DEEP LEARNING – TRANSFER LEARNING PROJECT
DOGS VS CATS DATASET

ROLL NO: 2018103592 **DATE: 12 May 2021**
NAME: SHRUTHI M

PROBLEM STATEMENT

Usually, deep learning models take a long time (sometimes even days) to get trained. Also, in most cases, we cannot construct a proper model architecture to train for dataset. The most significant advantage is saving a lot of training time, better performance of CNN architectures, and not needing lot of data. Usually, a lot of data is needed to train a neural network from scratch but access to that data isn't always available — this is where transfer learning comes in handy. With transfer learning a solid machine learning model can be built with comparatively little training data because the model is already pre-trained. With transfer learning, we basically try to exploit what has been learned in one task to improve generalization in another. We transfer the weights that a network has learned at "task A" to a new "task B."

Some of the best transfer learning models for image data include:

- ✓ Oxford VGG Model (VGG16 or VGG19)
- ✓ Google Inception Model (Inception v3)
- ✓ Microsoft Residual Network Model (ResNet)

In this assignment, we train a simple and common dataset with VGG16 model (pre-trained network) with and without fine tuning and compare the results.

DATASET DETAILS

https://www.kaggle.com/c/dogs-vs-cats/data

The dataset contains 25000 images for training set with 12500 images for cats and 12500 images for dogs. The test dataset contains 12500 images. The images in training set are stored in the format “cat.23.jpg”. The labels are not given. Hence, in the pre-processing step, we need to categorize and label each of the dataset so that it becomes a supervised learning.

MODULES

EXPLORING AND VISUALISING DATASET

The dataset is large and training such a huge dataset takes lot of time even with VGG16 model. Since, pre-trained models do not require large amount of data, we exploit this to our advantage. Here, we randomly choose 1000 cat pictures and 1000 dog pictures for training set out of 25000 images. Similarly, for testing, we predict the model with 100 randomly selected images out of 12500 images.

SPLIT TRAINING AND VALIDATION SET

We store the filename and category as a pandas dataframe. Now, we split this with a percentage value of 0.1 (i.e. 90% for training and 10% for validation)

DATA AUGMENTATION

Data Augmentation increases the dataset size by applying various transformations on the original dataset. This not only increases the size but also helps in creating the data with different perspectives and prevents over fitting. These are some of the transformations that are performed.

featurewise_center: Set input mean to 0 over the dataset
samplewise_center: Set each sample mean to 0
featurewise_std_normalization: Divide inputs by std of the dataset
samplewise_std_normalization: Divide each input by its std
zca_whitening: Dimension reduction
rotation_range: Randomly rotate images in the range
zoom_range: Randomly zoom image
width_shift_range: Randomly shift images horizontally
height_shift_range: Randomly shift images vertically
horizontal_flip: Randomly flip images
vertical_flip: Randomly flip images
rescale = 1./255: Rescale between 0 and 1

This is for training dataset only.

CREATING GENERATORS

The ImageDataGenerator class is very useful in image classification. There are several ways to use this generator, depending on the method we use, here we have used flow_from_directory method that takes a path to the directory containing images sorted in sub directories and image augmentation parameters.

This is created for training, validation and testing images.

THE NEXT 4 MODULES WILL BE ANALYSED FOR BOTH VGG16 WITHOUT FINE TUNING AS WELL AS WITH FINE TUNING

BUILDING MODEL

The model is built with VGG16 network. We train the model with and without fine tuning. We compare both the models and interpret.

FITTING THE MODEL

The images are fit into the model and trained

PLOTTING GRAPHS

We plot training and validation errors and accuracy graph.

PREDICTION & EVALUATION

The model is evaluated with the testing set and confusion matrix is plotted. Along with this classification report is also generated.

CODING SNAPSHOTS

MODULE I - EXPLORING AND VISUALISING DATASET

Importing packages

+ Code + Text

Importing Packages

```
[1] import numpy as np
    import pandas as pd
    import seaborn as sns
    import matplotlib.pyplot as plt
    import os
    from glob import glob
    import cv2
    import random

    # Visualisation
    import matplotlib as mpl
    import matplotlib.pyplot as plt
    import matplotlib.pylab as pylab
    import seaborn as sns

    # Configure visualisations
    %matplotlib inline
```

```
[2] from keras.models import Sequential
    from keras import layers, optimizers
    from keras.layers import Conv2D, MaxPooling2D, Activation, Dropout, Flatten, Dense, GlobalMaxPooling2D
    from tensorflow.keras.callbacks import EarlyStopping
    from keras.preprocessing.image import ImageDataGenerator, img_to_array, load_img
    from keras.applications.vgg16 import VGG16

    import matplotlib.pyplot as plt
    from PIL import Image
    from glob import glob
    import tensorflow as tf
```

Since, training takes a lot of time with CPU, we use GPU

```
[5] print(tf.test.gpu_device_name())
    from tensorflow.python.client import device_lib
    print(device_lib.list_local_devices())
    !cat /proc/meminfo
```

```
device_type: "GPU"
memory_limit: 15703311680
locality {
  bus_id: 1
  links {
  }
}
incarnation: 15018534220034758095
physical_device_desc: "device: 0, name: Tesla P100-PCIE-16GB, pci bus id: 0000:00:04.0, compute capability: 6.0"
]
MemTotal:       13305328 kB
MemFree:         3912592 kB
MemAvailable:   11837640 kB
Buffers:          170716 kB
Cached:          7531568 kB
SwapCached:            0 kB
Active:          2746976 kB
Inactive:        6011568 kB
Active(anon):     852672 kB
Inactive(anon):     8632 kB
Active(file):    1894304 kB
Inactive(file):  6002936 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:             0 kB
SwapFree:              0 kB
Dirty:               920 kB
Writeback:             0 kB
AnonPages:       1056216 kB
Mapped:           678884 kB
Shmem:              9468 kB
```

Executing (2m 31s) Cell > next () > next() > get batches of transformed sa... > load img()

The training set contains 25000 images 12500 each of cats and dogs. Since, it takes a lot of time to train, we randomly select 1500 images of each category and store it in directory “train_samp”.

Randomly shuffle training dataset and make a dataframe storing filename and category

```
[8] fname = os.listdir('./train_samp')
    random.shuffle(fname)
    print(fname)
```

```
['dog.3072.jpg', 'dog.3087.jpg', 'cat.1092.jpg', 'cat.599.jpg', 'dog.12264.jpg', 'dog.3280.jpg', 'dog.12150.jpg', 'dog.3571.jpg', 'cat.509.jpg', 'cat.1378.jpg', 'dog.11897.jpg', 'cat.
```

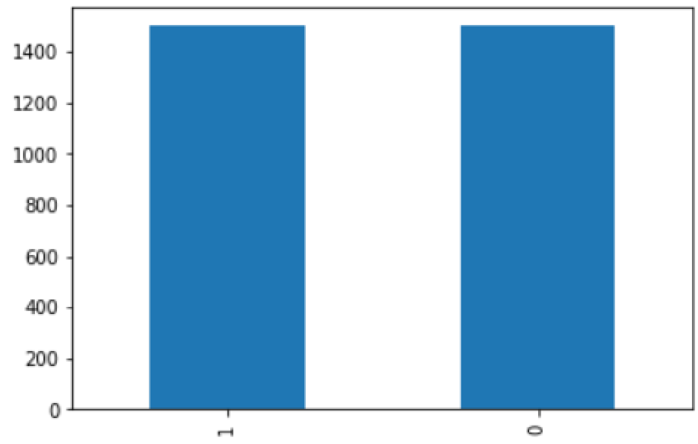
Since, the images are stored as jpeg files in directory and labels are not present, we need to label them. So, we create a **pandas dataframe** that stores the filename and the label. The filenames are stored as “cat.12.jpg” , “dog.256.jpg”. Hence, we split each of the filename based on the delimiter (“.”). So, if the first part is “cat”, it is labelled as 0 in the dataframe and “dogs” are labelled as 1.

```
label = []
for f in fname:
    category = f.split('.')
    if category[2] != 'jpg':
      print(category[2])
    if category[0] == "cat":
        label.append(0)
    else:
        label.append(1)
```

In the below image, we see there are 1501 images of each category.

```
[10] df = pd.DataFrame({ 'filename': fname, 'label': label })
     df['label'].value_counts().plot.bar()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fd86b5d4990>
```




```
[11] print(df.shape)
```

```
(3002, 2)
```

MODULE II - SPLIT TRAINING AND VALIDATION SET

Training and validation split is done based on the pandas dataframe. The filenames are stored separately in training and validation, which will then be generated later.

The split size is given as 0.2 (80% for training, 20% for validation)

Split train and validation set

```
[12] train_df, val_df = train_test_split(df, test_size=0.2)
     train_df = train_df.reset_index()
     val_df = val_df.reset_index()
```

```
[13] train_df = train_df.drop(['index'], axis = 1)
     val_df = val_df.drop(['index'], axis = 1)
     print(train_df.shape, val_df.shape)

     (2401, 2) (601, 2)
```

The category labels must be converted to string type because we use the flow_from_directory method for image generator.

Need to convert label to string type because of error generated during flow_from_directory method

TypeError: If class_mode="binary", y_col="label" column values must be strings.

```
[14] train_df['label'] = train_df['label'].astype('str')
     val_df['label'] = val_df['label'].astype('str')

     print(train_df.head())
     print(val_df.head())
     print(train_df.info(), val_df.info())
```

MODULE III – DATA AUGMENTATION

To perform augmentation, we use the inbuilt Keras ImageDataGenerator where we specify a set of transformations that should be applied to our original dataset. Data augmentation helps to prevent over fitting and trains the model with images from different perspectives. Here, we do normalization which divides each input by its standard deviation. Rotation range is set to 0.5, and zoom range to 0.1. The images are shifter by a percentage of 25 both horizontally as well as vertically. Similarly, horizontal and vertical flips are performed. Rescaling is also done.

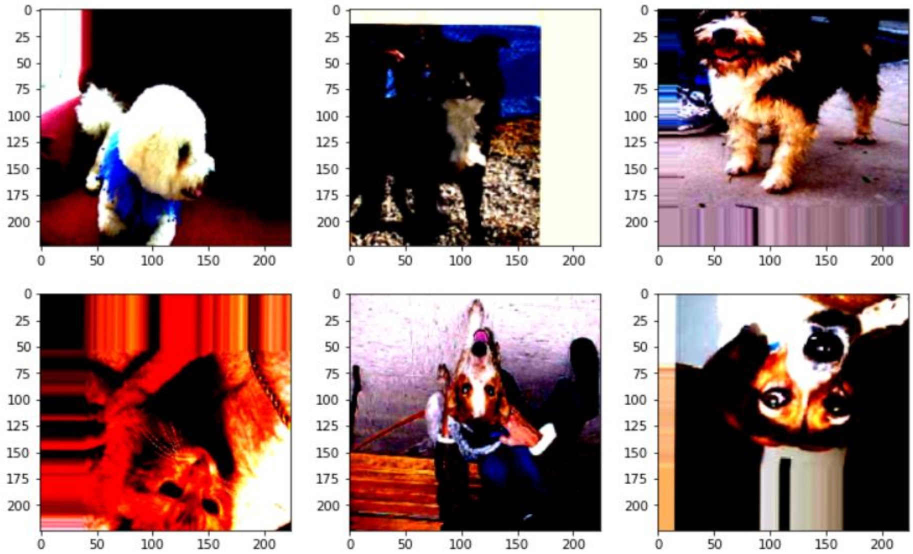
Data Augmentation

For training dataset alone, do augmentation. For validation and testing, just rescaling (normalizing) is enough

```
[15] datagen = ImageDataGenerator(
            featurewise_center=False,  # set input mean to 0 over the dataset
            samplewise_center=False,  # set each sample mean to 0
            featurewise_std_normalization=False,  # divide inputs by std of the dataset
            samplewise_std_normalization=True,  # divide each input by its std
            zca_whitening=False,  # dimesion reduction
            rotation_range=0.5,  # randomly rotate images in the range
            zoom_range = 0.1, # Randomly zoom image
            width_shift_range=0.25,  # randomly shift images horizontally
            height_shift_range=0.25,  # randomly shift images vertically
            horizontal_flip=True,  # randomly flip images
            vertical_flip=True, # randomly flip images
            rescale=1./255, # rescale between 0 and 1
     )
```

RESULT OF DATA AUGMENTATION

Here, we see that there is vertical shift is done for the dog picture in 1st row 3rd column. Similarly, rotation is done for 2 dog pictures in the second row along with rotation and vertical shift for a cat picture in 2nd row 1st column.



MODULE IV - CREATING GENERATORS

The ImageDataGenerator class is very useful in image classification. There are several ways to use this generator, depending on the method we use, here we have used flow_from_directory method that takes a path to the directory containing images sorted in sub directories and image augmentation parameters.

This is created for training, validation and testing images.

In flow_from_directory method, there are certain parameters that need to be set.

dataframe: Mentions the name of dataframe
directory: The directory from which the images must be read
x_col: The filename here
y_col: The label
batch_size: Size of batches for data
shuffle: If randomization must be performed
class_mode: binary here since there are only 2 classes
target_size: Size of image

Training Generator

Since images are stored as file and not as arrays, we need to extract using flow_from_dataframe method

```
[16] train_generator = datagen.flow_from_dataframe(
                    dataframe = train_df, # name of dataframe
                    directory = "./train_samp/", # direcory where images are stored/
                    x_col = "filename", # name of the file is stored in x_col
                    y_col = "label", # label in y_col
                    batch_size = 128, # size of batches of data
                    seed = 5, # random no for shuffling and transformation
                    shuffle = True, # whether to shuffle the data or not
                    class_mode = "binary", # binary implies 2 classes are present
                    target_size=(224, 224) # target size of images
     )

     Found 2401 validated image filenames belonging to 2 classes.
```

Validation Generator

```
[18] validation_datagen = ImageDataGenerator(rescale=1./255)

     validation_generator = validation_datagen.flow_from_dataframe(
                     dataframe = val_df, # name of dataframe
                     directory = "./train_samp/", # direcory where images are stored
                     x_col = "filename", # name of the file is stored in x_col
                     y_col = "label", # label in y_col
                     batch_size = 128, # size of batches of data
                     class_mode = 'binary', # binary implies 2 classes are present
                     target_size = (224, 224), # target size of images
     )
```

```
Found 601 validated image filenames belonging to 2 classes.
```

Test Generator

```
[20] test_gen = ImageDataGenerator(rescale=1./255)

     test_generator = test_gen.flow_from_dataframe(
                     dataframe = test_df, # name of dataframe
                     directory = "./test_samp", # direcory where images are stored
                     x_col = "filename", # name of the file is stored in x_col
                     y_col = None, # label in y_col
                     # batch_size = 16, # size of batches of data
                     class_mode = None, # binary implies 2 classes are present
                     target_size = (256, 256), # target size of images
                     # shuffle = False
     )
```

```
Found 100 validated image filenames.
```

MODULE V – BUILDING VGG16 MODEL WITHOUT FINE TUNING

```
from keras.applications import VGG16

input_shape = (image_size, image_size, 3)

pre_trained_model = VGG16(input_shape=input_shape, include_top=False, weights="imagenet")

last_layer = pre_trained_model.get_layer('block5_pool')
last_output = last_layer.output

# Flatten the output layer to 1 dimension
x = GlobalMaxPooling2D()(last_output)
# Add a fully connected layer with 512 hidden units and ReLU activation
x = Dense(512, activation='relu')(x)
# Add a dropout rate of 0.5
x = Dropout(0.5)(x)
# Add a final sigmoid layer for classification
x = layers.Dense(1, activation='sigmoid')(x)

model = Model(pre_trained_model.input, x)

model.compile(loss='binary_crossentropy',
              optimizer=optimizers.SGD(lr=1e-4, momentum=0.9),
              metrics=['accuracy'])

model.summary()
```

The VGG16 model has 16 deep layers with pre-trained model. **'include_top'** is set to **False.** The weights used here is from **"imagenet"** in which they are stored. This implies the finally connected layer is not used according to the model since the problem here is only to categorize between 2 classes. For the finally connected layer, we use **512 nodes** with **'relu'** activation and a **dropout of 0.5.** The final layer has **1 node** whose activation function is the **'sigmoid'** function. The model here uses **Stochastic Gradient Descent** model and **loss function** is **"binary cross entropy"**.

MODEL SUMMARY

```
Model: "model"

_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
input_1 (InputLayer)        [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)       (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)       (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)  (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)       (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)       (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)  (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)       (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)       (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)       (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)  (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)       (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)       (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)       (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)  (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)       (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)       (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)       (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)  (None, 7, 7, 512)         0
_________________________________________________________________
global_max_pooling2d (Global (None, 512)              0
_________________________________________________________________
dense (Dense)               (None, 512)               262656
_________________________________________________________________
dropout (Dropout)           (None, 512)               0
_________________________________________________________________
dense_1 (Dense)             (None, 1)                 513
=================================================================
Total params: 14,977,857
Trainable params: 14,977,857
Non-trainable params: 0
_________________________________________________________________
```

MODULE VI – TRAINING VGG16 MODEL WITHOUT FINE TUNING

Here, we use Learning Rate Scheduler which changes learning rate accordingly. The initial learning rate is 0.001. **EarlyStopping** is another technique that is used to prevent over fitting with a **patience** level of **5**. We set **epochs** to **50**.

Training Basic VGG Model without any fine tuning

Learning Rate Scheduler

```
[22] reduce_lr = LearningRateScheduler(lambda x: 1e-3 * 0.9 ** x)
```

```
[23] early_stopping = EarlyStopping(monitor='loss', patience=5, verbose=1)
     history = model.fit_generator((train_generator), epochs=50,
                                   validation_data = (validation_generator), verbose=1,
     #                               batch_size = 100,
                                   # steps_per_epoch = 5,
                                   callbacks = [reduce_lr, early_stopping])
```

Accuracy in the first epoch is around 54% whereas at the end of 38 epochs it is around 98%. We also see that early stopping has occurred at epoch number 38.

```
/usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:1844: UserWarning: `Model.fit_generator` is deprecated and
  warnings.warn('`Model.fit_generator` is deprecated and '
Epoch 1/50
19/19 [==============================] - 341s 17s/step - loss: 0.9188 - accuracy: 0.5402 - val_loss: 0.6525 - val_accuracy: 0.6090
Epoch 2/50
19/19 [==============================] - 45s 2s/step - loss: 0.5311 - accuracy: 0.7319 - val_loss: 0.7882 - val_accuracy: 0.5674
Epoch 3/50
19/19 [==============================] - 44s 2s/step - loss: 0.3204 - accuracy: 0.8628 - val_loss: 0.5546 - val_accuracy: 0.6905
Epoch 4/50
19/19 [==============================] - 44s 2s/step - loss: 0.2261 - accuracy: 0.9095 - val_loss: 0.5515 - val_accuracy: 0.6922
Epoch 5/50
19/19 [==============================] - 45s 2s/step - loss: 0.1754 - accuracy: 0.9292 - val_loss: 0.4253 - val_accuracy: 0.7770
Epoch 6/50
19/19 [==============================] - 44s 2s/step - loss: 0.1512 - accuracy: 0.9380 - val_loss: 0.4140 - val_accuracy: 0.7804
Epoch 7/50
19/19 [==============================] - 44s 2s/step - loss: 0.1345 - accuracy: 0.9419 - val_loss: 0.4880 - val_accuracy: 0.7354
Epoch 8/50
19/19 [==============================] - 45s 2s/step - loss: 0.1477 - accuracy: 0.9432 - val_loss: 0.4824 - val_accuracy: 0.7421
Epoch 9/50
19/19 [==============================] - 45s 2s/step - loss: 0.1132 - accuracy: 0.9505 - val_loss: 0.3595 - val_accuracy: 0.8170
Epoch 10/50
19/19 [==============================] - 45s 2s/step - loss: 0.1082 - accuracy: 0.9617 - val_loss: 0.2703 - val_accuracy: 0.8652
Epoch 11/50
19/19 [==============================] - 45s 2s/step - loss: 0.0993 - accuracy: 0.9617 - val_loss: 0.3686 - val_accuracy: 0.8087
Epoch 12/50
19/19 [==============================] - 45s 2s/step - loss: 0.0852 - accuracy: 0.9683 - val_loss: 0.3317 - val_accuracy: 0.8270
Epoch 13/50
19/19 [==============================] - 45s 2s/step - loss: 0.0929 - accuracy: 0.9651 - val_loss: 0.3110 - val_accuracy: 0.8469
Epoch 14/50
Epoch 33/50
19/19 [==============================] - 43s 2s/step - loss: 0.0504 - accuracy: 0.9818 - val_loss: 0.3014 - val_accuracy: 0.8519
Epoch 34/50
19/19 [==============================] - 44s 2s/step - loss: 0.0616 - accuracy: 0.9744 - val_loss: 0.3186 - val_accuracy: 0.8403
Epoch 35/50
19/19 [==============================] - 43s 2s/step - loss: 0.0557 - accuracy: 0.9805 - val_loss: 0.3108 - val_accuracy: 0.8486
Epoch 36/50
19/19 [==============================] - 43s 2s/step - loss: 0.0542 - accuracy: 0.9832 - val_loss: 0.3118 - val_accuracy: 0.8486
Epoch 37/50
19/19 [==============================] - 43s 2s/step - loss: 0.0545 - accuracy: 0.9818 - val_loss: 0.3079 - val_accuracy: 0.8519
Epoch 38/50
19/19 [==============================] - 45s 2s/step - loss: 0.0496 - accuracy: 0.9821 - val_loss: 0.3143 - val_accuracy: 0.8486
Epoch 00038: early stopping
```

MODULE VII – PLOTTING GRAPHS FOR VGG16 MODEL WITHOUT FINE TUNING

Plotting Graphs - Basic VGG Model

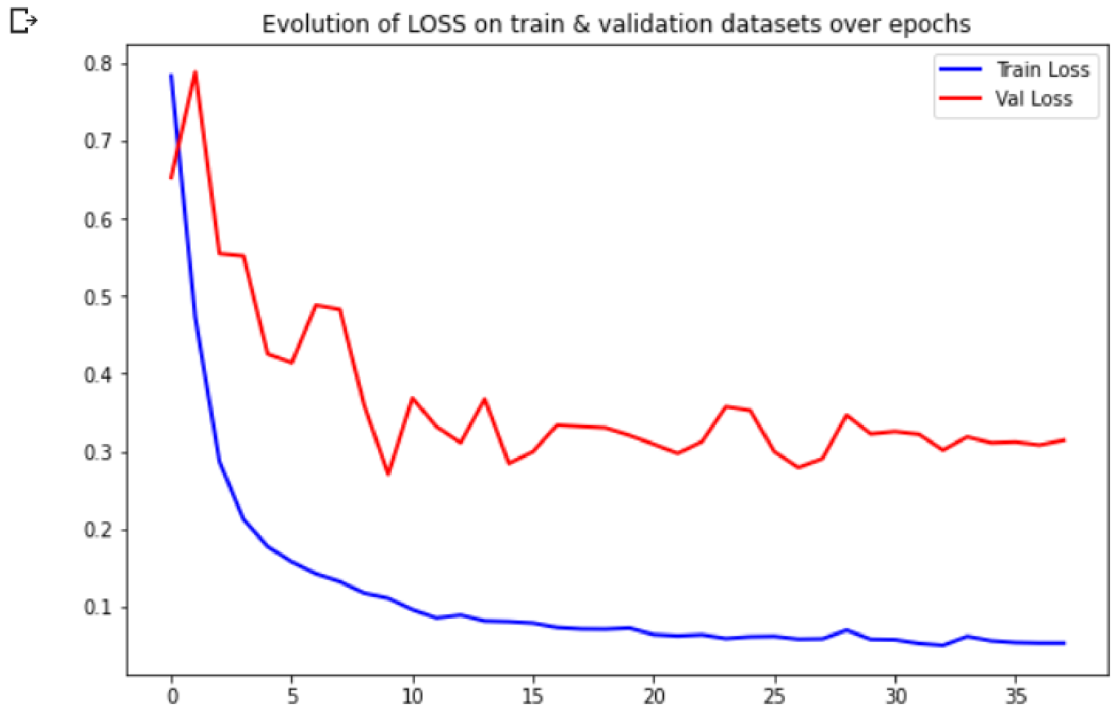
```
[24] # Check how loss & mae went down
     epoch_loss = history.history['loss']
     epoch_val_loss = history.history['val_loss']
     epoch_acc = history.history['accuracy']
     epoch_val_acc = history.history['val_accuracy']

     plt.figure(figsize=(20,6))
     plt.subplot(1,2,1)
     plt.plot(range(0,len(epoch_loss)), epoch_loss, 'b-', linewidth=2, label='Train Loss')
     plt.plot(range(0,len(epoch_val_loss)), epoch_val_loss, 'r-', linewidth=2, label='Val Loss')
     plt.title('Evolution of LOSS on train & validation datasets over epochs')
     plt.legend(loc='best')

     plt.subplot(1,2,2)
     plt.plot(range(0,len(epoch_acc)), epoch_acc, 'b-', linewidth=2, label='Train Accuracy')
     plt.plot(range(0,len(epoch_val_acc)), epoch_val_acc, 'r-', linewidth=2,label='Val Accuracy')
     plt.title('Evolution of ACCURACY on train & validation datasets over epochs')
     plt.legend(loc='best')

     plt.show()
```

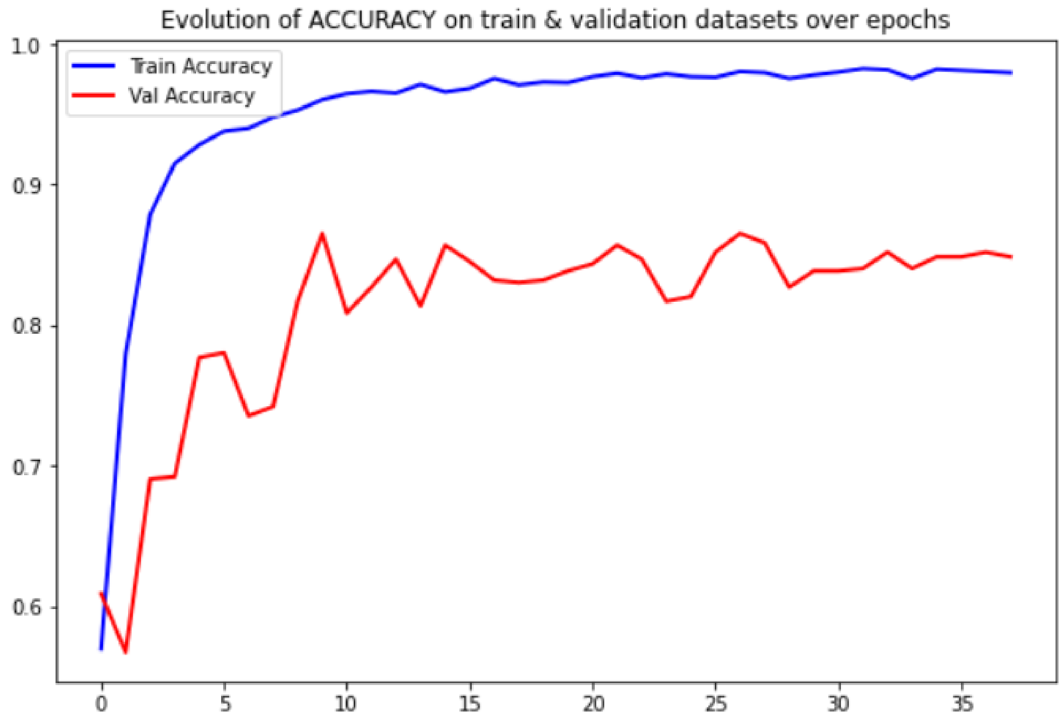
TRAINING AND VALIDATION GRAPHS FOR LOSS



We observe that training loss at end of 38 epochs is less than 0.1 and validation loss ia around 0.35

TRAINING AND VALIDATION GRAPHS FOR ACCURACY

Training accuracy is around 0.85 at the end of 38 epochs for validation and 985 for training. We see that this gap is acceptable.



VALIDATION LOSS AND ACCURACY

```
[25] loss, accuracy = model.evaluate_generator(validation_generator, workers=12)
     print("Validation: accuracy = %f  ;  loss = %f " % (accuracy, loss))

     /usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:1877
       warnings.warn('`Model.evaluate_generator` is deprecated and '
     Validation: accuracy = 0.848586  ;  loss = 0.314254
```

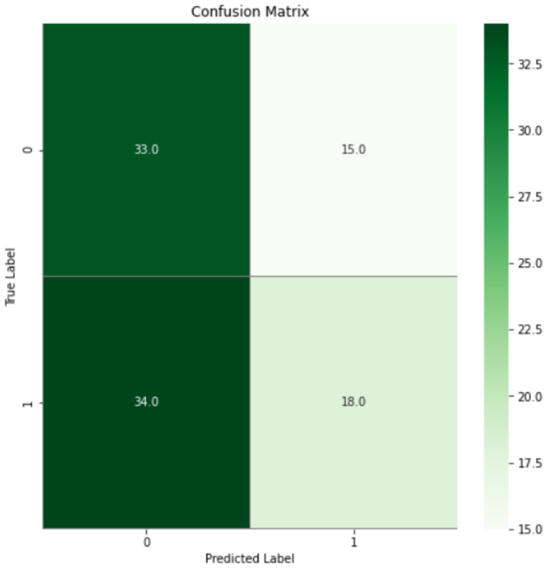
MODULE VIII - PREDICTION & EVALUATION

CONFUSION MATRIX

Confusion Matrix and Classification report- Basic VGG16 Model

```
[76] # compute the confusion matrix
     Y_test_actual = Y_test_actual.astype('str')
     Y_test_pred = Y_test_pred.astype('str')
     # y_final = y_final.reshape
     # print(type(Y_val), type(y_final), Y_val.shape, y_final.shape, len(Y_val_list[100]), len(Y_final_list))
     confusion_mtx = confusion_matrix(Y_test_actual, Y_test_pred)

     f,ax = plt.subplots(figsize=(8, 8))
     sns.heatmap(confusion_mtx, annot=True, linewidths=0.01,cmap="Greens",linecolor="gray", fmt= '.1f',ax=ax)
     plt.xlabel("Predicted Label")
     plt.ylabel("True Label")
     plt.title("Confusion Matrix")
     plt.show()
```



We see that on the testing set accuracy is only around 55 %

CLASSIFICATION REPORT

```
[41] report = classification_report(Y_test_actual, Y_test_pred, target_names=['0','1'])
     print(report)
```

```
              precision    recall  f1-score   support

           0       0.49      0.71      0.58        48
           1       0.55      0.33      0.41        52

    accuracy                           0.51       100
   macro avg       0.52      0.52      0.50       100
weighted avg       0.52      0.51      0.49       100
```

MODULE V – BUILDING VGG16 MODEL WITH FINE TUNING

In **fine-tuning**, we freeze a part of model and few layers of the model are not frozen. The layers that are not frozen learn new weights and are trained again with a very low learning rate. This can potentially achieve meaningful improvements, by incrementally adapting the pre-trained features to the new data.

The more similar the tasks are, the more number of layers must be frozen.
layers.trainable = False, implies the layer is frozen
layers.trainable = True implies the layer is ready to be trained.

Here, we freeze the bottom 15 layers and the other layers are left to be trained.

Building model - Basic VGG16 model with fine tuning

[26]
```
from keras.applications import VGG16

image_size = 224
input_shape = (image_size, image_size, 3)

pre_trained_model2 = VGG16(input_shape=input_shape, include_top=False, weights="imagenet")

for layer in pre_trained_model2.layers[:15]:
    layer.trainable = False

for layer in pre_trained_model2.layers[15:]:
    layer.trainable = True

for layer in pre_trained_model2.layers:
  print(layer.name, layer.trainable)
```

```
input_2 False
block1_conv1 False
block1_conv2 False
block1_pool False
block2_conv1 False
block2_conv2 False
block2_pool False
block3_conv1 False
block3_conv2 False
block3_conv3 False
block3_pool False
block4_conv1 False
block4_conv2 False
block4_conv3 False
block4_pool False
block5_conv1 True
block5_conv2 True
block5_conv3 True
block5_pool True
```

The last 4 layers are not frozen and left to be trained.

```python
last_layer = pre_trained_model2.get_layer('block5_pool')
last_output = last_layer.output

# Flatten the output layer to 1 dimension
x = GlobalMaxPooling2D()(last_output)
# Add a fully connected layer with 512 hidden units and ReLU activation
x = Dense(512, activation='relu')(x)
# Add a dropout rate of 0.5
x = Dropout(0.5)(x)
# Add a final sigmoid layer for classification
x = layers.Dense(1, activation='sigmoid')(x)

VGG16model_finetuned = Model(pre_trained_model2.input, x)

VGG16model_finetuned.compile(loss='binary_crossentropy',
              optimizer=optimizers.Adam(lr=1e-4),
              metrics=['accuracy'])

VGG16model_finetuned.summary()
```

As said above, this fine-tuned model has few layers that are not frozen. These layers will be trained.

The VGG16 model has 16 deep layers with pre-trained model. **'include_top'** is set to **False.** The weights used here is from **"imagenet"** in which they are stored. This implies the finally connected layer is not used according to the model since the problem here is only to categorize between 2 classes. For the finally connected layer, we use **512 nodes** with **'relu'** activation and a **dropout of 0.5.** The final layer has **1 node** whose activation function is the **'sigmoid'** function. The model here uses **Adam optimiser** and **loss function** is **"binary cross entropy"**.

MODEL SUMMARY

```
Model: "model_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
global_max_pooling2d_1 (Glob (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 513
=================================================================
Total params: 14,977,857
Trainable params: 7,342,593
Non-trainable params: 7,635,264
_________________________________________________________________
```

MODULE VI – TRAINING VGG16 MODEL WITH FINE TUNING

Here, we use Learning Rate Scheduler which changes learning rate accordingly. The initial learning rate is 0.001. We set **epochs** to **25** and **patience level** of 5 since fine tuning yields results faster and better.

Training Basic VGG Model with fine tuning

```
early_stopping = EarlyStopping(monitor='loss', patience=5, verbose=1)
history = VGG16model_finetuned.fit_generator((train_generator), epochs=25,
                              validation_data = (validation_generator), verbose=1,
                              callbacks = [early_stopping]
                              )
```

Accuracy in the first epoch is around 66%. But we see at the second epoch itself, this has increased to 89%. This is attributed to the fine-tuning we have done. The final accuracy is 97%.

```
Epoch 1/25
38/38 [==============================] - 38s 980ms/step - loss: 0.7285 - accuracy: 0.6643 - val_loss: 0.6856 - val_accuracy: 0.6040
Epoch 2/25
38/38 [==============================] - 37s 980ms/step - loss: 0.2400 - accuracy: 0.8917 - val_loss: 0.6054 - val_accuracy: 0.6672
Epoch 3/25
38/38 [==============================] - 37s 974ms/step - loss: 0.1625 - accuracy: 0.9333 - val_loss: 0.5745 - val_accuracy: 0.6822
Epoch 4/25
38/38 [==============================] - 37s 970ms/step - loss: 0.1384 - accuracy: 0.9471 - val_loss: 0.4740 - val_accuracy: 0.7554
Epoch 5/25
38/38 [==============================] - 37s 970ms/step - loss: 0.0961 - accuracy: 0.9624 - val_loss: 0.4390 - val_accuracy: 0.7737
Epoch 6/25
38/38 [==============================] - 37s 967ms/step - loss: 0.0969 - accuracy: 0.9646 - val_loss: 0.4350 - val_accuracy: 0.7937
Epoch 7/25
38/38 [==============================] - 37s 972ms/step - loss: 0.0871 - accuracy: 0.9677 - val_loss: 0.4253 - val_accuracy: 0.7837
Epoch 8/25
38/38 [==============================] - 37s 969ms/step - loss: 0.0768 - accuracy: 0.9704 - val_loss: 0.4130 - val_accuracy: 0.8136
Epoch 9/25
38/38 [==============================] - 37s 972ms/step - loss: 0.0473 - accuracy: 0.9844 - val_loss: 0.2671 - val_accuracy: 0.8752
Epoch 10/25
38/38 [==============================] - 37s 979ms/step - loss: 0.0581 - accuracy: 0.9742 - val_loss: 0.3629 - val_accuracy: 0.8386
Epoch 11/25
38/38 [==============================] - 37s 978ms/step - loss: 0.0502 - accuracy: 0.9827 - val_loss: 0.2890 - val_accuracy: 0.8752
Epoch 12/25
38/38 [==============================] - 37s 975ms/step - loss: 0.0491 - accuracy: 0.9862 - val_loss: 0.3509 - val_accuracy: 0.8403
Epoch 13/25
38/38 [==============================] - 37s 976ms/step - loss: 0.0512 - accuracy: 0.9799 - val_loss: 0.2325 - val_accuracy: 0.8918
Epoch 14/25
38/38 [==============================] - 37s 975ms/step - loss: 0.0331 - accuracy: 0.9891 - val_loss: 0.2176 - val_accuracy: 0.9035
Epoch 15/25
38/38 [==============================] - 37s 972ms/step - loss: 0.0286 - accuracy: 0.9893 - val_loss: 0.2603 - val_accuracy: 0.8902
```

```
Epoch 21/25
38/38 [==============================] - 37s 969ms/step - loss: 0.0310 - accuracy: 0.9875 - val_loss: 0.2227 - val_accuracy: 0.8935
Epoch 22/25
38/38 [==============================] - 37s 971ms/step - loss: 0.0114 - accuracy: 0.9970 - val_loss: 0.3187 - val_accuracy: 0.8785
Epoch 23/25
38/38 [==============================] - 37s 964ms/step - loss: 0.0168 - accuracy: 0.9960 - val_loss: 0.2573 - val_accuracy: 0.8985
Epoch 24/25
38/38 [==============================] - 37s 965ms/step - loss: 0.0111 - accuracy: 0.9965 - val_loss: 0.6943 - val_accuracy: 0.7720
Epoch 25/25
38/38 [==============================] - 37s 964ms/step - loss: 0.0607 - accuracy: 0.9784 - val_loss: 0.2913 - val_accuracy: 0.8686
```

MODULE VII – PLOTTING GRAPHS FOR VGG16 MODEL WITH FINE TUNING

Plotting Graphs - Basic VGG Model

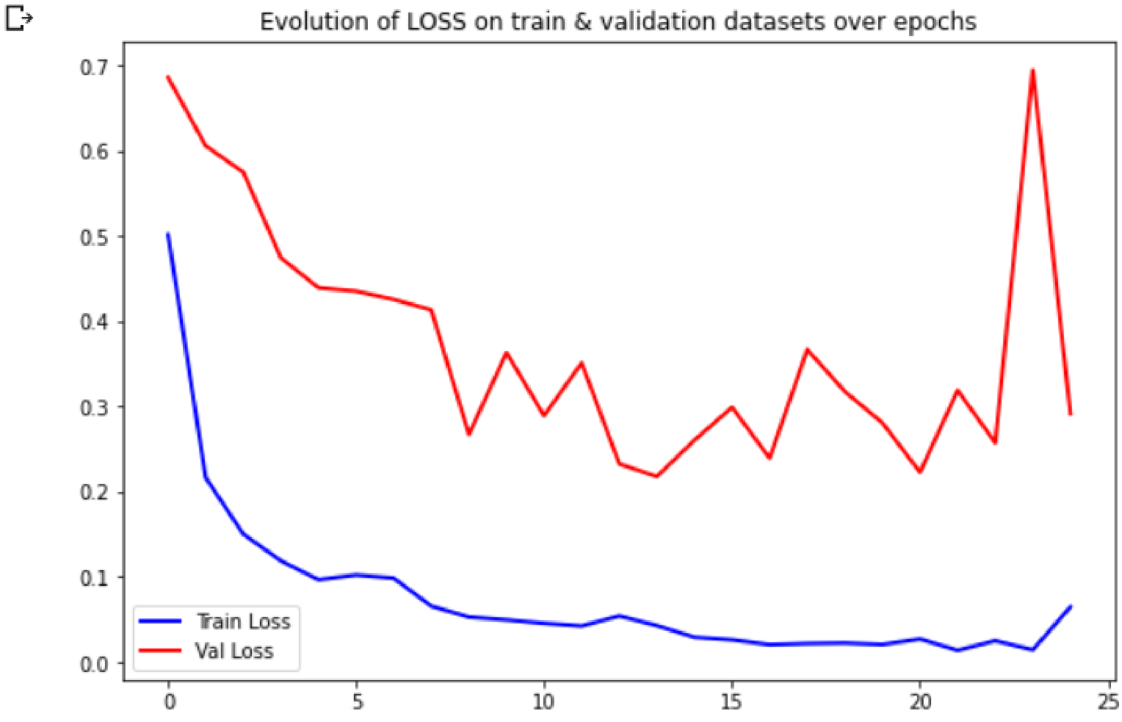
```
[24] # Check how loss & mae went down
     epoch_loss = history.history['loss']
     epoch_val_loss = history.history['val_loss']
     epoch_acc = history.history['accuracy']
     epoch_val_acc = history.history['val_accuracy']

     plt.figure(figsize=(20,6))
     plt.subplot(1,2,1)
     plt.plot(range(0,len(epoch_loss)), epoch_loss, 'b-', linewidth=2, label='Train Loss')
     plt.plot(range(0,len(epoch_val_loss)), epoch_val_loss, 'r-', linewidth=2, label='Val Loss')
     plt.title('Evolution of LOSS on train & validation datasets over epochs')
     plt.legend(loc='best')

     plt.subplot(1,2,2)
     plt.plot(range(0,len(epoch_acc)), epoch_acc, 'b-', linewidth=2, label='Train Accuracy')
     plt.plot(range(0,len(epoch_val_acc)), epoch_val_acc, 'r-', linewidth=2,label='Val Accuracy')
     plt.title('Evolution of ACCURACY on train & validation datasets over epochs')
     plt.legend(loc='best')

     plt.show()
```

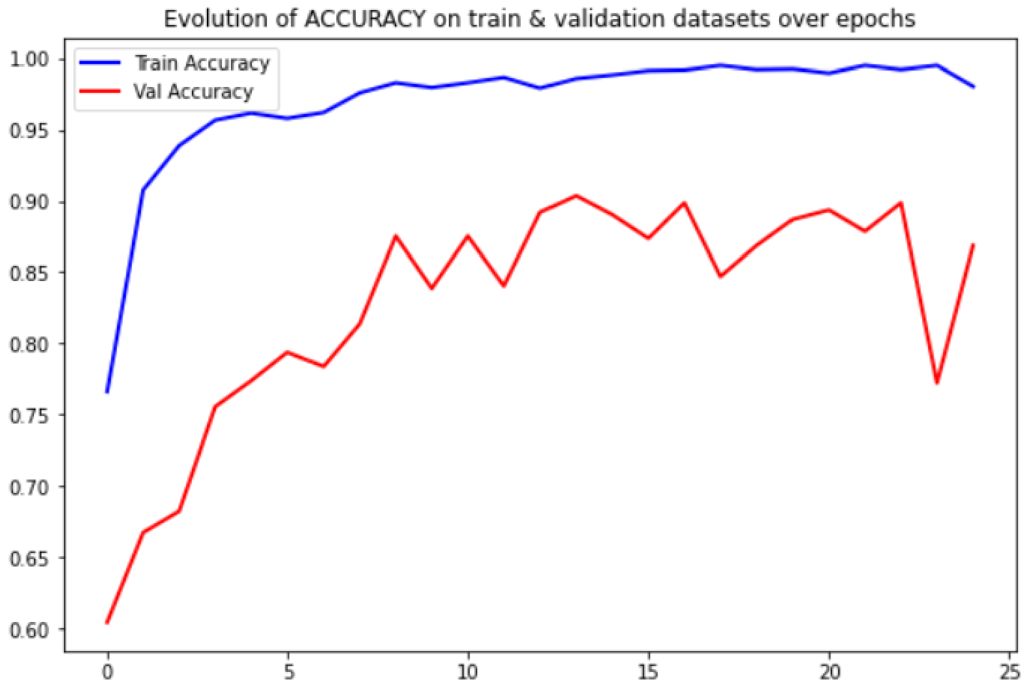
TRAINING AND VALIDATION GRAPHS FOR LOSS



We observe that training loss at end of 25 epochs is less than 0.1 and validation loss has fluctuated in the end

TRAINING AND VALIDATION GRAPHS FOR ACCURACY

Training accuracy is around 97% at the end of 25 epochs which is better than the model that was not fine-tuned (85% accuracy). Similarly, validation accuracy is also around 85%



VALIDATION LOSS AND ACCURACY

```
[79] loss, accuracy = VGG16model_finetuned.evaluate_generator(validation_generator, workers=12)
     print("Validation: accuracy = %f  ;  loss = %f " % (accuracy, loss))

     /usr/local/lib/python3.7/dist-packages/tensorflow/python/keras/engine/training.py:1877: UserWarning:
       warnings.warn('`Model.evaluate_generator` is deprecated and '
     Validation: accuracy = 0.868552  ;  loss = 0.291348
```

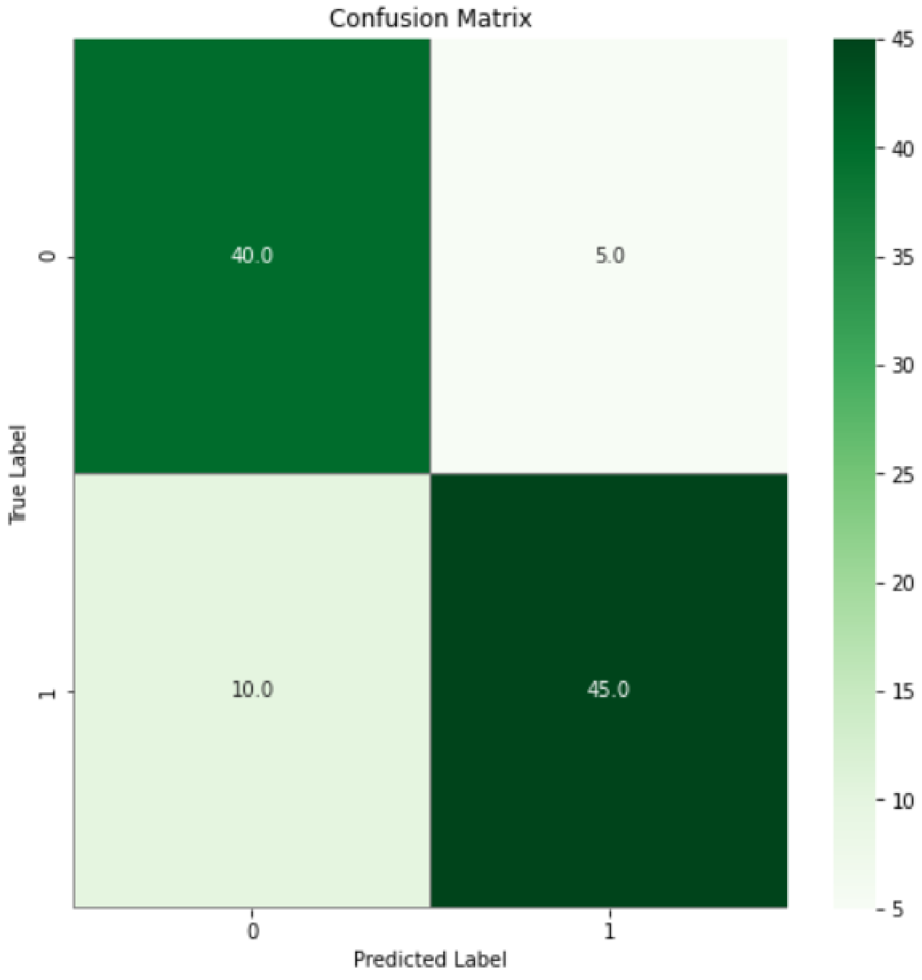
MODULE VIII - PREDICTION & EVALUATION

CONFUSION MATRIX

Confusion Matrix and Classification report- Basic VGG16 Model

```
[76] # compute the confusion matrix
     Y_test_actual = Y_test_actual.astype('str')
     Y_test_pred = Y_test_pred.astype('str')
     # y_final = y_final.reshape
     # print(type(Y_val), type(y_final), Y_val.shape, y_final.shape, len(Y_val_list[100]), len(Y_final_list))
     confusion_mtx = confusion_matrix(Y_test_actual, Y_test_pred)

     f,ax = plt.subplots(figsize=(8, 8))
     sns.heatmap(confusion_mtx, annot=True, linewidths=0.01,cmap="Greens",linecolor="gray", fmt= '.1f',ax=ax)
     plt.xlabel("Predicted Label")
     plt.ylabel("True Label")
     plt.title("Confusion Matrix")
     plt.show()
```



This is much times better than the results without fine-tuned model

CLASSIFICATION REPORT

We observe that accuracy here is 85% on testing set compared to 51% on the VGG16 without any fine tuning.

```
[90] report = classification_report(Y_test_actual, Y_test_pred, target_names=['0','1'])
     print(report)
```

```
              precision    recall  f1-score   support

           0       0.80      0.89      0.84        45
           1       0.90      0.82      0.86        55

    accuracy                           0.85       100
   macro avg       0.85      0.85      0.85       100
weighted avg       0.85      0.85      0.85       100
```

PREDICTING IMAGES

The images in the first row are predicted properly. In the second row, second column, the dog is mislabelled as cat(0).