	Param #
sequential (Sequential)	(32, 256, 256, 3)	0
conv2d (Conv2D)	(32, 254, 254, 32)	896
max_pooling2d (MaxPooling2D)	(32, 127, 127, 32)	0
conv2d_1 (Conv2D)	(32, 125, 125, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(32, 62, 62, 64)	0
conv2d_2 (Conv2D)	(32, 60, 60, 64)	36,928
max_pooling2d_2 (MaxPooling2D)	(32, 30, 30, 64)	0
conv2d_3 (Conv2D)	(32, 28, 28, 64)	36,928
max_pooling2d_3 (MaxPooling2D)	(32, 14, 14, 64)	0
conv2d_4 (Conv2D)	(32, 12, 12, 64)	36,928
max_pooling2d_4 (MaxPooling2D)	(32, 6, 6, 64)	0
conv2d_5 (Conv2D)	(32, 4, 4, 64)	36,928
max_pooling2d_5 (MaxPooling2D)	(32, 2, 2, 64)	0
flatten (Flatten)	(32, 256)	0
dense (Dense)	(32, 64)	16,448
dense_1 (Dense)	(32, 4)	260

Total params: 183,812 (718.02 KB)
Trainable params: 183,812 (718.02 KB)

Compiling the Model

We use `adam` Optimizer, `SparseCategoricalCrossentropy` for losses, `accuracy` as a metric

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

```
history = model.fit(
    train_ds,
    batch_size=BATCH_SIZE,
    validation_data=val_ds,
    verbose=1,
    epochs=200,
)
```

Epoch 183/200

```
Epoch 183/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 29s 552ms/step - accuracy: 0.9032 - loss: 0.2622 - val_accuracy: 0.9271 - val_loss: 0.2431
Epoch 184/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 31s 580ms/step - accuracy: 0.9309 - loss: 0.1776 - val_accuracy: 0.9010 - val_loss: 0.2903
Epoch 185/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 29s 553ms/step - accuracy: 0.9251 - loss: 0.1856 - val_accuracy: 0.9062 - val_loss: 0.3719
Epoch 186/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 30s 556ms/step - accuracy: 0.9384 - loss: 0.1813 - val_accuracy: 0.8906 - val_loss: 0.3072
Epoch 187/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 30s 564ms/step - accuracy: 0.9351 - loss: 0.1748 - val_accuracy: 0.9271 - val_loss: 0.2912
Epoch 188/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 30s 557ms/step - accuracy: 0.9345 - loss: 0.1701 - val_accuracy: 0.9115 - val_loss: 0.3341
Epoch 189/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 29s 553ms/step - accuracy: 0.9314 - loss: 0.1753 - val_accuracy: 0.8646 - val_loss: 0.5369
Epoch 190/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 30s 556ms/step - accuracy: 0.9248 - loss: 0.2070 - val_accuracy: 0.9167 - val_loss: 0.3125
Epoch 191/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 30s 570ms/step - accuracy: 0.9223 - loss: 0.1900 - val_accuracy: 0.8698 - val_loss: 0.4357
Epoch 192/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 29s 551ms/step - accuracy: 0.8992 - loss: 0.2486 - val_accuracy: 0.9427 - val_loss: 0.2654
Epoch 193/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 30s 556ms/step - accuracy: 0.9513 - loss: 0.1396 - val_accuracy: 0.9115 - val_loss: 0.3138
Epoch 194/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 30s 560ms/step - accuracy: 0.9588 - loss: 0.1268 - val_accuracy: 0.9062 - val_loss: 0.3192
Epoch 195/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 30s 567ms/step - accuracy: 0.9410 - loss: 0.1594 - val_accuracy: 0.8958 - val_loss: 0.3476
Epoch 196/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 29s 550ms/step - accuracy: 0.9297 - loss: 0.1910 - val_accuracy: 0.9219 - val_loss: 0.2810
Epoch 197/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 30s 557ms/step - accuracy: 0.9281 - loss: 0.1857 - val_accuracy: 0.8542 - val_loss: 0.5960
Epoch 198/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 30s 568ms/step - accuracy: 0.9113 - loss: 0.2330 - val_accuracy: 0.9219 - val_loss: 0.2918
Epoch 199/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 29s 553ms/step - accuracy: 0.9507 - loss: 0.1308 - val_accuracy: 0.9219 - val_loss: 0.2841
Epoch 200/200
53/53 ━━━━━━━━━━━━━━━━━━━━ 29s 554ms/step - accuracy: 0.9562 - loss: 0.1278 - val_accuracy: 0.8854 - val_loss: 0.3657
```

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

```
8/8 ━━━━━━━━━━━━━━━━━━━━ 10s 21ms/step - accuracy: 0.8972 - loss: 0.2460
```

You can see above that we get 100.00% accuracy for our test dataset. This is considered to be a pretty good accuracy

```
scores
```

```
[0.27661430835723877, 0.9140625]
```

Scores is just a list containing loss and accuracy value

Plotting the Accuracy and Loss Curves

```
history
```

```
<keras.src.callbacks.history.History at 0x78427829e150>
```

```
history.params
```

```
{'verbose': 1, 'epochs': 200, 'steps': 53}
```

```
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

```
type(history.history['loss'])
```

```
list
```

```
len(history.history['loss'])
```

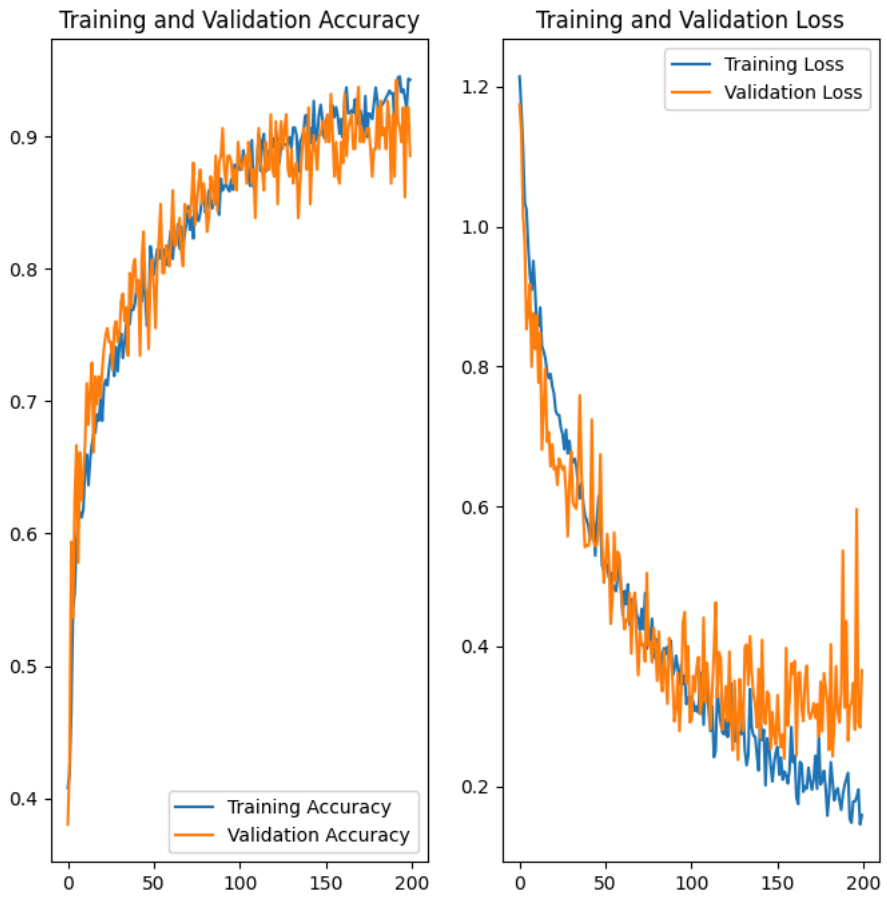
```
200
```

```
history.history['loss'][:5] # show loss for first 5 epochs
```

```
[1.2143535614013672,
 1.1679009199142456,
 1.1125059127807617,
 1.034356951713562,
 1.0241425037384033]
```

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

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

```
plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(range(len(acc)), acc, label='Training Accuracy')
plt.plot(range(len(val_acc)), val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(range(len(loss)), loss, label='Training Loss')
plt.plot(range(len(val_loss)), val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```

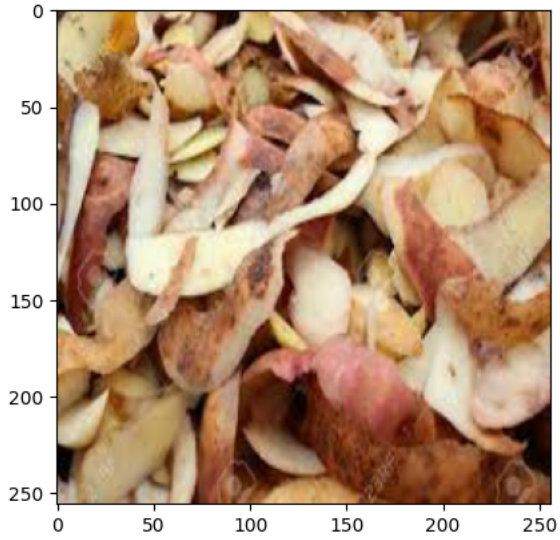


Run prediction on a sample image

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

    first_image = images_batch[0].numpy().astype('uint8')
    first_label = labels_batch[0].numpy()

    print("first image to predict")
```

```
    plt.imshow(first_image)
    print("actual label:",class_names[first_label])

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

```
first image to predict
actual label: food
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 797ms/step
predicted label: food
```



Write a function for inference

```
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
```

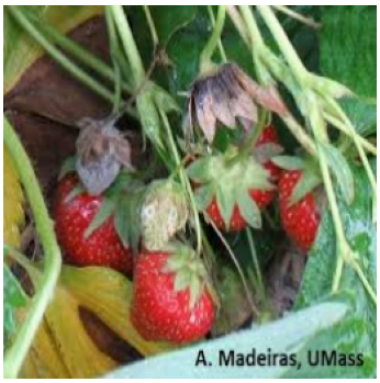
Now run inference on few sample images

```
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")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step
```

Actual: paper,
Predicted: paper.
Confidence: 96.4000015258789%

Actual: food,
Predicted: food.
Confidence: 79.13999938964844%



Actual: plastic,
Predicted: paper.
Confidence: 60.47999954223633%

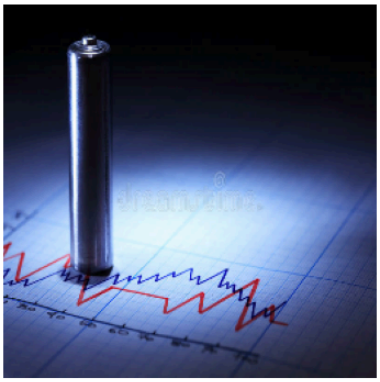
Actual: food,
Predicted: food.
Confidence: 98.19999694824219%

Actual: paper,
Predicted: paper.
Confidence: 74.23999786376953%

Actual: paper,
Predicted: paper.
Confidence: 99.98999786376953%

Actual: paper,
Predicted: e-waste.
Confidence: 92.05999755859375%

Actual: e-waste,
Predicted: plastic.
Confidence: 58.119998931884766%



Actual: e-waste,
Predicted: e-waste.
Confidence: 70.76000213623047%

Saving the Model

We append the model to the list of models as a new version

```
model.save("../trash.keras")
```

```
import os
print(os.getcwd())

print(os.path.exists("../models"))
!ls /models
```

```
/content
False
ls: cannot access '/models': No such file or directory
```

```
import tensorflow as tf
from tensorflow.keras import metrics

model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=[
        'accuracy',
        metrics.Precision(name='precision'),
        metrics.Recall(name='recall')
    ]
)
```

Calculates the confusion matrix based on the true and predicted labels.The code plots the confusion matrix using Seaborn's heatmap, displaying the counts of true positives, false positives, and false negatives for each class. The resulting heatmap provides a visual representation of the model's classification performance, with labeled axes for easy interpretation.

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# Step 1: Generate predictions on the test dataset
y_true = []
y_pred = []

for images, labels in test_ds:
    predictions = model.predict(images)
    predicted_classes = np.argmax(predictions, axis=1)

    y_true.extend(labels.numpy())
    y_pred.extend(predicted_classes)

# Step 2: Calculate the confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)

# Step 3: Plot the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=class_names, yticklabels=class_names)
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```