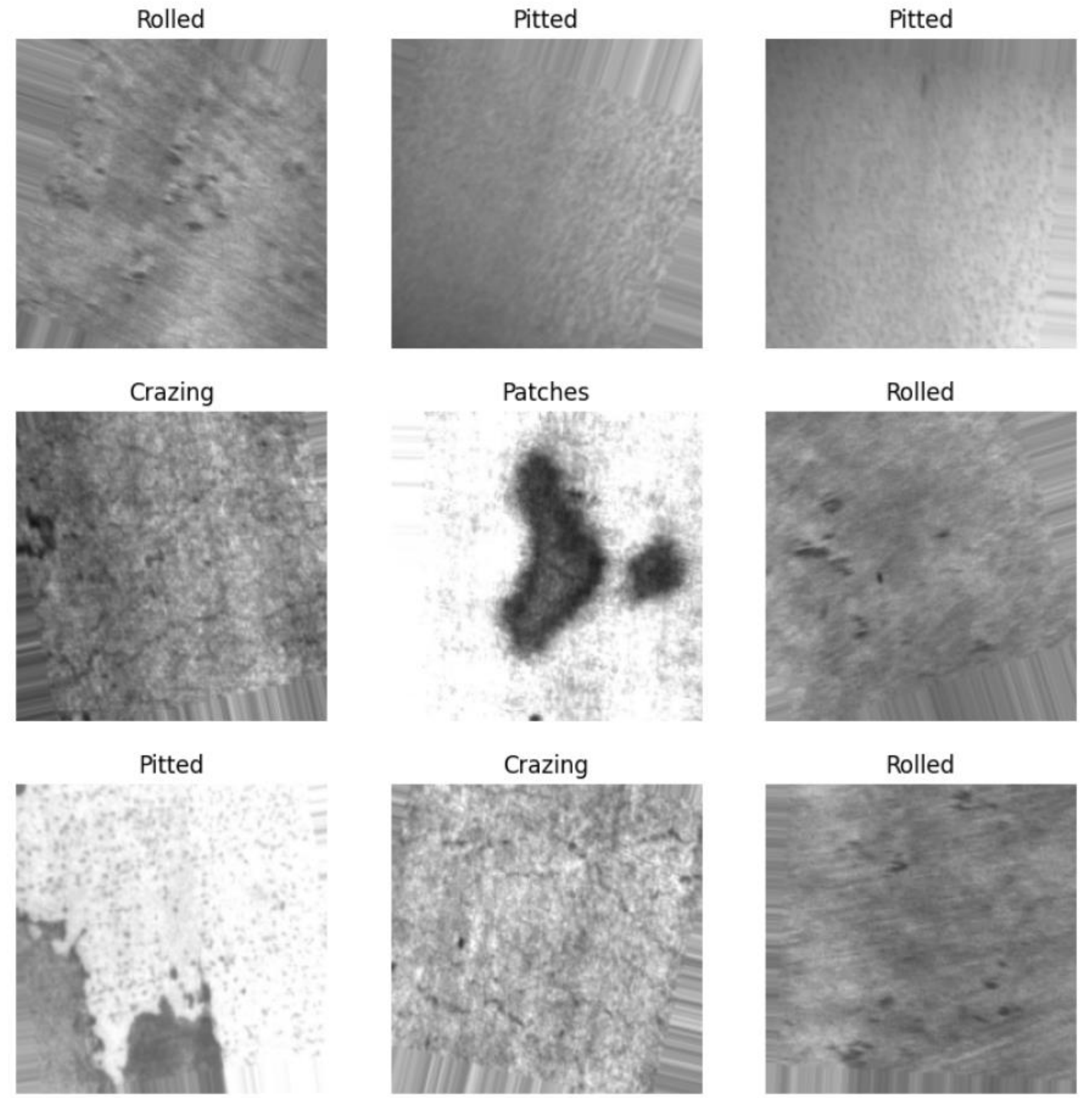
Report on NEU-DET Steel Surface Defect Detection

1.Introduction

The objective of the project is to build a Tensorflow Deep Learning model which can identify defect location on steel surface. The model used here is MobileNetV2 with. For this we have used the open surface defect database from Northeastern University(NEU).

In this database, six kinds of typical surface defects of the hot-rolled steel strip are collected, i.e., rolled-in scale (RS), patches (Pa), crazing (Cr), pitted surface (PS), inclusion (In) and scratches (Sc). The images provided in the database are in grayscale and of uniform dimensions 200x200. A snapshot of how the images looks like are shown below:-



The database can be found in the below links:-

http://faculty.neu.edu.cn/yunhyan/NEU_surface_defect_database.html

https://drive.google.com/drive/folders/1B4NnQb8AXvbKVC70mQyq6jSYU-LmupYh?usp=sharing

The description of the database in the above url states that the database includes 1,800 grayscale images: 300 samples each of six different kinds of typical surface defects.

2.Requirements

- Software Requirements

Google Colab/ Jupyter Notebook

Numpy

Pandas

Matplotlib

Tensorflow

- Hardware Requirements

GPU

3.Methodology

A] Data Loading:

We load data from out Google Drive into three main sets :- Training Data, Testing Data, Validation Data.

B] Data Preprocessing:

For data preprocessing we use ImageDataGenerator from tensorflow.keras.preprocessing.image module.

We create two datagen: train_datagen, test_datagen.

We apply the following preprocessing on train_datagen:

- `rescale=1./255`
- `rotation_range=20`
- `width_shift_range=0.1`

```
height_shift_range=0.1
horizontal_flip=True
```

On test_datagen, we only rescale to 1./255

Then using the flow_from_directory method, we load our images from out directories and apply the preprocessing and divide the data into batches of 32 and fix the image size as (200,200).

C] Modelling:

We create 4 models with the model_summaries as shown below.

- **Model 1**

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 199, 199, 32)      416

 max_pooling2d (MaxPooling2  (None, 99, 99, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 98, 98, 64)        8256

 max_pooling2d_1 (MaxPoolin  (None, 49, 49, 64)        0
 g2D)

 conv2d_2 (Conv2D)           (None, 48, 48, 128)       32896

 max_pooling2d_2 (MaxPoolin  (None, 24, 24, 128)       0
 g2D)

 flatten (Flatten)           (None, 73728)             0

 dense (Dense)               (None, 256)               18874624

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 6)                 1542

=================================================================
Total params: 18917734 (72.17 MB)
Trainable params: 18917734 (72.17 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- **Model 2**

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 199, 199, 32)      416

 max_pooling2d_3 (MaxPoolin  (None, 99, 99, 32)        0
 g2D)

 conv2d_4 (Conv2D)           (None, 98, 98, 64)        8256

 max_pooling2d_4 (MaxPoolin  (None, 49, 49, 64)        0
 g2D)

 conv2d_5 (Conv2D)           (None, 48, 48, 128)       32896

 max_pooling2d_5 (MaxPoolin  (None, 24, 24, 128)       0
 g2D)

 flatten_1 (Flatten)         (None, 73728)             0

 dense_2 (Dense)             (None, 256)               18874624

 dropout_1 (Dropout)         (None, 256)               0

 dense_3 (Dense)             (None, 6)                 1542

=================================================================
Total params: 18917734 (72.17 MB)
Trainable params: 18917734 (72.17 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

- **MobileNetV2 Model**

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 mobilenetv2_1.00_224 (Func  (None, 7, 7, 1280)        2257984
 tional)

 global_average_pooling2d (  (None, 1280)              0
 GlobalAveragePooling2D)

 dense_4 (Dense)             (None, 256)               327936

 dropout_2 (Dropout)         (None, 256)               0

 dense_5 (Dense)             (None, 6)                 1542

=================================================================
Total params: 2587462 (9.87 MB)
Trainable params: 329478 (1.26 MB)
Non-trainable params: 2257984 (8.61 MB)
```

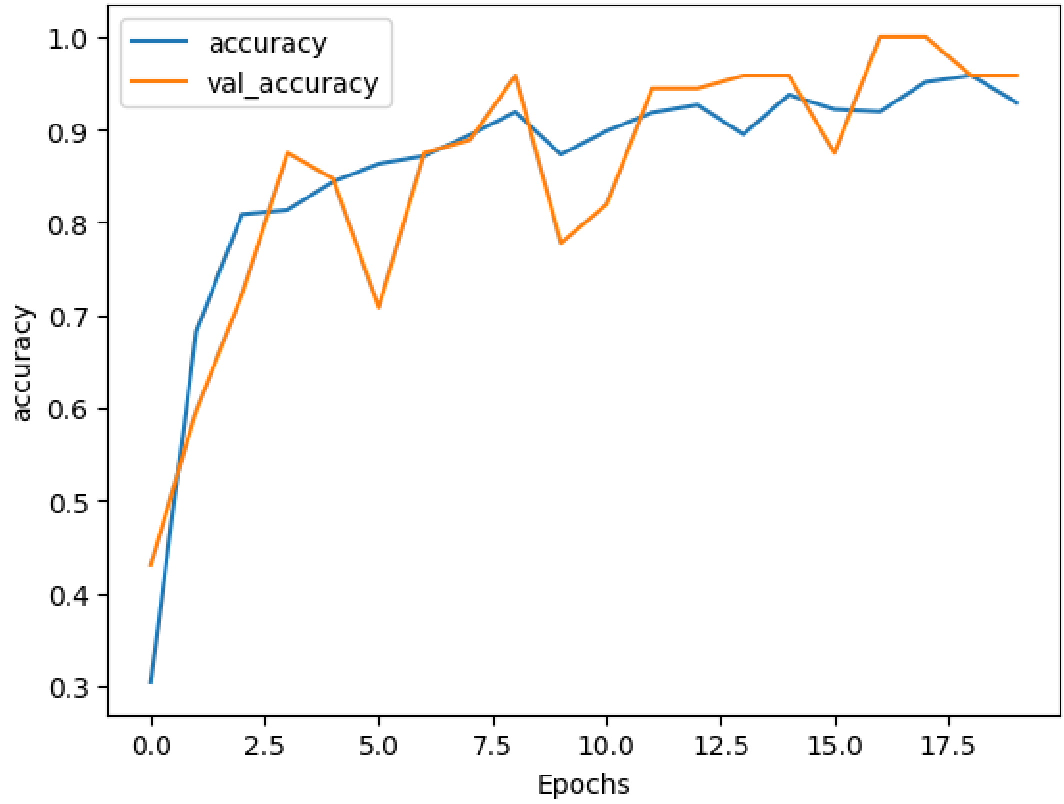
We are using Transfer Learning here and we have customised the model according to out required output with keeping only few layers trainable.

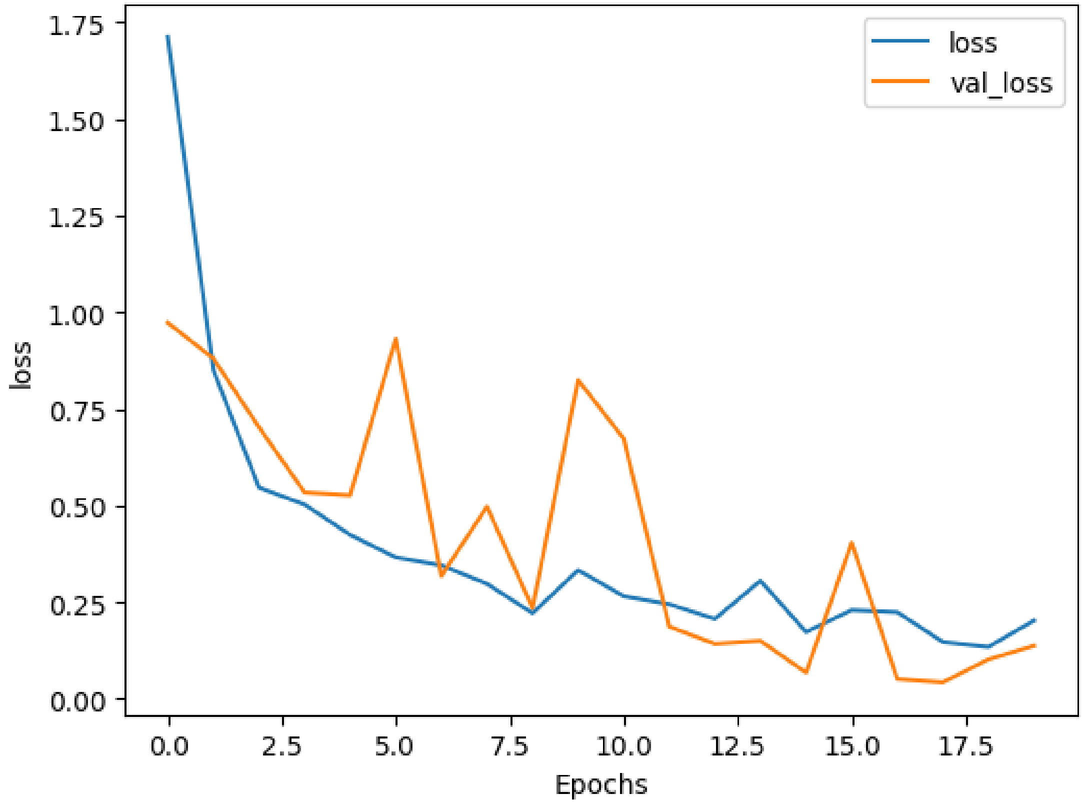
- **EfficientNetB1 Model**

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 efficientnetb1 (Functional  (None, 7, 7, 1280)        6575239
 )

 global_average_pooling2d_2  (None, 1280)              0
  (GlobalAveragePooling2D)

 dense_8 (Dense)             (None, 256)               327936

 dropout_4 (Dropout)         (None, 256)               0

 dense_9 (Dense)             (None, 6)                 1542

=================================================================
Total params: 6904717 (26.34 MB)
Trainable params: 329478 (1.26 MB)
Non-trainable params: 6575239 (25.08 MB)
_________________________________________________________________
```

4.Model Performance

- **Model 1:**

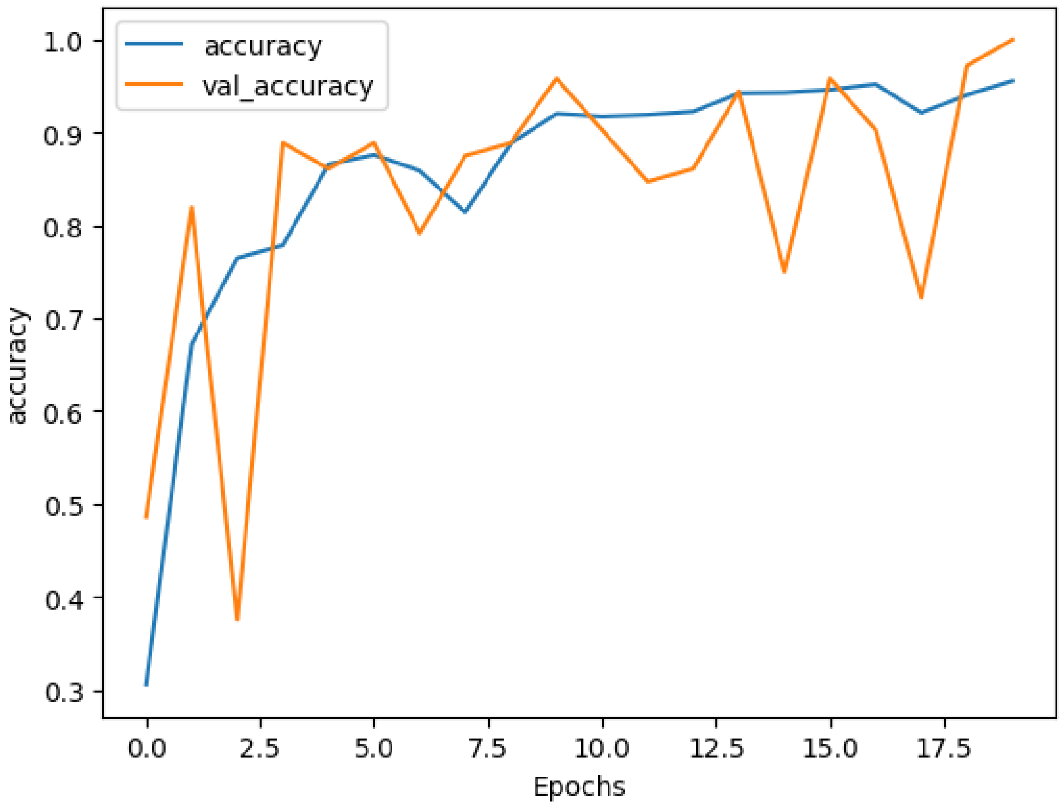


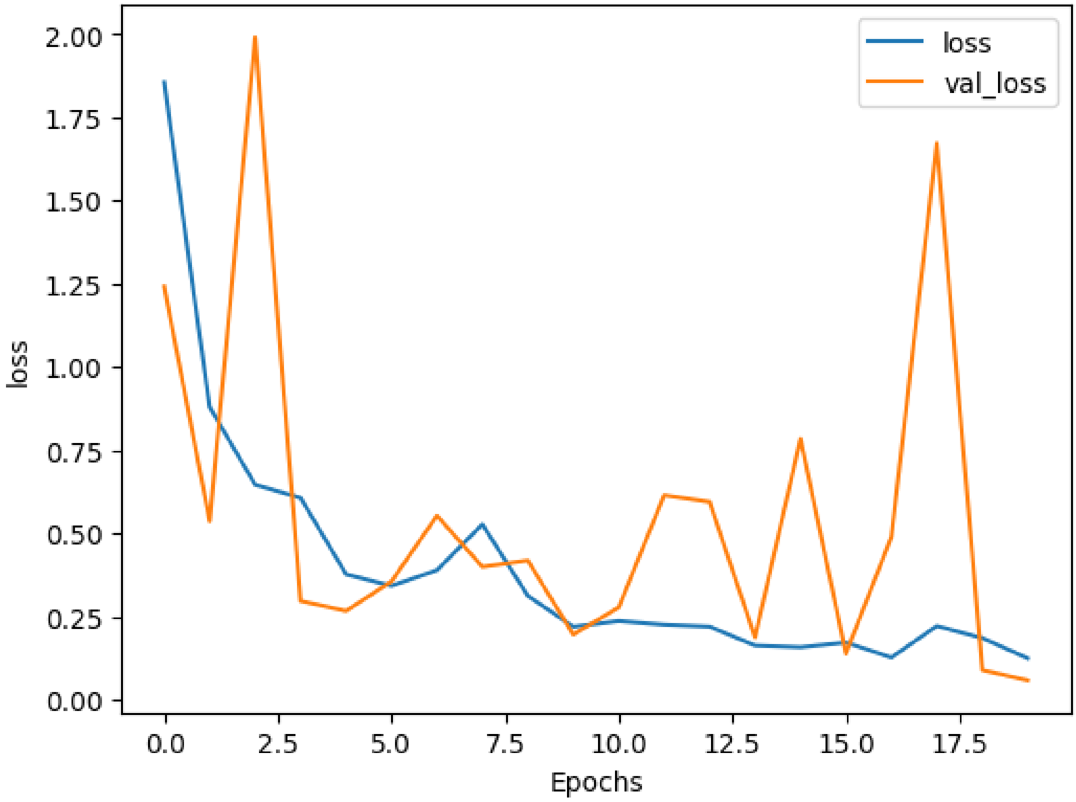


```
[ ]  result1 = model1.evaluate(test_generator)
     print("Test loss, Test accuracy : ", result1)
```

```
3/3 [==============================] - 0s 71ms/step - loss: 0.3064 - accuracy: 0.8611
Test loss, Test accuracy :  [0.3064206540584564, 0.8611111044883728]
```

- **Model 2:**

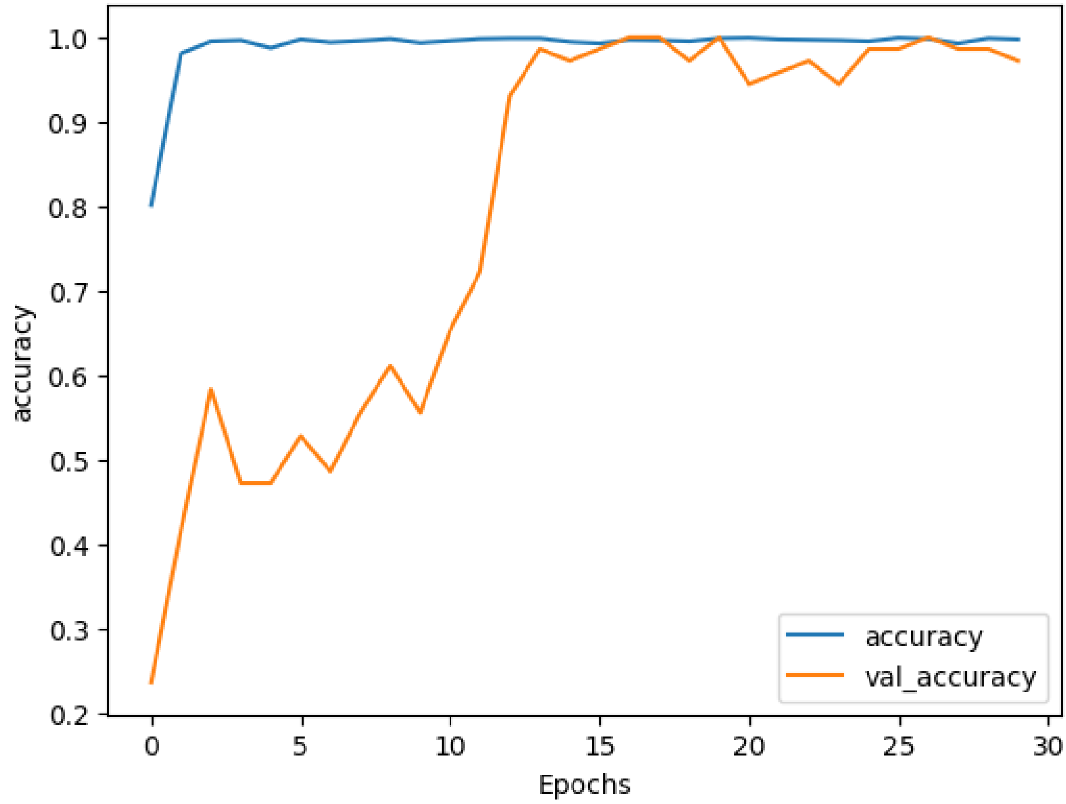


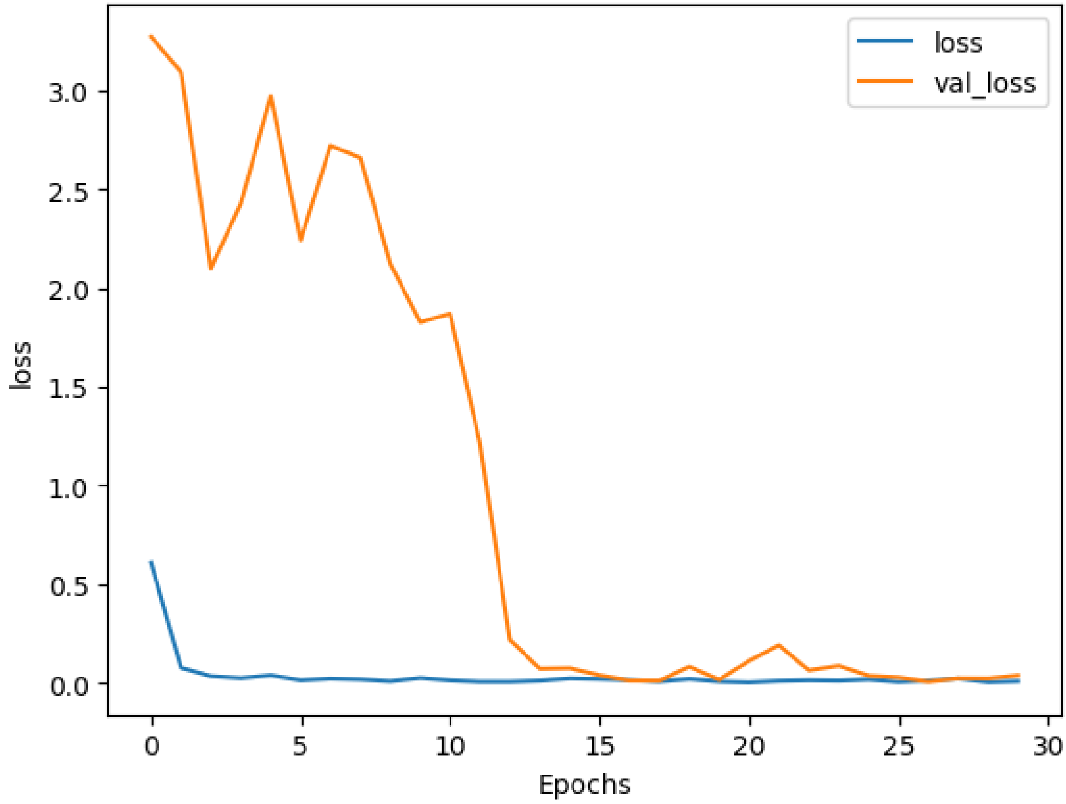



```
result2 = model2.evaluate(test_generator)
print("Test loss, Test accuracy : ", result2)
```

```
3/3 [==============================] - 1s 133ms/step - loss: 0.3472 - accuracy: 0.8750
Test loss, Test accuracy :  [0.3471944034099579, 0.875]
```

- **MobileNetV2 Model:**

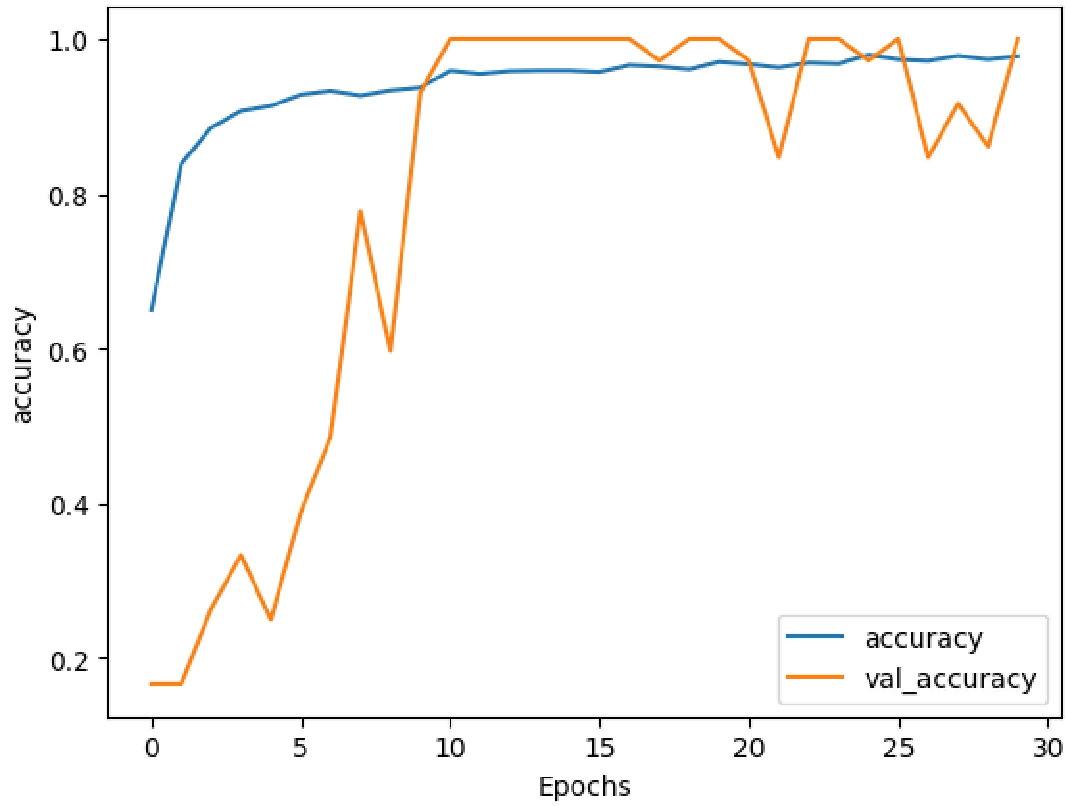


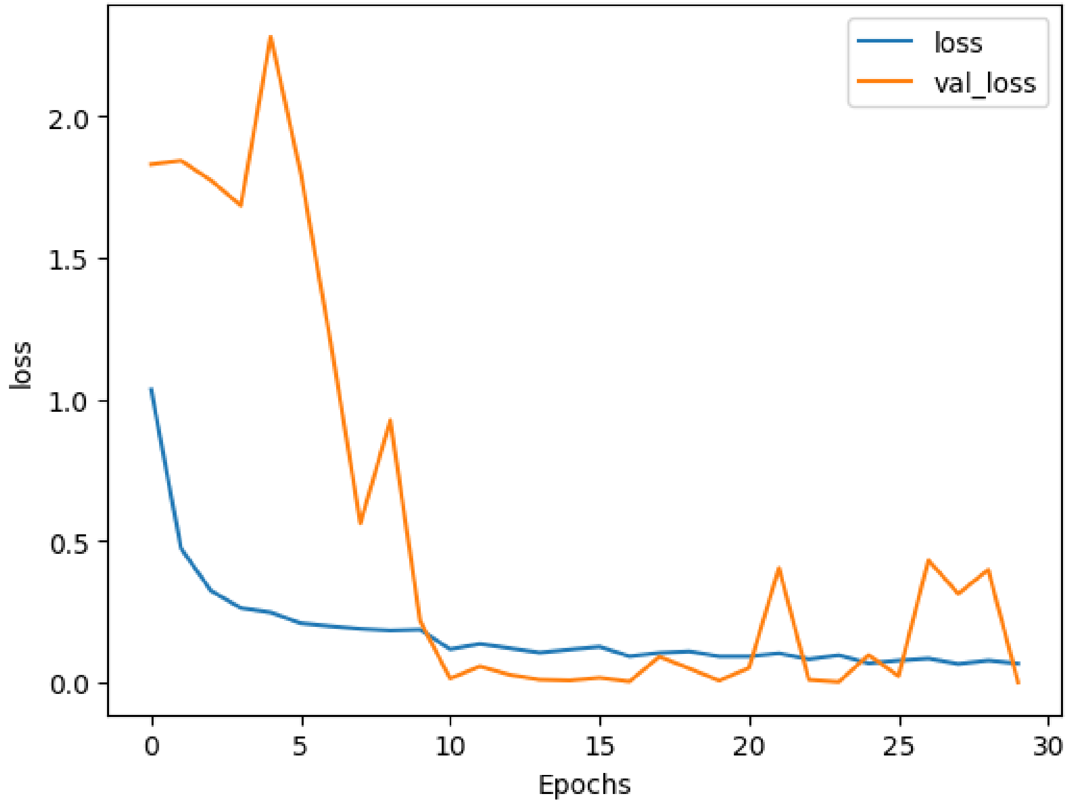



```
result_mobilenet = model_mobilenet.evaluate(test_generator)
print("Test loss, Test accuracy : ", result_mobilenet)
```

```
3/3 [==============================] - 0s 109ms/step - loss: 0.0761 - accuracy: 0.9861
Test loss, Test accuracy :  [0.0761433020234108, 0.9861111044883728]
```

- **EfficientNetB1 Model:**






```
result_efficient = model_efficient.evaluate(test_generator)
print("Test loss, Test accuracy : ", result_efficient)
```

```
3/3 [==============================] - 1s 227ms/step - loss: 5.5828e-04 - accuracy: 1.0000
Test loss, Test accuracy :  [0.0005582821322605014, 1.0]
```

Comparing between different models, we get:

	Model	Accuracy	Loss
0	Model 1	0.929214	0.202334
1	Model 2	0.955609	0.124339
2	MobileNetV2	0.997600	0.007912
3	EfficientNetB1	0.977804	0.066639

From this, we can infer that MobileNetV2 works best for our objective.
We get maximum accuracy with minimum loss in case of MobileNetV2 so we will use it for our operations.

-------------------Thank You------------------