2. IMAGE CLASSIFICATION MODEL FOR ACCURATE PREDICTIONS OF TRAFFIC LIGHT

2.1 DATA PREPARATION

Two separate folders within a zip file contain 100 images of red signals and 100 images of green signals, making up our image dataset. Following the upload of our zip file to Google Drive, the drive is first mounted in Google Colab, and then the zip file is unzipped.

To facilitate the process of data preparation, the image dataset can be loaded and preprocessed from the directory using TensorFlow's Keras API (tf.keras.utils.image_dataset_from_directory). This utility function's description is displayed below.

```
Signature:
tf.keras.utils.image_dataset_from_directory(directory,
labels='inferred', label_mode='int',
class_names=None, color_mode='rgb',
batch_size=32, image_size=(2 items) (256, 256),
shuffle=True, seed=None, validation_split=None,
subset=None, interpolation='bilinear',
follow_links=False, crop_to_aspect_ratio=False,
**kwargs)
Source:
@keras_export(
    "keras.utils.image_dataset_from_directory",

"keras.preprocessing.image_dataset_from_directory",
    v1=[],
)
def image_dataset_from_directory(
    directory,
    labels="inferred",
    label_mode="int",
    class_names=None,
    color_mode="rgb",
    batch_size=32,
    image_size=(256, 256),
    shuffle=True,
    seed=None,
    validation_split=None,
    subset=None,
    interpolation="bilinear",
    follow_links=False,
```

It is evident that this utility function resizes the images to (256,256) and applies labels to them automatically based on the names of our folders. The dataset has a batch size of 32.

```
# Building a image dataset for our model

data = tf.keras.utils.image_dataset_from_directory(data_dir)
```

```
Found 200 files belonging to 2 classes.
```

```
[ ] # Coverting the data into numpy iterator to access each batch from our data.

    data_iterator = data.as_numpy_iterator()
```

```
[ ] # Grabing the first batch

    batch = data_iterator.next()
```

```
[ ] # Shape of images in the first batch

    batch[0].shape
```

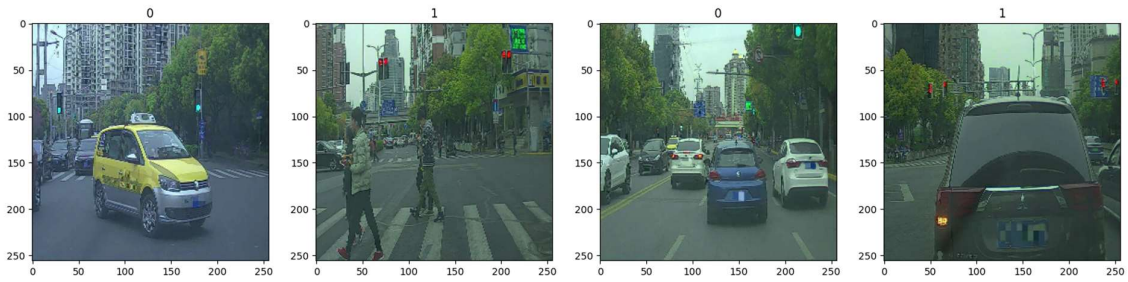
```
(32, 256, 256, 3)
```

```
[ ] # Labels of images in the first batch

    batch[1]
```

```
array([0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1,
       0, 1, 0, 0, 1, 0, 1, 0, 1, 1], dtype=int32)
```

From the image above, we can observe that the utility function receives our unzipped directory first. It then applies the image processing steps we previously saw and discovers that our dataset comprises 200 images divided into the red and green classes.

After that, we convert our TensorFlow dataset into a NumPy iterator so that we can access each batch. A batch consists of two elements: the 32 images (256,256,3) that make up its first element, and their corresponding labels as the second element.



The red images are labelled as 1, and the green images are labelled as 0.

Before the data is fed into the CNN model, the image's pixel value is scaled.

```
# Minimum and Maximum pixel value of the images.

print("Minimum value before scaling: ", batch[0].min())
print("Maximum value before scaling: ", batch[0].max())
```

```
Minimum value before scaling:  0.0
Maximum value before scaling:  255.0
```

```
[ ] # Scaling the pixel values between [0, 1]

data = data.map(lambda x, y : (x/255, y))

data_iterator = data.as_numpy_iterator()
batch = data_iterator.next()
print("Minimum value after scaling: ", batch[0].min())
print("Maximum value after scaling: ", batch[0].max())
```

```
Minimum value after scaling:  0.0
Maximum value after scaling:  1.0
```

Our dataset comprises 7 batches, as shown in the figure below: 5 batches are used for training, 1 batch is used for validation, and 1 batch is used for testing.

```
# Total number of batches

len(data)
```

```
7
```

```
[ ] # Deciding the no of batches for training, validation and testing

train_size = int(len(data) * .7) + 1
val_size = int(len(data) * .2)
test_size = int(len(data) * .1) + 1

print(train_size)
print(val_size)
print(test_size)
```

```
5
1
1
```

```
[ ] # Spliting the data into training, validation and testing

train_data = data.take(train_size)
val_data = data.skip(train_size).take(val_size)
test_data = data.skip(train_size + val_size).take(test_size)
```

The data augmentation technique of random flipping is used to horizontally flip images, thereby improving the robustness of our training data and enabling our model to generalize well to new, unseen images.

```
[ ]  def augment_image(image, label):

         # Apply data augmentation such as random flip horizontally
         image = tf.image.random_flip_left_right(image)

         return image, label
```

```
[ ]  # Apply data augmentation to the training dataset
     train_data = train_data.map(augment_image)
```

2.2 CNN MODEL

Built on a sequential model, the Convolutional Neural Network (CNN) Architecture consists of three convolutional layers: the first and third layers have 16 filters each with a 3x3 kernel and a stride of 1, while the second convolutional layer has 32 filters with the same 3x3 kernel and a stride of 1. The activation function used by all these convolutional layers is ReLU.

All convolutional layers are succeeded by a max-pooling layer to decrease the spatial dimensions of those layers. Subsequently, the 3D output is converted to 1D by adding a single flatten layer, and then it is passed to the fully connected layers. The fully connected layer is built using 256 neurons, which also uses ReLU activation function. Finally, a dense output layer with single neuron and sigmoid activation function is added. Below is the built model.

```
model = Sequential()
```

```
model.add(Conv2D(16, (3,3), 1, activation='relu', input_shape=(256, 256, 3)))
model.add(MaxPooling2D())

model.add(Conv2D(32, (3,3), 1, activation='relu'))
model.add(MaxPooling2D())

model.add(Conv2D(16, (3,3), 1, activation='relu'))
model.add(MaxPooling2D())

model.add(Flatten())

model.add(Dense(256, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

The model is compiled using the Adam optimizer, which computes the loss based on the accuracy metrics using binary cross-entropy.

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

The figure below displays the model's summary.

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_15 (Conv2D)          (None, 254, 254, 16)      448

 max_pooling2d_15 (MaxPooli  (None, 127, 127, 16)      0
 ng2D)

 conv2d_16 (Conv2D)          (None, 125, 125, 32)      4640

 max_pooling2d_16 (MaxPooli  (None, 62, 62, 32)        0
 ng2D)

 conv2d_17 (Conv2D)          (None, 60, 60, 16)        4624

 max_pooling2d_17 (MaxPooli  (None, 30, 30, 16)        0
 ng2D)

 flatten_5 (Flatten)         (None, 14400)             0

 dense_10 (Dense)            (None, 256)               3686656

 dense_11 (Dense)            (None, 1)                 257

=================================================================
Total params: 3696625 (14.10 MB)
Trainable params: 3696625 (14.10 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

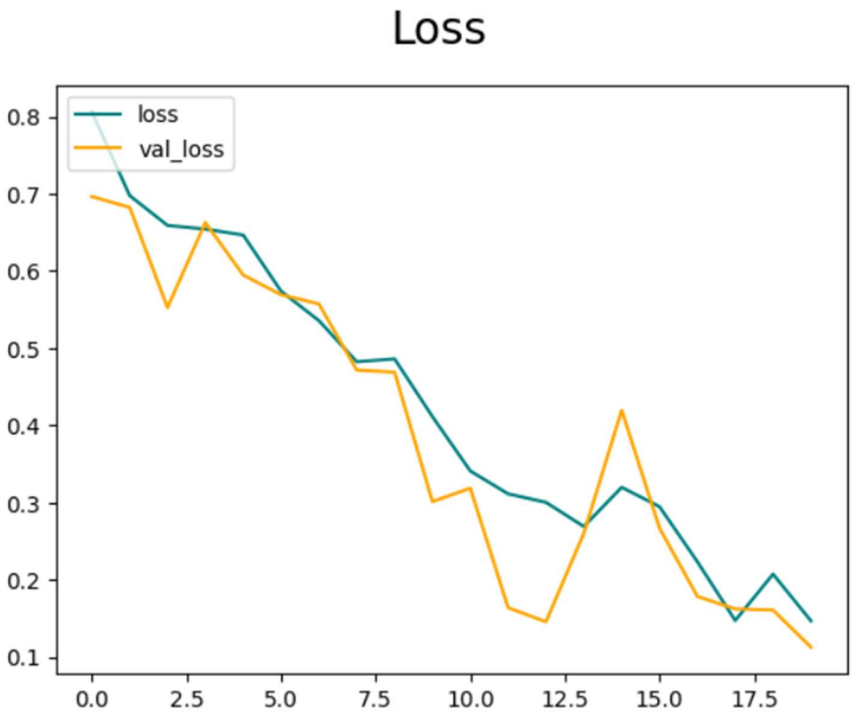
We can see the output shapes and the quantity of parameters for every layer from the model summary. There is a small loss of information that was extracted from each max-pooling layer because we didn't set the padding, as evidenced by the variations in the convolutional layers' output shapes. This has no effect on the performance of our models, so the padding is left unset.

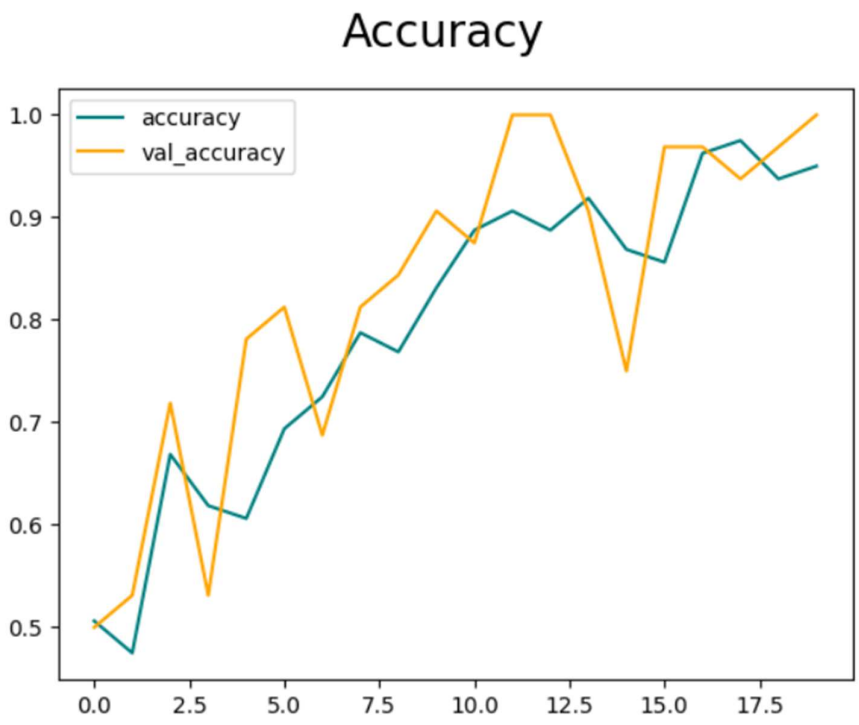
2.3 TRAINING THE MODEL

Now the built model is trained for 20 epochs on the training set and validated using validation set.

```
[ ]  History = model.fit(train_data, epochs = 20, validation_data=val_data)
```

Below are the results of both the training and validation.





It's clear from the above results that our model performs well because the loss decreases with each epoch and the training and validation accuracies increase. However, it should be noted that by increasing the number of epochs, the loss can still be reduced and ultimately brought to a global minimum, which could improve the performance of our model even further.

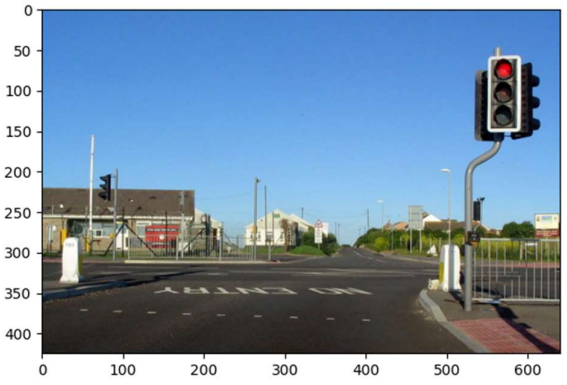
2.4 EVALUATING THE MODELS PERFORMANCE

With the images in the test set, we can evaluate the model's performance and determine how well it works.

```
Confusion Matrix:
 [[4 0]
 [1 3]]

Classification Report:
              precision    recall  f1-score   support

           0       0.80      1.00      0.89         4
           1       1.00      0.75      0.86         4

    accuracy                           0.88         8
   macro avg       0.90      0.88      0.87         8
weighted avg       0.90      0.88      0.87         8

ROC AUC: 0.875
Precision: 1.000
Recall: 0.750
F1-Score: 0.857
Accuracy: 0.875
```

The results above demonstrate that, with an accuracy of 87.5% overall, our model correctly classifies all the green signal images while incorrectly classifying one red signal image as green. Using a random selection of traffic signal images from Google, we further test the generalisation of our model. The result of this testing is shown below.

2.5 TESTING THE MODEL WITH TRAFFIC LIGHT IMAGES FROM THE WEB




```
1/1 [==============================] - 0s 20ms/step
[[1.]]

Predicted class is Red
```

```
1/1 [==============================] - 0s 20ms/step
[[0.00615904]]

Predicted class is Green
```

```
1/1 [==============================] - 0s 20ms/step
[[0.1897628]]

Predicted class is Green
```

```
1/1 [==============================] - 0s 26ms/step
[[0.07291652]]

Predicted class is Green
```

```
1/1 [==============================] - 0s 18ms/step
[[0.24886955]]

Predicted class is Green
```

```
1/1 [==============================] - 0s 19ms/step
[[3.322106e-06]]

Predicted class is Green
```

The probabilities of each output are displayed, and it is evident that the two red signal images were incorrectly classified as green. For the unseen red signal images, the model's generalization is poor. By letting the model train for additional epochs and preventing overfitting, we can attempt to further improve it.

2.6 INCREASING THE MODELS PERFORMANCE

Reducing the batch size speeds up the convergence process by allowing the model's weights to be updated more frequently. Additionally, a smaller batch size increases variability, which adds more noise to the training process and may improve the model's ability to generalize.

Considering this theory, the dataset's batch size is reduced to 16, the model is trained for 50 epochs, and the early stopping technique is applied to end the training process before the model begins to overfit. This process is illustrated in the figures below.

```
[ ]  # Changing the batch size

     data = tf.keras.utils.image_dataset_from_directory(data_dir, batch_size=16)
```

```
     Found 200 files belonging to 2 classes.
```

```
[ ]  data_iterator = data.as_numpy_iterator()
     batch = data_iterator.next()
```

```
[ ]  batch[0].shape
```

```
     (16, 256, 256, 3)
```

```
[ ]  batch[1]
```

```
     array([0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1], dtype=int32)
```

Using the map method, the pixel values of every image in each batch are scaled.

```
[ ]  data = data.map(lambda x, y : (x/255, y))

     data_iterator = data.as_numpy_iterator()
     batch = data_iterator.next()
     print("Minimum value after scaling: ", batch[0].min())
     print("Maximum value after scaling: ", batch[0].max())
```

```
     Minimum value after scaling:  0.0
     Maximum value after scaling:  1.0
```

```
len(data)
```

```
13
```

```
train_size = int(len(data) * .7) + 1
val_size = int(len(data) * .2)
test_size = int(len(data) * .1)

print(train_size)
print(val_size)
print(test_size)
```

```
10
2
1
```

```
train_data = data.take(train_size)
val_data = data.skip(train_size).take(val_size)
test_data = data.skip(train_size + val_size).take(test_size)

print(len(train_data))
print(len(val_data))
print(len(test_data))
```

```
10
2
1
```

After separating the dataset into a batch of 16 each, there are total of 13 batches where 10 batches are used for training, 2 for validation and 1 for testing.

The model has now been trained and its performance has been tracked, as illustrated in the figures below, after the previously developed data augmentation method was applied to our training data.

```
train_data = train_data.map(augment_image)
```

```
model = Sequential()
model.add(Conv2D(16, (3,3), 1, activation='relu', input_shape=(256, 256, 3)))
model.add(MaxPooling2D())

model.add(Conv2D(32, (3,3), 1, activation='relu'))
model.add(MaxPooling2D())

model.add(Conv2D(16, (3,3), 1, activation='relu'))
model.add(MaxPooling2D())

model.add(Flatten())

model.add(Dense(256, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

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

```
model.summary()
```

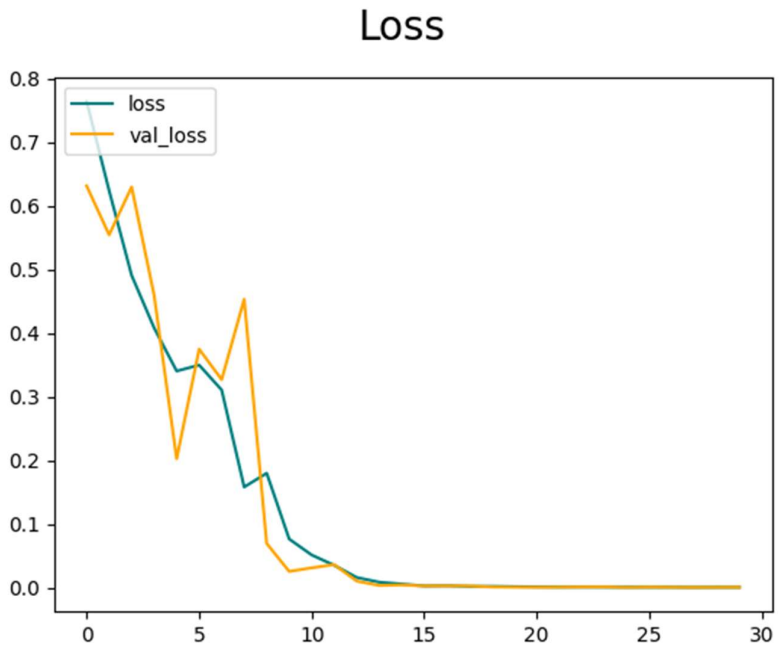
```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_21 (Conv2D)          (None, 254, 254, 16)      448

 max_pooling2d_21 (MaxPooli  (None, 127, 127, 16)      0
 ng2D)

 conv2d_22 (Conv2D)          (None, 125, 125, 32)      4640

 max_pooling2d_22 (MaxPooli  (None, 62, 62, 32)        0
 ng2D)

 conv2d_23 (Conv2D)          (None, 60, 60, 16)        4624

 max_pooling2d_23 (MaxPooli  (None, 30, 30, 16)        0
 ng2D)

 flatten_8 (Flatten)         (None, 14400)             0

 dense_16 (Dense)            (None, 256)               3686656

 dense_17 (Dense)            (None, 1)                 257

=================================================================
Total params: 3696625 (14.10 MB)
Trainable params: 3696625 (14.10 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

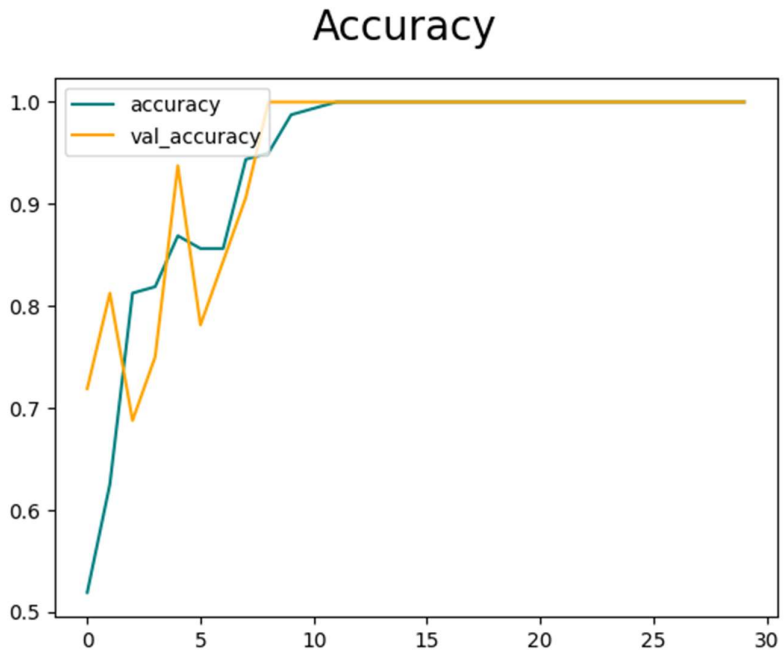
When the validation loss of our model remains constant for five consecutive epochs, the training process is terminated by using the early stopping with patience set to 5, and the best weights before the stop is applied to the model.

```
from tensorflow.keras.callbacks import EarlyStopping

# Early Stopping Callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

History = model.fit(train_data, epochs=50, validation_data=val_data, callbacks=[early_stopping])
```

As we can see from the image below, that the models training process is stopped at the 30th epoch, before overfitting starts.

```
Epoch 25/50
10/10 [==============================] - 2s 132ms/step - loss: 5.0067e-04 - accuracy: 1.0000 - val_loss: 2.0728e-04 - val_accuracy: 1.0000
Epoch 26/50
10/10 [==============================] - 2s 189ms/step - loss: 4.6938e-04 - accuracy: 1.0000 - val_loss: 7.0896e-04 - val_accuracy: 1.0000
Epoch 27/50
10/10 [==============================] - 2s 149ms/step - loss: 4.5280e-04 - accuracy: 1.0000 - val_loss: 4.2028e-04 - val_accuracy: 1.0000
Epoch 28/50
10/10 [==============================] - 3s 227ms/step - loss: 2.5487e-04 - accuracy: 1.0000 - val_loss: 2.1143e-04 - val_accuracy: 1.0000
Epoch 29/50
10/10 [==============================] - 3s 188ms/step - loss: 3.8930e-04 - accuracy: 1.0000 - val_loss: 3.7597e-04 - val_accuracy: 1.0000
Epoch 30/50
10/10 [==============================] - 2s 133ms/step - loss: 3.5077e-04 - accuracy: 1.0000 - val_loss: 3.7757e-04 - val_accuracy: 1.0000
```





It is evident that the loss has been sufficiently reduced, the model has attained a high validation accuracy, and it is stable. It makes it very evident that this model is superior to the one that came before it.

Below are the model's performance metrics on the test set.

```
Confusion Matrix:
 [[4 0]
 [0 4]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         4
           1       1.00      1.00      1.00         4

    accuracy                           1.00         8
   macro avg       1.00      1.00      1.00         8
weighted avg       1.00      1.00      1.00         8

ROC AUC: 1.000
Precision: 1.000
Recall: 1.000
F1-Score: 1.000
Accuracy: 1.000
```

Every image in the test batch has been accurately classified by the model with a 100% accuracy rate. The traffic signal images that are collected from the internet are also used to test the model.

```
1/1 [==============================] - 0s 17ms/step
[[1.]]

Predicted class is Red
```

```
1/1 [==============================] - 0s 17ms/step
[[4.1645562e-05]]

Predicted class is Green
```

```
1/1 [==============================] - 0s 24ms/step
[[0.9999993]]

Predicted class is Red
```

```
1/1 [==============================] - 0s 20ms/step
[[0.0009644]]

Predicted class is Green
```

```
1/1 [==============================] - 0s 19ms/step
[[0.8841285]]

Predicted class is Red
```

```
1/1 [==============================] - 0s 26ms/step
[[6.2701916e-13]]

Predicted class is Green
```

The model's ability to generalize to previously unseen, novel images is demonstrated by its accurate classification of every web image. As a result, the traffic light signals can be 100% accurately classified using this CNN model.

2.7 TRANSFER LEARNING WITH VGG16 ARCHITECTURE

To ensure that our model is appropriate for this problem, we must first compare it to other, more widely used models, like VGG16, which was designed specifically for this type of problem.

The VGG16 model is instantiated by setting the pre-trained weights from the ImageNet, the model's final dense layers are also excluded by setting 'include_top=False'. Additionally, a loop is used to freeze the model's layers, preventing the pre-trained weights from changing while the model is being trained. After the convolutional base architecture, a flatten layer and a dense layer with 32 neurons that have ReLU activation are added. Lastly, a single neuron with a sigmoid activation function is added to the output layer.

```
[141] from keras.applications import VGG16
      import keras
      from keras import layers
      conv_base = VGG16(weights='imagenet', include_top=False, input_shape=(256, 256, 3))
```

```
for layer in conv_base.layers:
    layer.trainable = False
model_vgg = Sequential([
    conv_base,
    Flatten(),
    Dense(32, activation='relu'),
    Dense(1, activation = "sigmoid")
])

model_vgg.compile(optimizer ='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

```
[143] model_vgg.summary()

Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 8, 8, 512)         14714688

 flatten_4 (Flatten)         (None, 32768)             0

 dense_8 (Dense)             (None, 32)                1048608

 dense_9 (Dense)             (None, 1)                 33

=================================================================
Total params: 15763329 (60.13 MB)
Trainable params: 1048641 (4.00 MB)
Non-trainable params: 14714688 (56.13 MB)
_________________________________________________________________
```

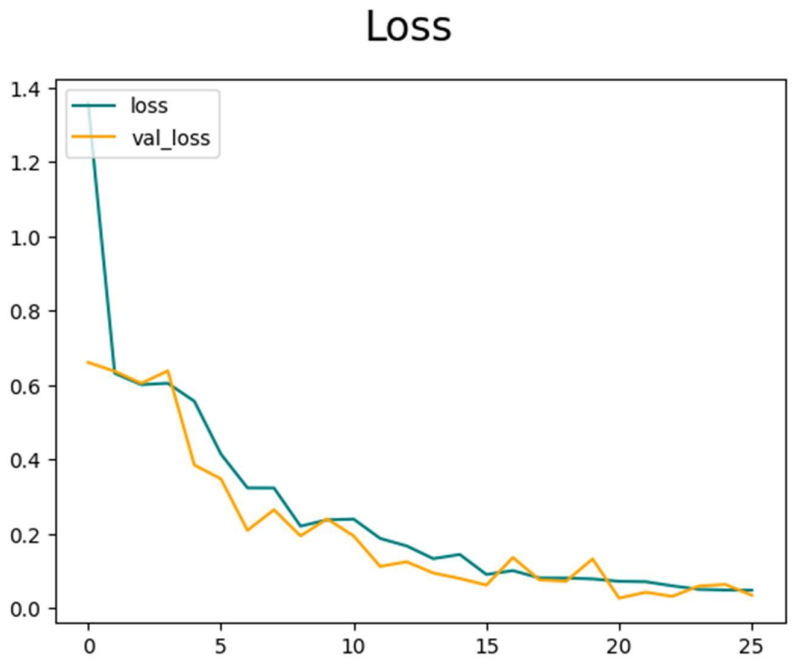
The summary of the model above shows us how many parameters are trainable and how many are frozen.

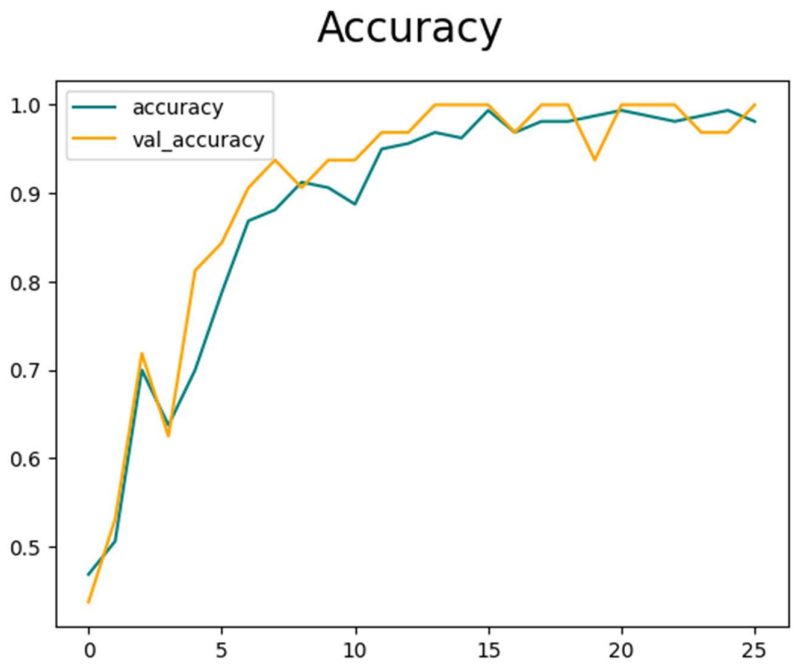
After the model is constructed, it is trained for 50 epochs on the training batch, with early stopping set to 'patience=5' to track the validation loss.

```
from tensorflow.keras.callbacks import EarlyStopping

# Early Stopping Callback
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

History = model_vgg.fit(train_data, epochs=50, validation_data=val_data, callbacks=[early_stopping])
```

```
Epoch 1/50
10/10 [==============================] - 6s 427ms/step - loss: 1.3584 - accuracy: 0.4688 - val_loss: 0.6613 - val_accuracy: 0.4375
Epoch 2/50
10/10 [==============================] - 3s 244ms/step - loss: 0.6322 - accuracy: 0.5063 - val_loss: 0.6376 - val_accuracy: 0.5312
Epoch 3/50
10/10 [==============================] - 3s 210ms/step - loss: 0.6017 - accuracy: 0.7000 - val_loss: 0.6052 - val_accuracy: 0.7188
Epoch 4/50
10/10 [==============================] - 2s 202ms/step - loss: 0.6054 - accuracy: 0.6375 - val_loss: 0.6386 - val_accuracy: 0.6250
Epoch 5/50
10/10 [==============================] - 4s 384ms/step - loss: 0.5570 - accuracy: 0.7000 - val_loss: 0.3854 - val_accuracy: 0.8125
Epoch 6/50
10/10 [==============================] - 3s 212ms/step - loss: 0.4150 - accuracy: 0.7875 - val_loss: 0.3482 - val_accuracy: 0.8438
Epoch 7/50
10/10 [==============================] - 3s 203ms/step - loss: 0.3239 - accuracy: 0.8687 - val_loss: 0.2097 - val_accuracy: 0.9062
Epoch 8/50
10/10 [==============================] - 2s 203ms/step - loss: 0.3236 - accuracy: 0.8813 - val_loss: 0.2652 - val_accuracy: 0.9375
Epoch 9/50
10/10 [==============================] - 3s 207ms/step - loss: 0.2210 - accuracy: 0.9125 - val_loss: 0.1947 - val_accuracy: 0.9062
Epoch 10/50
10/10 [==============================] - 3s 211ms/step - loss: 0.2379 - accuracy: 0.9062 - val_loss: 0.2405 - val_accuracy: 0.9375
Epoch 11/50
10/10 [==============================] - 2s 203ms/step - loss: 0.2399 - accuracy: 0.8875 - val_loss: 0.1947 - val_accuracy: 0.9375
Epoch 12/50
10/10 [==============================] - 3s 200ms/step - loss: 0.1879 - accuracy: 0.9500 - val_loss: 0.1128 - val_accuracy: 0.9688
Epoch 13/50
10/10 [==============================] - 2s 196ms/step - loss: 0.1677 - accuracy: 0.9563 - val_loss: 0.1251 - val_accuracy: 0.9688
Epoch 14/50
10/10 [==============================] - 4s 386ms/step - loss: 0.1335 - accuracy: 0.9688 - val_loss: 0.0949 - val_accuracy: 1.0000
Epoch 15/50
10/10 [==============================] - 3s 204ms/step - loss: 0.1448 - accuracy: 0.9625 - val_loss: 0.0802 - val_accuracy: 1.0000
Epoch 16/50
10/10 [==============================] - 2s 202ms/step - loss: 0.0910 - accuracy: 0.9937 - val_loss: 0.0625 - val_accuracy: 1.0000
Epoch 17/50
10/10 [==============================] - 2s 202ms/step - loss: 0.1014 - accuracy: 0.9688 - val_loss: 0.1366 - val_accuracy: 0.9688
Epoch 18/50
10/10 [==============================] - 2s 206ms/step - loss: 0.0816 - accuracy: 0.9812 - val_loss: 0.0766 - val_accuracy: 1.0000
Epoch 19/50
10/10 [==============================] - 4s 390ms/step - loss: 0.0815 - accuracy: 0.9812 - val_loss: 0.0726 - val_accuracy: 1.0000
Epoch 20/50
10/10 [==============================] - 2s 200ms/step - loss: 0.0790 - accuracy: 0.9875 - val_loss: 0.1327 - val_accuracy: 0.9375
Epoch 21/50
10/10 [==============================] - 3s 208ms/step - loss: 0.0726 - accuracy: 0.9937 - val_loss: 0.0273 - val_accuracy: 1.0000
Epoch 22/50
10/10 [==============================] - 2s 201ms/step - loss: 0.0715 - accuracy: 0.9875 - val_loss: 0.0431 - val_accuracy: 1.0000
Epoch 23/50
10/10 [==============================] - 3s 233ms/step - loss: 0.0605 - accuracy: 0.9812 - val_loss: 0.0322 - val_accuracy: 1.0000
Epoch 24/50
10/10 [==============================] - 2s 196ms/step - loss: 0.0510 - accuracy: 0.9875 - val_loss: 0.0596 - val_accuracy: 0.9688
Epoch 25/50
10/10 [==============================] - 2s 203ms/step - loss: 0.0490 - accuracy: 0.9937 - val_loss: 0.0641 - val_accuracy: 0.9688
Epoch 26/50
10/10 [==============================] - 3s 209ms/step - loss: 0.0486 - accuracy: 0.9812 - val_loss: 0.0352 - val_accuracy: 1.0000
```

It is evident that the model ceased training at the 26th epoch to avoid overfitting. Every epoch's loss and accuracy are noted and displayed below.



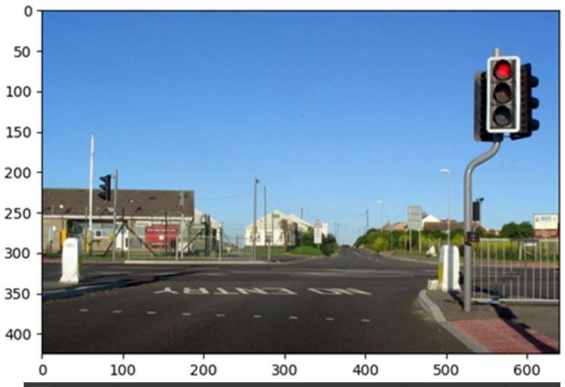


Since it is abundantly evident from the above plots that the model is operating as intended. To evaluate the model's performance on the unseen images, a test batch is now used.


```
Confusion Matrix:
 [[4 0]
 [0 4]]

Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         4
           1       1.00      1.00      1.00         4

    accuracy                           1.00         8
   macro avg       1.00      1.00      1.00         8
weighted avg       1.00      1.00      1.00         8

ROC AUC: 1.0
Precision:1.0
Recall:1.0
F1-Score: 1.0
Accuracy: 1.0
```

The findings demonstrate that the model accurately predicts the test images 100% of the time. Now, we test our model on six images from the web to see

how well it can generalize to predict images that are slightly different from those in our dataset. The results are displayed below.



```
1/1 [==============================] - 0s 27ms/step
[[0.6366783]]

Predicted class is Red
```

```
1/1 [==============================] - 0s 18ms/step
[[0.8228613]]

Predicted class is Red
```

```
1/1 [==============================] - 0s 23ms/step
[[0.78479236]]

Predicted class is Red
```

```
1/1 [==============================] - 0s 18ms/step
[[0.05569917]]

Predicted class is Green
```

```
1/1 [==============================] - 0s 19ms/step
[[0.07956914]]

Predicted class is Green
```

```
1/1 [==============================] - 0s 17ms/step
[[0.14818284]]

Predicted class is Green
```

The results above show that 1 red was incorrectly classified as green and 1 green as red by the transfer learning model VGG16. As a result, while both the previously developed CNN model and the VGG16 transfer learning model correctly classified all the test set images, it is clear from the performance of the web traffic signal images that our CNN model outperformed it because of its superior ability to generalise to new, unseen images.

This is because, although the VGG16 architecture might be more appropriate for other tasks, the CNN model architecture which is built in this process is more appropriate for this particular task.

REFERENCES

Géron, A. (2022). *Hands-on machine learning with Scikit-Learn and TensorFlow concepts, tools, and techniques to build intelligent systems* (3rd ed.). O'Reilly Media, Inc.

tf.keras.utils.image_dataset_from_directory | TensorFlow Core v2.14.0. (n.d.). TensorFlow. https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_dataset_from_directory