Implementation TARP Assignment 3:

Lokesh Reddy B (20BCE0943)

B N Mahesh Reddy (20BCE0959)

Jayanth T (20BCE0967)

In [ ]:
```
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers
```

```
C:\Users\Jayanth\AppData\Roaming\Python\Python310\site-packages\requests\__init__.py:
102: RequestsDependencyWarning: urllib3 (1.26.9) or chardet (5.0.0)/charset_normalize
r (2.0.12) doesn't match a supported version!
  warnings.warn("urllib3 ({}) or chardet ({})/charset_normalizer ({}) doesn't match a
supported "
```

Setting up our Data

Before we begin with creating and training our model, we will first set the size of the batches for our training, as well as the image height and width to set for our model

In [ ]:
```
batch_size = 100
img_height = 250
img_width = 250
```

The dataset that we are using has 3 different folders, and each of these have 2 folders within them having a folder for accident images and non accident images. Do look and scroll through them to verify and see the structure.
In order to get our:

1. train,
2. test
3. and validation split,

we will use keras's inbuilt *image_dataset_from_directory()* function which is able to generate a tf dataset containing the images as well as their corresponding classes from the folder that we pass into the parameter.

In [ ]:
```
training_ds = tf.keras.preprocessing.image_dataset_from_directory(
    'data/train',
    seed=101,
    image_size= (img_height, img_width),
    batch_size=batch_size
```

```
)

testing_ds = tf.keras.preprocessing.image_dataset_from_directory(
    'data/test',
    seed=101,
    image_size= (img_height, img_width),
    batch_size=batch_size)

validation_ds =  tf.keras.preprocessing.image_dataset_from_directory(
    'data/val',
    seed=101,
    image_size= (img_height, img_width),
    batch_size=batch_size)
```

```
Found 791 files belonging to 2 classes.
Found 89 files belonging to 2 classes.
Found 98 files belonging to 2 classes.
```

Notice the output reading the files as well as the classes it recognises!

Now, we'll set up a few performace parameters that will enhance runtime training of our model. I've learnt to use this from this excellent notebook here, so do check that out as well!

In [ ]:
```
class_names = training_ds.class_names

## Configuring dataset for performance
AUTOTUNE = tf.data.experimental.AUTOTUNE
training_ds = training_ds.cache().prefetch(buffer_size=AUTOTUNE)
testing_ds = testing_ds.cache().prefetch(buffer_size=AUTOTUNE)
```

Defining our Pre-Trained Model

The next step is defining and creating our model. In order to increase accuracy and speed up training process, we'll go ahead and use a pre trained model for this task. Why you may ask? This is because a pretrained convnet already has a very good idea of what features to look for in an image and can find them very effectively since it hs been trained on millions of images. So, if we can determine the presence of features all the rest of the model needs to do is determine which combination of features makes a specific image.
So all we've to do is:

1. Define the base pretrained layer
2. Add final few layers that are specific to our function and task to enhance ability in those categories
3. Train our model!
   Lets use Googles MobileNetV2 for this purpose...

In [ ]:
```
img_shape = (img_height, img_width, 3)

base_model = tf.keras.applications.MobileNetV2(input_shape=img_shape,
                                               include_top=False,
                                               weights='imagenet')

base_model.trainable = False
```

```
WARNING:tensorflow:`input_shape` is undefined or non-square, or `rows` is not in [96,
128, 160, 192, 224]. Weights for input shape (224, 224) will be loaded as the defaul
t.
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mo
bilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h5
9412608/9406464 [==============================] - 2s 0us/step
9420800/9406464 [==============================] - 2s 0us/step
```

Notice how we set trainable to false in order to make sure model won't make any changes to the weights of any layers that are already frozen during training.
We also exclude the top of the model since we will perform classification on our own.

Creating Final Model

We now go ahead and create our final model which consists of the base model, and 3 more layers for performing convolution. The 2d output of the convolution layer is flattened and fed to a dense output layer to perform the classification.

In [ ]:
```
model = tf.keras.Sequential([
    base_model,
    layers.Conv2D(32, 3, activation='relu'),
    layers.Conv2D(64, 3, activation='relu'),
    layers.Conv2D(128, 3, activation='relu'),
    layers.Flatten(),
    layers.Dense(len(class_names), activation= 'softmax')
])
```

In [ ]:
```
model.compile(optimizer='adam',loss='sparse_categorical_crossentropy', metrics=['accur
```

We'll let our model run for 50 epochs, which seems like a decent enough number. Increasing the epochs should result in an increase in accuracy uptil a certain point only though...

In [ ]:
```
history = model.fit(training_ds, validation_data = validation_ds, epochs = 50)
```

```
Epoch 1/50
8/8 [==============================] - 33s 3s/step - loss: 0.8273 - accuracy: 0.5714
- val_loss: 0.6246 - val_accuracy: 0.5918
Epoch 2/50
8/8 [==============================] - 22s 3s/step - loss: 0.5839 - accuracy: 0.6675
- val_loss: 0.5431 - val_accuracy: 0.7347
Epoch 3/50
8/8 [==============================] - 22s 3s/step - loss: 0.4208 - accuracy: 0.8116
- val_loss: 0.4812 - val_accuracy: 0.7857
Epoch 4/50
8/8 [==============================] - 28s 4s/step - loss: 0.2951 - accuracy: 0.8761
- val_loss: 0.2533 - val_accuracy: 0.9082
Epoch 5/50
8/8 [==============================] - 24s 3s/step - loss: 0.1693 - accuracy: 0.9444
- val_loss: 0.2854 - val_accuracy: 0.8469
Epoch 6/50
8/8 [==============================] - 21s 3s/step - loss: 0.1011 - accuracy: 0.9671
- val_loss: 0.4581 - val_accuracy: 0.8469
Epoch 7/50
8/8 [==============================] - 27s 4s/step - loss: 0.0907 - accuracy: 0.9633
- val_loss: 0.2451 - val_accuracy: 0.8878
Epoch 8/50
8/8 [==============================] - 26s 3s/step - loss: 0.1040 - accuracy: 0.9684
- val_loss: 0.1845 - val_accuracy: 0.9184
Epoch 9/50
8/8 [==============================] - 27s 3s/step - loss: 0.0849 - accuracy: 0.9772
- val_loss: 0.1706 - val_accuracy: 0.9388
Epoch 10/50
8/8 [==============================] - 29s 4s/step - loss: 0.0422 - accuracy: 0.9874
- val_loss: 0.1413 - val_accuracy: 0.9388
Epoch 11/50
8/8 [==============================] - 27s 3s/step - loss: 0.0432 - accuracy: 0.9861
- val_loss: 0.2488 - val_accuracy: 0.8878
Epoch 12/50
8/8 [==============================] - 24s 3s/step - loss: 0.0246 - accuracy: 0.9912
- val_loss: 0.2409 - val_accuracy: 0.8980
Epoch 13/50
8/8 [==============================] - 23s 3s/step - loss: 0.0203 - accuracy: 0.9924
- val_loss: 0.2348 - val_accuracy: 0.9082
Epoch 14/50
8/8 [==============================] - 27s 3s/step - loss: 0.0123 - accuracy: 0.9975
- val_loss: 0.2105 - val_accuracy: 0.9388
Epoch 15/50
8/8 [==============================] - 27s 3s/step - loss: 0.0130 - accuracy: 0.9949
- val_loss: 0.1961 - val_accuracy: 0.9184
Epoch 16/50
8/8 [==============================] - 27s 3s/step - loss: 0.0084 - accuracy: 0.9949
- val_loss: 0.2975 - val_accuracy: 0.8980
Epoch 17/50
8/8 [==============================] - 26s 3s/step - loss: 0.0103 - accuracy: 0.9949
- val_loss: 0.2438 - val_accuracy: 0.8980
Epoch 18/50
8/8 [==============================] - 27s 3s/step - loss: 0.0083 - accuracy: 0.9937
- val_loss: 0.3128 - val_accuracy: 0.8980
Epoch 19/50
8/8 [==============================] - 28s 4s/step - loss: 0.0099 - accuracy: 0.9949
- val_loss: 0.2352 - val_accuracy: 0.9082
Epoch 20/50
8/8 [==============================] - 27s 3s/step - loss: 0.0088 - accuracy: 0.9949
- val_loss: 0.3289 - val_accuracy: 0.8980
```

```
Epoch 21/50
8/8 [==============================] - 27s 3s/step - loss: 0.0099 - accuracy: 0.9937
- val_loss: 0.1969 - val_accuracy: 0.9082
Epoch 22/50
8/8 [==============================] - 26s 3s/step - loss: 0.0091 - accuracy: 0.9962
- val_loss: 0.3163 - val_accuracy: 0.8980
Epoch 23/50
8/8 [==============================] - 26s 3s/step - loss: 0.0095 - accuracy: 0.9937
- val_loss: 0.1930 - val_accuracy: 0.9184
Epoch 24/50
8/8 [==============================] - 27s 3s/step - loss: 0.0084 - accuracy: 0.9962
- val_loss: 0.3371 - val_accuracy: 0.9082
Epoch 25/50
8/8 [==============================] - 27s 3s/step - loss: 0.0097 - accuracy: 0.9949
- val_loss: 0.2175 - val_accuracy: 0.8980
Epoch 26/50
8/8 [==============================] - 26s 3s/step - loss: 0.0069 - accuracy: 0.9975
- val_loss: 0.2630 - val_accuracy: 0.8980
Epoch 27/50
8/8 [==============================] - 26s 3s/step - loss: 0.0088 - accuracy: 0.9937
- val_loss: 0.2473 - val_accuracy: 0.9082
Epoch 28/50
8/8 [==============================] - 27s 3s/step - loss: 0.0065 - accuracy: 0.9949
- val_loss: 0.2909 - val_accuracy: 0.9082
Epoch 29/50
8/8 [==============================] - 24s 3s/step - loss: 0.0078 - accuracy: 0.9949
- val_loss: 0.2532 - val_accuracy: 0.9082
Epoch 30/50
8/8 [==============================] - 26s 3s/step - loss: 0.0066 - accuracy: 0.9924
- val_loss: 0.2600 - val_accuracy: 0.9082
Epoch 31/50
8/8 [==============================] - 31s 4s/step - loss: 0.0071 - accuracy: 0.9949
- val_loss: 0.2580 - val_accuracy: 0.9082
Epoch 32/50
8/8 [==============================] - 29s 4s/step - loss: 0.0066 - accuracy: 0.9949
- val_loss: 0.2785 - val_accuracy: 0.9082
Epoch 33/50
8/8 [==============================] - 26s 3s/step - loss: 0.0070 - accuracy: 0.9949
- val_loss: 0.2832 - val_accuracy: 0.9082
Epoch 34/50
8/8 [==============================] - 29s 4s/step - loss: 0.0070 - accuracy: 0.9937
- val_loss: 0.2807 - val_accuracy: 0.9082
Epoch 35/50
8/8 [==============================] - 29s 4s/step - loss: 0.0069 - accuracy: 0.9937
- val_loss: 0.2777 - val_accuracy: 0.9082
Epoch 36/50
8/8 [==============================] - 26s 3s/step - loss: 0.0067 - accuracy: 0.9949
- val_loss: 0.2821 - val_accuracy: 0.9082
Epoch 37/50
8/8 [==============================] - 24s 3s/step - loss: 0.0067 - accuracy: 0.9949
- val_loss: 0.2887 - val_accuracy: 0.9082
Epoch 38/50
8/8 [==============================] - 28s 4s/step - loss: 0.0067 - accuracy: 0.9949
- val_loss: 0.2897 - val_accuracy: 0.9082
Epoch 39/50
8/8 [==============================] - 27s 3s/step - loss: 0.0066 - accuracy: 0.9949
- val_loss: 0.2955 - val_accuracy: 0.9082
Epoch 40/50
8/8 [==============================] - 25s 3s/step - loss: 0.0067 - accuracy: 0.9949
- val_loss: 0.3059 - val_accuracy: 0.8980
```

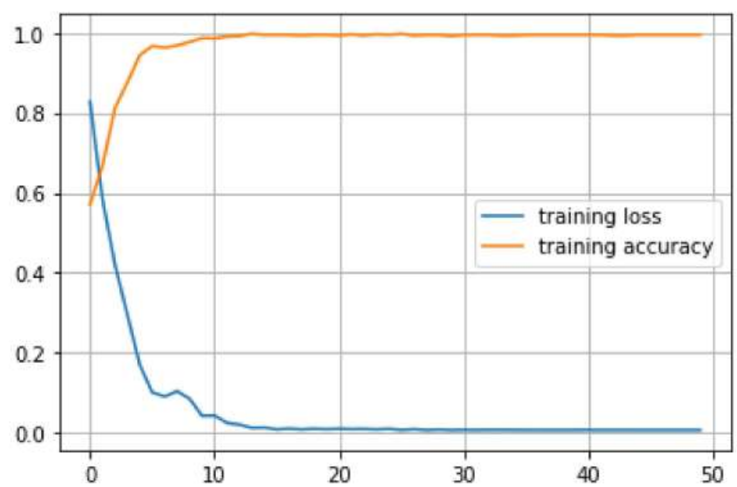
```
Epoch 41/50
8/8 [==============================] - 29s 4s/step - loss: 0.0070 - accuracy: 0.9949
- val_loss: 0.2872 - val_accuracy: 0.8980
Epoch 42/50
8/8 [==============================] - 26s 3s/step - loss: 0.0065 - accuracy: 0.9949
- val_loss: 0.2827 - val_accuracy: 0.8980
Epoch 43/50
8/8 [==============================] - 24s 3s/step - loss: 0.0067 - accuracy: 0.9937
- val_loss: 0.2813 - val_accuracy: 0.9082
Epoch 44/50
8/8 [==============================] - 23s 3s/step - loss: 0.0066 - accuracy: 0.9937
- val_loss: 0.3004 - val_accuracy: 0.8980
Epoch 45/50
8/8 [==============================] - 26s 3s/step - loss: 0.0065 - accuracy: 0.9949
- val_loss: 0.2987 - val_accuracy: 0.9082
Epoch 46/50
8/8 [==============================] - 23s 3s/step - loss: 0.0065 - accuracy: 0.9949
- val_loss: 0.2947 - val_accuracy: 0.9082
Epoch 47/50
8/8 [==============================] - 25s 3s/step - loss: 0.0065 - accuracy: 0.9949
- val_loss: 0.2979 - val_accuracy: 0.9082
Epoch 48/50
8/8 [==============================] - 26s 3s/step - loss: 0.0066 - accuracy: 0.9949
- val_loss: 0.2912 - val_accuracy: 0.8980
Epoch 49/50
8/8 [==============================] - 24s 3s/step - loss: 0.0065 - accuracy: 0.9949
- val_loss: 0.2897 - val_accuracy: 0.8980
Epoch 50/50
8/8 [==============================] - 24s 3s/step - loss: 0.0067 - accuracy: 0.9949
- val_loss: 0.2916 - val_accuracy: 0.9082
```

In [ ]:

```python
plt.plot(history.history['loss'], label = 'training loss')
plt.plot(history.history['accuracy'], label = 'training accuracy')
plt.grid(True)
plt.legend()
```

Out[ ]:

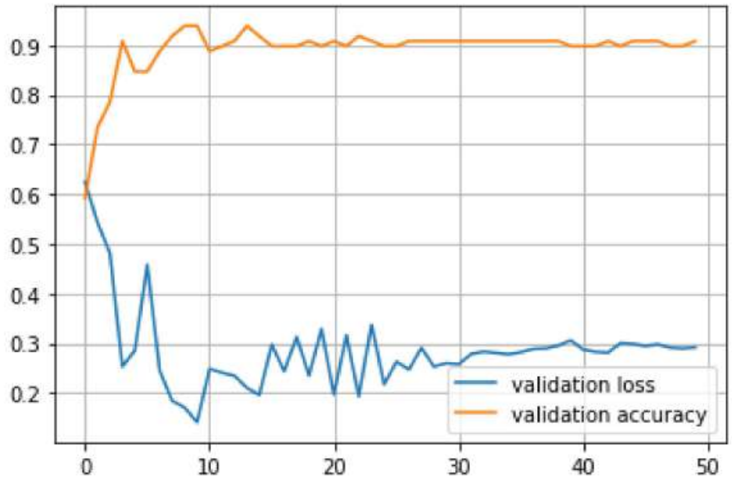
```
<matplotlib.legend.Legend at 0x1fd010eb520>
```



In [ ]:

```python
plt.plot(history.history['val_loss'], label = 'validation loss')
plt.plot(history.history['val_accuracy'], label = 'validation accuracy')
plt.grid(True)
plt.legend()
```

Out[ ]: <matplotlib.legend.Legend at 0x1fd01146020>



The function below looks a bit complicated, but is a simple helper function which shows the image, the predicted class and the actual class for each image in the test dataset. Run it and have a look at how accurate the model seems and where it seems to be struggling.

In [ ]:
```
AccuracyVector = []
plt.figure(figsize=(30, 30))
for images, labels in testing_ds.take(1):
    predictions = model.predict(images)
    predlabel = []
    prdlbl = []

    for mem in predictions:
        predlabel.append(class_names[np.argmax(mem)])
        prdlbl.append(np.argmax(mem))

    AccuracyVector = np.array(prdlbl) == labels
    for i in range(40):
        ax = plt.subplot(10, 4, i + 1)
        plt.imshow(images[i].numpy().astype("uint8"))
        plt.title('Pred: '+ predlabel[i]+' actl:'+class_names[labels[i]] )
        plt.axis('off')
        plt.grid(True)
```

We can go ahead and view the models layers through the plot_model function below provided by keras for an intuitive view.

In [ ]:
```
from keras.utils.vis_utils import plot_model
plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=True)
```

```
You must install pydot (`pip install pydot`) and install graphviz (see instructions a
t https://graphviz.gitlab.io/download/) for plot_model/model_to_dot to work.
```

And thats all! We've successfully creating a model with an accuracy of around 90%. Notice that this can be further improved by performing image manipulation, performing pooling and training our model for a longer epoch or even adding more layers.. However, for our use case, this model we created is perfectly fine.

In [ ]:
```
print(class_names)
```

```
['Accident', 'Non Accident']
```

Testing Model on Videos

In order to use our model on a video, which is our expected use case of a CCTV footage, we will have to use OpenCV in order get the individual frames.
Lets define a function which takes in each frame and converts it into a tensor and then predicts the output class.

In [ ]:
```
def predict_frame(img):
    img_array = tf.keras.utils.img_to_array(img)
    img_batch = np.expand_dims(img_array, axis=0)
    prediction=(model.predict(img_batch) > 0.5).astype("int32")
    if(prediction[0][0]==0):
        return("Accident Detected")
    else:
        return("No Accident")
```

The following code below makes use of OpenCV. Firstly, we read the video in and grab every 20th frame(in order to reduce total computation for this demonstration) and then we can resize the image and run our function on it.
We'll store the label and the image in a list which we can easily access.

In [ ]:
```
import cv2
image=[]
label=[]

c=1
cap= cv2.VideoCapture('data/video.mp4')
while True:
    grabbed, frame = cap.read()
    if c%30==0:
        print(c)
        resized_frame=tf.keras.preprocessing.image.smart_resize(frame, (img_height, in
        image.append(frame)
        label.append(predict_frame(resized_frame))
        if(len(image)==75):
            break
    c+=1

cap.release()
```

```
30
60
90
120
150
180
210
240
270
300
330
360
390
420
450
480
510
540
570
600
630
660
690
720
750
780
810
840
870
900
930
960
990
1020
1050
1080
1110
1140
1170
1200
1230
1260
1290
1320
1350
1380
1410
1440
1470
1500
1530
1560
1590
1620
1650
1680
1710
1740
1770
1800
```

```
1830
1860
1890
1920
1950
1980
2010
2040
2070
2100
2130
2160
2190
2220
2250
```

Lets see any random frame and see what the outcome is...

In [ ]:
```python
print(label[10])
print(plt.imshow(image[10]))
```

```
No Accident
AxesImage(54,36;334.8x217.44)
```

Looks about right! There seems to be an accident occuring in this frame. Our model generalizes well and can be used for practical applications.

Converting to TFLite Model

While we've made our model, it is true that Tensor Flow models are very large and bulky and not suitable for the small processing powers that a CCTV surveillance system will handle. For this purpose, we'll convert our Tf model into a TFLite model through the API's available by keras.

In [ ]:
```python
# Convert the model.
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

# Save the model.
with open('tf_lite_model.tflite', 'wb') as f:
    f.write(tflite_model)
```

```
WARNING:absl:Function `_wrapped_model` contains input name(s) mobilenetv2_1.00_224_in
put with unsupported characters which will be renamed to mobilenetv2_1_00_224_input i
n the SavedModel.
INFO:tensorflow:Assets written to: C:\Users\Jayanth\AppData\Local\Temp\tmpms_r2fmn\as
sets
INFO:tensorflow:Assets written to: C:\Users\Jayanth\AppData\Local\Temp\tmpms_r2fmn\as
sets
WARNING:absl:Buffer deduplication procedure will be skipped when flatbuffer library i
s not properly loaded
```

A TFLite model is referred to as an interpreter. We open it up and have a look at the input and output shape. It should be a single image of height and width 250 by 250 with 3 colour channels.
The output can be of 2 types only. Accident or Non Accident.

In [ ]:
```
interpreter = tf.lite.Interpreter(model_path = 'tf_lite_model.tflite')
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
print("Input Shape:", input_details[0]['shape'])
print("Input Type:", input_details[0]['dtype'])
print("Output Shape:", output_details[0]['shape'])
print("Output Type:", output_details[0]['dtype'])
```

```
Input Shape: [  1 250 250   3]
Input Type: <class 'numpy.float32'>
Output Shape: [1 2]
Output Type: <class 'numpy.float32'>
```

While the steps below aren't necessary, I'll still show you incase you have to perform a similair task for a different model where the input tensor might change or be different.

In [ ]:
```
interpreter.resize_tensor_input(input_details[0]['index'], (1, 250, 250,3))
interpreter.resize_tensor_input(output_details[0]['index'], (1, 2))
interpreter.allocate_tensors()
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()
print("Input Shape:", input_details[0]['shape'])
print("Input Type:", input_details[0]['dtype'])
print("Output Shape:", output_details[0]['shape'])
print("Output Type:", output_details[0]['dtype'])
```

```
Input Shape: [  1 250 250   3]
Input Type: <class 'numpy.float32'>
Output Shape: [1 2]
Output Type: <class 'numpy.float32'>
```

Trying Our TFLite Model Out

We'll try our TFLite model on a random image and see what our output is and if it works.

In [ ]:
```
from PIL import Image
im=Image.open("data/train/Non Accident/5_17.jpg").resize((250,250))
img_array = tf.keras.utils.img_to_array(im)
img_batch = np.expand_dims(img_array, axis=0)
```

The below lines are equivalent to performing a prediction in a TF model. *interpretor.get_tensor()* performs the prediction.

In [ ]:
```
interpreter.set_tensor(input_details[0]['index'], img_batch)
interpreter.invoke()
tflite_model_predictions = interpreter.get_tensor(output_details[0]['index'])
print("Prediction results:", tflite_model_predictions)
print(plt.imshow(im))
```

```
Prediction results: [[4.0345520e-04 9.9959654e-01]]
AxesImage(54,36;334.8x217.44)
```

It works. We've got a complete end to end system for accident detection now that should work very well indeed.