TYRE QUALITY INSPECTION USING CONVOLUTIONAL NEURAL NETWORK

INTRODUCTION

Tyre health is crucial for road safety, and early detection of defects is vital for preventing accidents. Traditional methods of tyre inspection often rely on manual visual assessments, which can be time-consuming, subjective, and prone to human error. This project explores the application of Convolutional Neural Networks (CNNs) for automated tyre defect classification, leveraging a dataset of 1854 digital tyre images which is available on https://data.mendeley.com/datasets/bn7ch8tvyp/1.

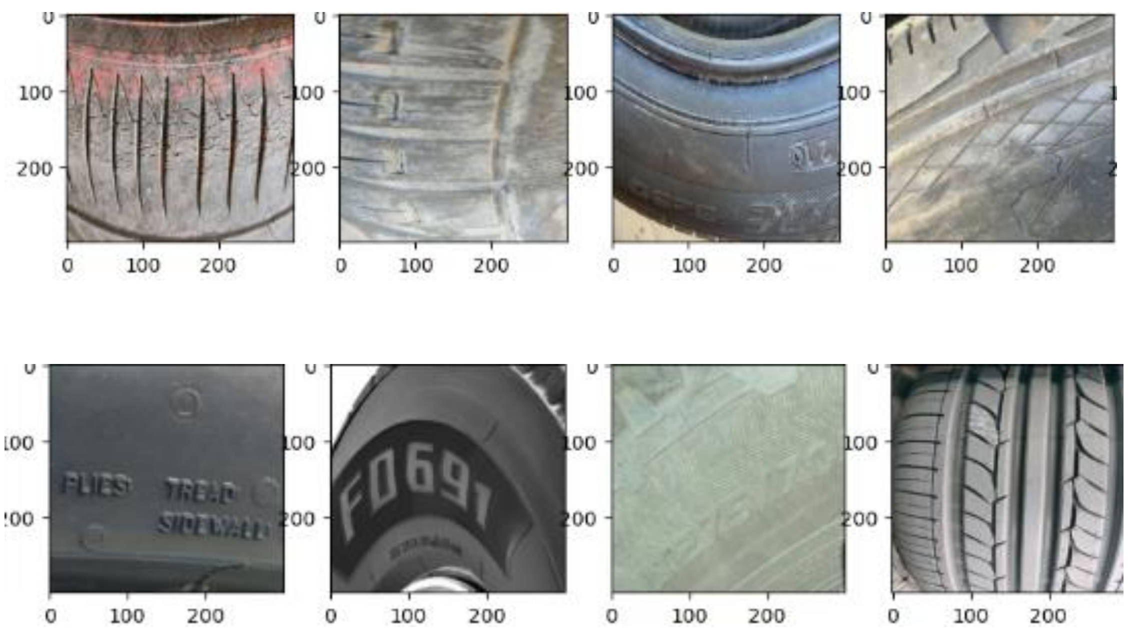
The provided dataset offers a valuable resource for researchers and practitioners in the transportation and automotive sectors. Each image is labelled as either "defective" or "good condition," encompassing a total of two classes. This data allows for the training and testing of machine learning models capable of identifying tyre health from digital images.
The successful development of a CNN-based classification system for tyre defects holds significant promise. Potential benefits include:

- **Enhanced Quality Control:** Automating defect detection can significantly improve the tyre industry's quality control processes, ensuring consistent and reliable assessments.
- **Reduced Accident Risk:** Early identification of defective tyres can prevent accidents caused by tyre failure, leading to improved road safety.
- **Increased Efficiency:** Automating inspections can streamline the process, saving time and resources compared to manual methods.

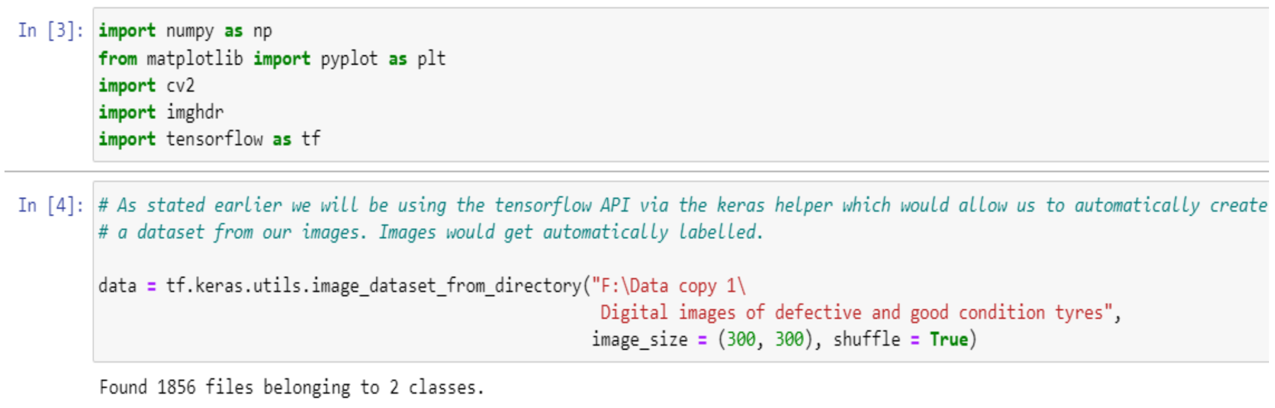
This project investigates the feasibility of utilizing CNNs for tyre defect classification. By training a model on the provided dataset, the project aims to demonstrate the effectiveness of this approach and contribute to the development of more accurate and efficient tyre inspection systems.

INITIAL EXPLORATION AND PREPROCESSING

As stated earlier, the dataset has 1856 total images, with each image labelled either as ‘defective’ or ‘good’. Let us have a look at the tyre images from the dataset. In the images below a category label 1 is a defective tyre



It should also be noted that the images are not of equal size. The images range from size 100 KB to 2.5 MB which means we would have to take care of the fact that all images are resized to equal sizes during the preprocessing stage. For ease of computation, we will keep the image sizes to 300 x 300 pixels. For loading the dataset, we will be using Tensorflow's dataset API. It allows us to create data pipelines. Rather than loading everything into memory, it allows us to create a data pipeline thereby helping us to scale to larger datasets and also gives us a repeatable set of steps. Rather than using it directly we will be using it via a Keras utility.



```
In [3]: import numpy as np
        from matplotlib import pyplot as plt
        import cv2
        import imghdr
        import tensorflow as tf
```

```
In [4]: # As stated earlier we will be using the tensorflow API via the keras helper which would allow us to automatically create
        # a dataset from our images. Images would get automatically labelled.

        data = tf.keras.utils.image_dataset_from_directory("F:\Data copy 1\
                                                            Digital images of defective and good condition tyres",
                                                           image_size = (300, 300), shuffle = True)

        Found 1856 files belonging to 2 classes.
```

We will be dealing with the images such that they are used in batches of 32. Therefore, we will be directly splitting the data batches into train, validation and test. Additionally, we will be scaling the image’s pixel intensities to 0 and 1 by dividing pixel intensities by 255.

```
In [13]: data = data.map(lambda x, y: (x/255, y))
```

SPLIT DATA

```
In [15]: train_size = int(len(data)*0.7)
         val_size = int(len(data)*0.2)
         test_size = int(len(data)*0.1)
```

```
In [16]: train_size, val_size, test_size

Out[16]: (40, 11, 5)
```

```
In [17]: train = data.take(train_size)
         val = data.skip(train_size).take(val_size)
         test = data.skip(train_size+val_size).take(test_size)
```

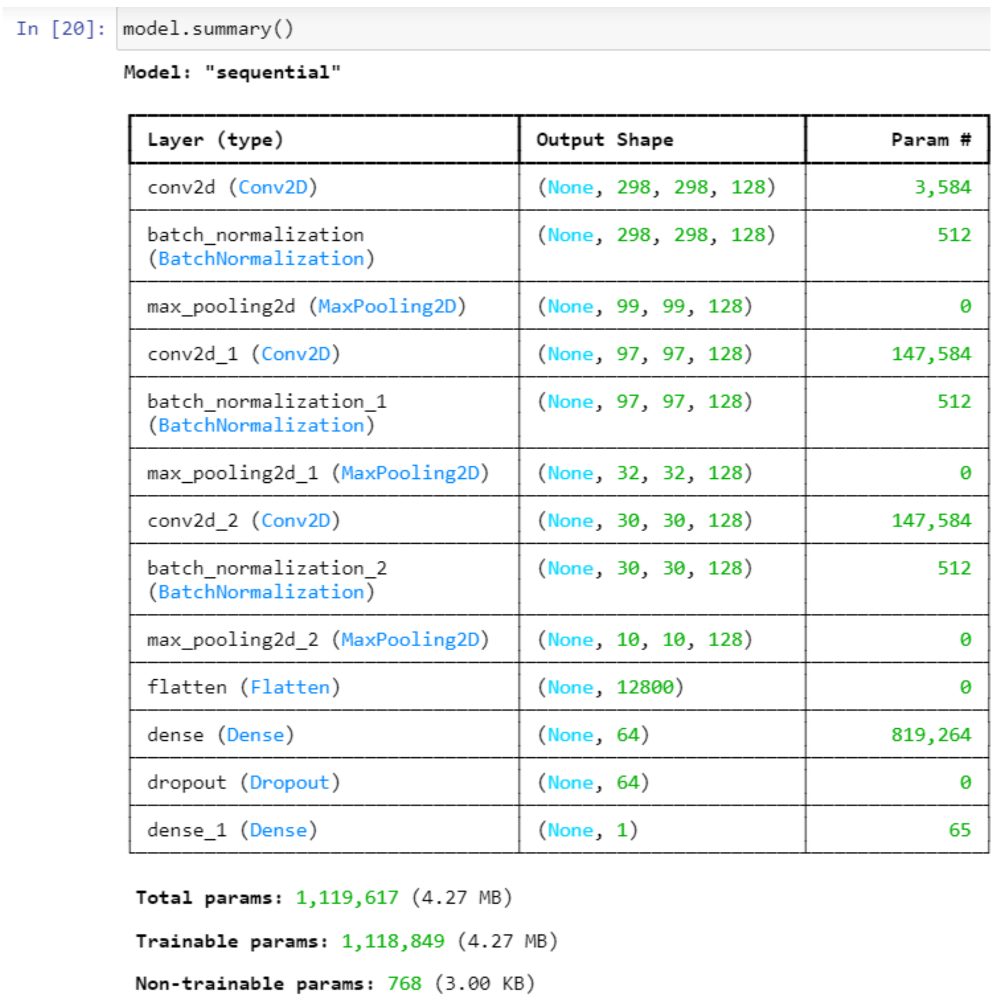
MODEL BUILDING

We will be training our own model from scratch, the main reason being that training our own model allows us the flexibility to train a skilled model that too with lesser number of parameters. Other pretrained models that are used via the method of transfer learning usually have significantly larger number of parameters. For instance, the model that we will be using has only 1,119,617 parameters, while the popular ones usually have parameters in excess of 2 million.

Our model's architecture would have three convolutional layers, three batch normalization layers, three max pooling layers and finally 1 dense layer. We will also be using dropout equal to 0.5. Note that we will be using the Rectified linear unit (ReLu) activation function and instead of the Glorot Uniform initializer, we will be using He-uniform initializer because it works better with the ReLu activation. The overall architecture can be observed from the code block given below:

```
In [19]: model = Sequential()

         # Convolutional layers
         model.add(Conv2D(128, (3,3),activation = 'relu', input_shape = (300, 300, 3),
                          kernel_initializer=keras.initializers.HeUniform()))
         model.add(keras.layers.BatchNormalization())
         model.add(MaxPooling2D(pool_size = (3,3)))
         #He uniform because we are using relu activation function

         model.add(Conv2D(128, (3,3), activation = 'relu',
                          kernel_initializer=keras.initializers.HeUniform()))
         model.add(keras.layers.BatchNormalization())
         model.add(MaxPooling2D(pool_size = (3,3)))

         model.add(Conv2D(128, (3,3),activation = 'relu',
                          kernel_initializer=keras.initializers.HeUniform()))
         model.add(keras.layers.BatchNormalization())
         model.add(MaxPooling2D(3,3))


         # Fully connected layers
         model.add(Flatten())
         model.add(Dense(64, activation = 'relu',
                         kernel_initializer=keras.initializers.HeUniform()))
         model.add(Dropout(0.5))
         model.add(Dense(1, activation = 'sigmoid'))

         # Compile the model

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

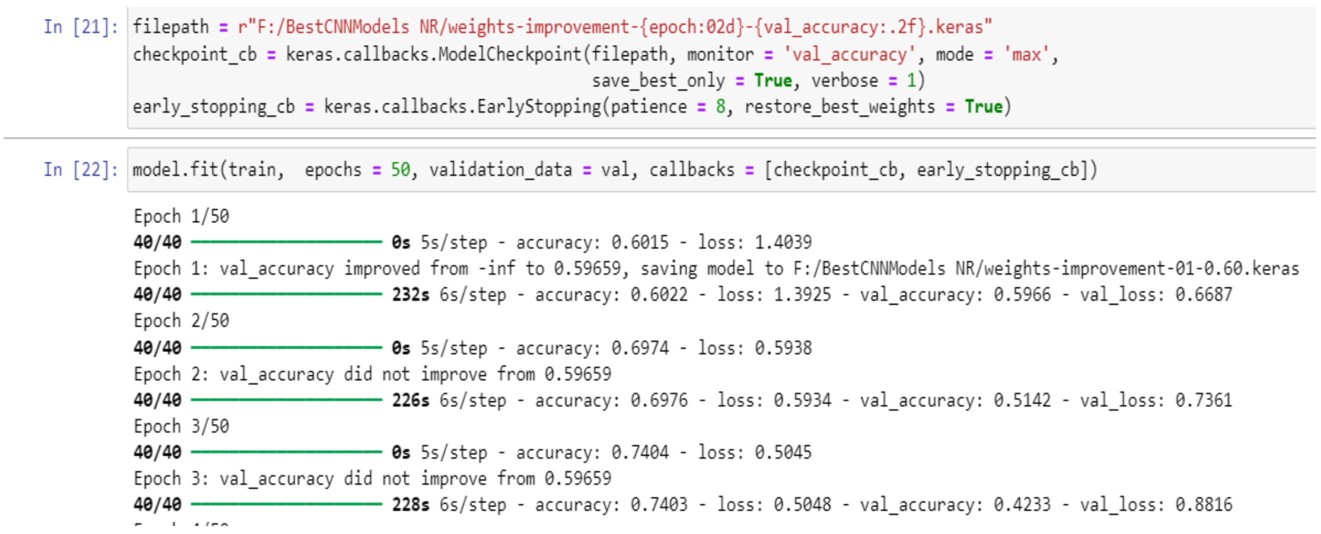
Here is the model summary:



```
In [20]: model.summary()

Model: "sequential"

┃ Layer (type)                     ┃ Output Shape              ┃     Param # ┃
│ conv2d (Conv2D)                  │ (None, 298, 298, 128)     │       3,584 │
│ batch_normalization              │ (None, 298, 298, 128)     │         512 │
│ (BatchNormalization)             │                           │             │
│ max_pooling2d (MaxPooling2D)     │ (None, 99, 99, 128)       │           0 │
│ conv2d_1 (Conv2D)                │ (None, 97, 97, 128)       │     147,584 │
│ batch_normalization_1            │ (None, 97, 97, 128)       │         512 │
│ (BatchNormalization)             │                           │             │
│ max_pooling2d_1 (MaxPooling2D)   │ (None, 32, 32, 128)       │           0 │
│ conv2d_2 (Conv2D)                │ (None, 30, 30, 128)       │     147,584 │
│ batch_normalization_2            │ (None, 30, 30, 128)       │         512 │
│ (BatchNormalization)             │                           │             │
│ max_pooling2d_2 (MaxPooling2D)   │ (None, 10, 10, 128)       │           0 │
│ flatten (Flatten)                │ (None, 12800)             │           0 │
│ dense (Dense)                    │ (None, 64)                │     819,264 │
│ dropout (Dropout)                │ (None, 64)                │           0 │
│ dense_1 (Dense)                  │ (None, 1)                 │          65 │

 Total params: 1,119,617 (4.27 MB)

 Trainable params: 1,118,849 (4.27 MB)

 Non-trainable params: 768 (3.00 KB)
```

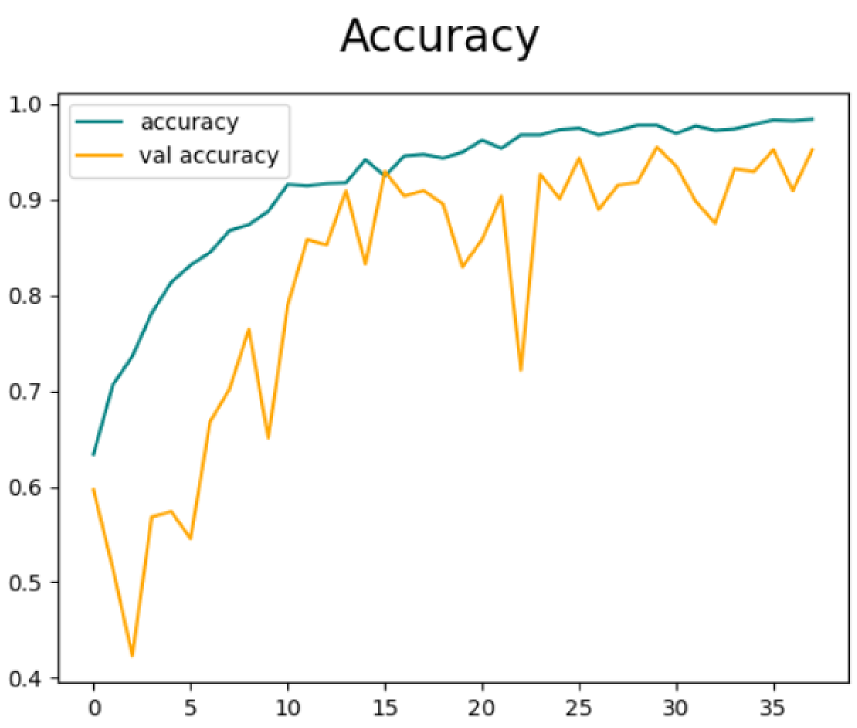
TRAINING THE MODEL

We will be using two checkpoints: Early stopping and Model Checkpoint. The model will be trained on 50 epochs:



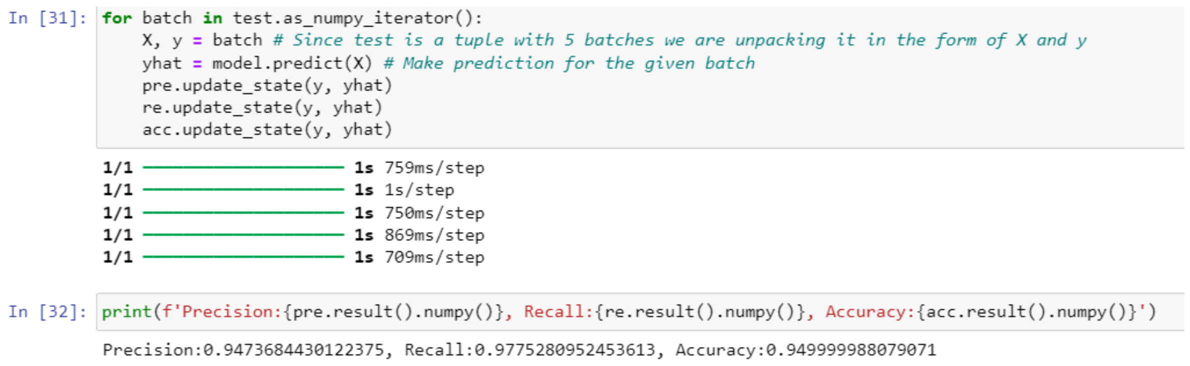
```
In [21]: filepath = r"F:/BestCNNModels NR/weights-improvement-{epoch:02d}-{val_accuracy:.2f}.keras"
         checkpoint_cb = keras.callbacks.ModelCheckpoint(filepath, monitor = 'val_accuracy', mode = 'max',
                                                         save_best_only = True, verbose = 1)
         early_stopping_cb = keras.callbacks.EarlyStopping(patience = 8, restore_best_weights = True)

In [22]: model.fit(train,  epochs = 50, validation_data = val, callbacks = [checkpoint_cb, early_stopping_cb])

         Epoch 1/50
         40/40 ━━━━━━━━━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.6015 - loss: 1.4039
         Epoch 1: val_accuracy improved from -inf to 0.59659, saving model to F:/BestCNNModels NR/weights-improvement-01-0.60.keras
         40/40 ━━━━━━━━━━━━━━━━━━━━ 232s 6s/step - accuracy: 0.6022 - loss: 1.3925 - val_accuracy: 0.5966 - val_loss: 0.6687
         Epoch 2/50
         40/40 ━━━━━━━━━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.6974 - loss: 0.5938
         Epoch 2: val_accuracy did not improve from 0.59659
         40/40 ━━━━━━━━━━━━━━━━━━━━ 226s 6s/step - accuracy: 0.6976 - loss: 0.5934 - val_accuracy: 0.5142 - val_loss: 0.7361
         Epoch 3/50
         40/40 ━━━━━━━━━━━━━━━━━━━━ 0s 5s/step - accuracy: 0.7404 - loss: 0.5045
         Epoch 3: val_accuracy did not improve from 0.59659
         40/40 ━━━━━━━━━━━━━━━━━━━━ 228s 6s/step - accuracy: 0.7403 - loss: 0.5048 - val_accuracy: 0.4233 - val_loss: 0.8816
```

The model accuracy with respect to each epoch looked as follows:



MODEL EVALUATION

The model history clearly shows an unstable improvement in the model's validation accuracy., with every epoch. The model checkpoint callback helped us save the best model which had a 95% accuracy on the validation data. Now it is time to see how the model performs on the test set. For evaluating the model on the test set, we will use accuracy, precision and f1score. The best model had the following results:



```
In [31]: for batch in test.as_numpy_iterator():
             X, y = batch # Since test is a tuple with 5 batches we are unpacking it in the form of X and y
             yhat = model.predict(X) # Make prediction for the given batch
             pre.update_state(y, yhat)
             re.update_state(y, yhat)
             acc.update_state(y, yhat)

         1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 759ms/step
         1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 1s/step
         1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 750ms/step
         1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 869ms/step
         1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 709ms/step

In [32]: print(f'Precision:{pre.result().numpy()}, Recall:{re.result().numpy()}, Accuracy:{acc.result().numpy()}')

         Precision:0.9473684430122375, Recall:0.9775280952453613, Accuracy:0.949999988079071
```

Let us try to get into more details of these results:

- **Precision (0.9473):** This metric indicates that out of all the images our model classified as defective, **94.73% were truly defective**. In other words, the model has a low rate of false positives, meaning it rarely misclassified good condition tyres as defective.

- **Recall (0.9775):** This metric tells us that out of all the actual defective tyres in the dataset, the model correctly classified **97.75%**. This means the model has a good ability to identify true defects and miss very few of them.
- **Accuracy (0.95):** This is the overall proportion of correct predictions made by the model. In this case, the model correctly classified nearly **95%** of the tyres.

Overall, these results suggest that our CNN model performs well in classifying defective and good condition tyres. The high precision indicates the model effectively avoids misclassifications, while the high recall demonstrates its ability to capture most of the true defects.

Additionally:

- A high precision with a slightly lower recall suggests the model prioritizes avoiding false positives. This might be preferable if the cost of misclassifying a good tyre as defective is high (e.g., replacing a good tyre unnecessarily).
- Conversely, a higher recall with a bit lower precision suggests the model prioritizes finding all true defects, even if it leads to some false positives. This might be preferable if the cost of missing a defective tyre is significant (e.g., a tyre failure on the road).

In conclusion, our model demonstrates promising performance for classifying tyre defects.

Let us now try to improve the model using data augmentation. Data augmentation works by creating modified versions of our existing images while preserving the class labels (defective or good condition in our case). These modifications introduce variations that the model might encounter in real-world scenarios, improving its robustness and generalizability. By data augmentation our model would be able to generalise better, but we would not train it from scratch rather we will use our best model from previous training and we will start training it on the new data. The target would be to beat 95% validation accuracy.

For data augmentation, we would do the following:

```
In [44]: batch_size = 32

         # This is the augmentation configuration we will be using for training
         train_datagen = ImageDataGenerator(rescale = 1./255,
                                             rotation_range = 0.2,
                                            zoom_range = 0.2,
                                            horizontal_flip = True,
                                            width_shift_range = 0.2)

         # This is the augmentation configuration we will be using for validation
         # We are simply rescaling the data
         val_datagen = ImageDataGenerator(rescale = 1./255)


         train_generator = train_datagen.flow_from_directory(
                         r"F:\Split_data\train", # This is the target directory
                         target_size = (300, 300), # All images will be resized to 228*228
                         batch_size = batch_size,
                         class_mode = 'binary') # Since we are using binary_cross_entropy

         val_generator = val_datagen.flow_from_directory(
                         r"F:\Split_data\val", # thios is the target directory
                         target_size = (300, 300),
                         batch_size = batch_size,
                         class_mode = 'binary')

         Found 1436 images belonging to 2 classes.
         Found 412 images belonging to 2 classes.
```

Like earlier, we will again use our model checkpoint and early stopping callbacks. By running the model on 50 epochs again, the accuracy of the best model came out to be 93% that too in just 8 epochs. The loss in accuracy is not surprising because the model is being trained on far more different images than earlier. The early stopping callback kicked in which halted the training. So, even though the model's accuracy is lesser than earlier this one is likely to generalize better on new data.

In conclusion our model is performing quite well in differentiating between defective and non-defective tyres.