TRANSFER LEARNING BASED CONCRETE CRACK DETECTION

(Through use of the EfficientNetB0)

EEE 312

A2 Group 1

Member IDs: 1706034-37

Introduction

In the concrete jungle that is our modern infrastructure, it goes without saying that crack detection is of utmost importance for the safety of people and their wellbeing. The current method of detection involves a specified technician or engineer working with related paraphernalia to detect related problems. This is costly, time consuming, and the use of instruments can bring about further problems or hazards.

Considering the great boom of avenues being generated that uses machine learning and image processing, it can be rightly assumed that a smoother, hassle free method of crack detection can be found from this sort of learning. Human neuron like neural networks can efficiently detect cracks of which photos are taken. There are already feature vector extraction software that use deep learning, and they have yielded good results. But the complexity and diversity involved in the nature of cracks and their detection make this field a constantly improving and ongoing field. In our project, we have tried to emulate a paper that has been already published on the matter by *Chao Su and Wenjun Wang et al.* We have tried our very own methods and coding, with alterations here and there to the spirit of the model described in the paper, to come up with something that is superior to what we set a baseline for our goal.

A brief literature review of the tasks performed in this field:

Convolutional neural networks, sharing both the prejudices of connectivity and local parameter sharing, are considered a steeple in large scale image processing tasks. The road was paved by Zhang et al, who applied CNN to road crack detection photos taken by smartphones. Utilizing augmentation within the training and test dataset, it was possible to be shown by Pauly et al. that there is a positive correlation between depth of network and accuracy. Later, Xu et al. formulated an end to end bridge crack detection model with greatly reduced number of parameters, achieving a whopping 96.37% accuracy. But probably the greatest breakthrough was done by Yang, who could incorporate pixel level detection on a full CNN with 97.96% accuracy and a precision of 81.73%. As it is very time consuming to train a model from the ground up, **transfer learning** was incorporated into this field, rendering certain core layers untrainable, used by Zhu and Song, and also by the paper we tried to emulate: who used transfer learning on EfficientNetB0 to achieve their means.

In our project we tried to improve on the EfficientNetB0 model by progressing using our very own (and emulated from other sources) algorithms, and we compared it to other models we implemented to achieve the same means, which are, InceptionV3 and MobileNetV2 models.

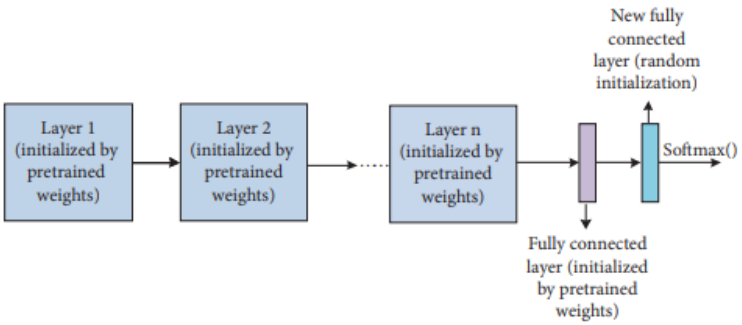


Figure: A block based explanation of transfer learning

About CNNs in general and EfficientnetB0:

CNNs, when used in image processing, incorporate input data from input layer using techniques like convolution operation, pooling, and non-linear activation mapping. Through the proper use of its input, convolution, activation, pooling, fully connected and Softmax layer, one will be able to achieve meaningful predictions regarding the data.

The EfficientNetB0 is a network that uses the smoothened ReLU like Swish activation function. The function basically is defined as

$f(x) = x.sigmoid(x)$

and its effectiveness has been ascertained through its use in some large neural networks.

The EfficientNet implements a lower parameter count and higher accuracy and boasts of great feature extraction abilities. The basic actions happening in the crannies of the layers are simple: a 3x3 convolution is done on input image, with inverted bottleneck convolution modules wrenching further features off the input. After a 1x1 convolution and average pooling, results apparate to the connected layer. The convolution functions are always followed by a batch normalization, and Swish activation is there to ensure efficient activation. MBconv handles the batch normalization and convolution, and a squeeze and activation block handles the pooling and the 1x1 convolution. The work processes are summarized through block diagrams.

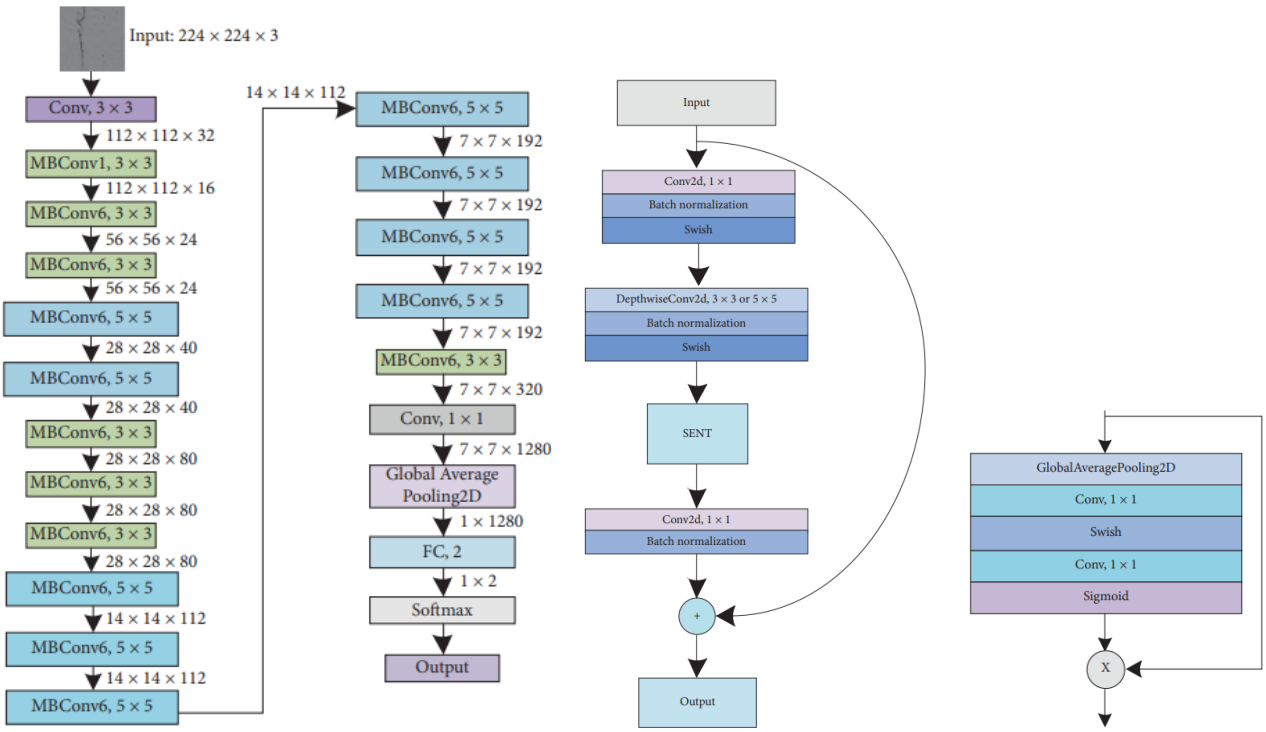


Figure: EfficientNet Architecture (left), MBconv framework (middle) and Squeeze and excitation block (right).

Processes we followed and some background regarding them:

Dataset:

We collected data from mainly two sources (links are given at the last of the report). The first dataset (*Mendeley)* contains 40000 images that are 227x227 pixels in dimension, broken into negative and positive folders. This makes up our initial dataset, and we formulate this into training, validation and test set in ratio of 6:2:2. Another source from Utah University consists of 7700 images, which we added to the initial dataset to make the *MendeleySDadded dataset*, broke into 6:2:2 ratio. For model convenience and to maintain similarity with our followed paper, we converted the images into 224x224 pixels and the breaking into dataset types had been handled by the amalgamation of Google Colaboratory, tensorflow built in codes and google drive.

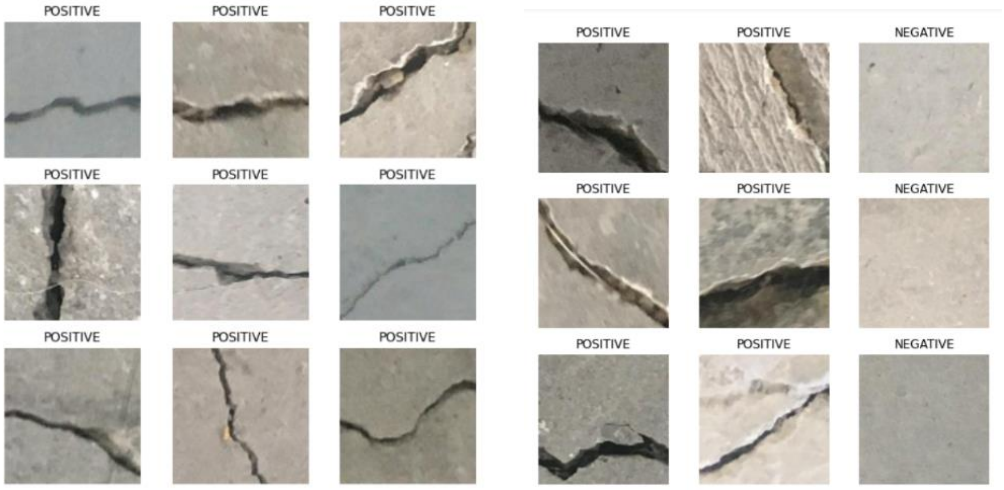


Figure: Samples of Mendeley Dataset (left) and MendeleySDadded dataset (right)

Assigning hyperparameters and data augmentations:

We import the necessary libraries, like keras and tensorflow to enjoy the privilege of built in codes. Adam optimizer is also imported.

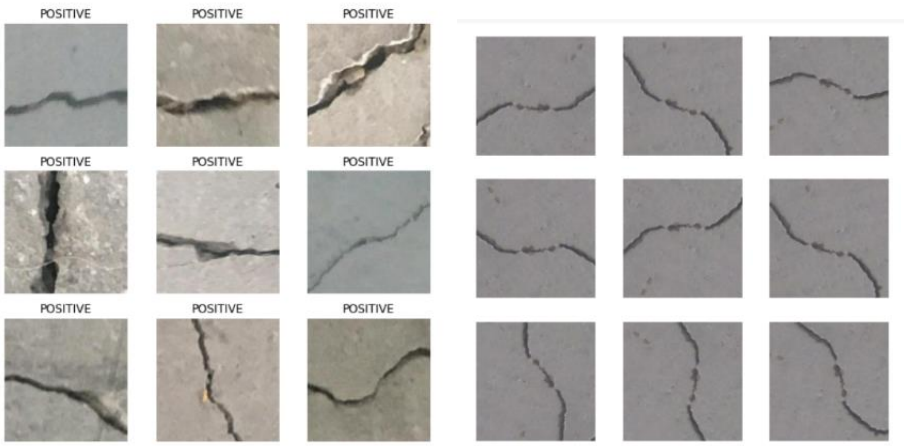
The dataset directories are defined, and the datasets are assigned to variables.

Hyperparameters are assigned. Learning rate, which is a proportion of how much the model will change in response to the difference between predicted value and the actual, is set as 0.0001. Batch size of the images are taken to be 32.

Preprocessing and augmentation of the data includes random horizontal and vertical flipping.

We visualize some data before and after augmentation.

Before: *After:*



Afterwards, auto tuning helps us to efficiently use the input pipeline and model pipeline and prefeteching the data helps us with just that. We do this so that none of the input fetching and modelling process has to stay idle for bottlenecking from one another.

FREEZING LAYERS AND PREPROCESSING:

We issue the model.trainable false command to freeze up the pre-trained EfficientNet model layers, and add further layer for learning purposes. This is important in transfer learning as we do not want to build everything from ground up, rather add on to the layers that have already been trained beforehand of importing.

The last layer, that is, the prediction layer, has been assigned an activation of the sigmoid function nature.

The preprocessing commands that we apply on input adequate the input data so that it becomes fully eligible for the model we are using, that is EfficientNet.

Dropout rates, pooling, and Optimizer:

Dropout is a regularization method that approximates training a large number of neural networks with different architectures in parallel. During training, some number of layer outputs are randomly ignored or dropped out. This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different view of the configured layer. We added a dropout rate of 0.3 for our modelling purposes.

We also implement global average pooling 2d. Global Average Pooling is a pooling operation designed to replace fully connected layers in classical CNNs. The idea is to generate one feature map for each corresponding category of the classification task in the last mlpconv layer. Instead of adding fully connected layers on top of the feature maps, we take the average of each feature map, and the resulting vector is fed directly into the softmax layer.

In addition to this, we implement the Adam optimizer. Optimizers optimize the learning rates and weights in layers to minimize the loss. Adam, or adaptive moment estimation, does a slow search when it nears the minimum. It keeps storage of an exponentially decaying average of past squared gradients and also of past gradients. The first and second order momentums are taken into consideration.

After all the settings are complete, we initiate the training.

Fine tuning as an incremental step:

We had thought of 30 epochs for our training. After 15 epochs, until which the model has trained to convergence safely, we decide to change the model a little bit, that is, unfreeze the layers and run the last 15 epochs with unfrozen layers. This is the fine tuning step. We implement the layers.trainable true command, and we drop the learning rate by a factor of 10. This step has been known to give incremental boosts to models (with a risk of overfitting at times) but we stay cautious.

Performance testing

The training is allowed to complete, once for the initial Mendeley (initial) dataset, and once for the SDnet added (mixed) dataset. The model evaluation command does evaluation on the test dataset and finds out related parameters.

The parameters we are finding out are:

Accuracy: (TP+TN)/ (TP+FP+TN+FN)

Precision: TP/ (TP+FP)

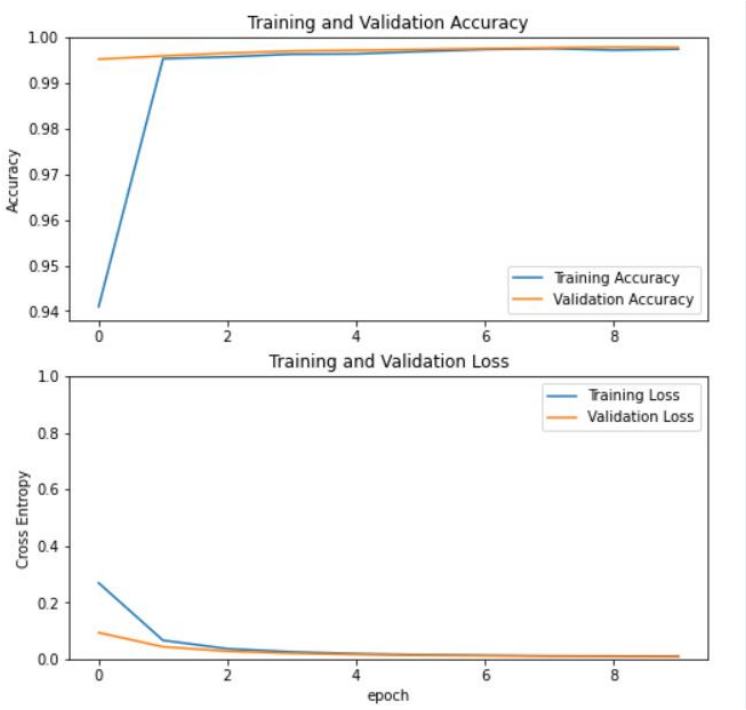
Recall: TP/ (TP+FN)

F-score: 2 x precision x recall/ (precision + recall)

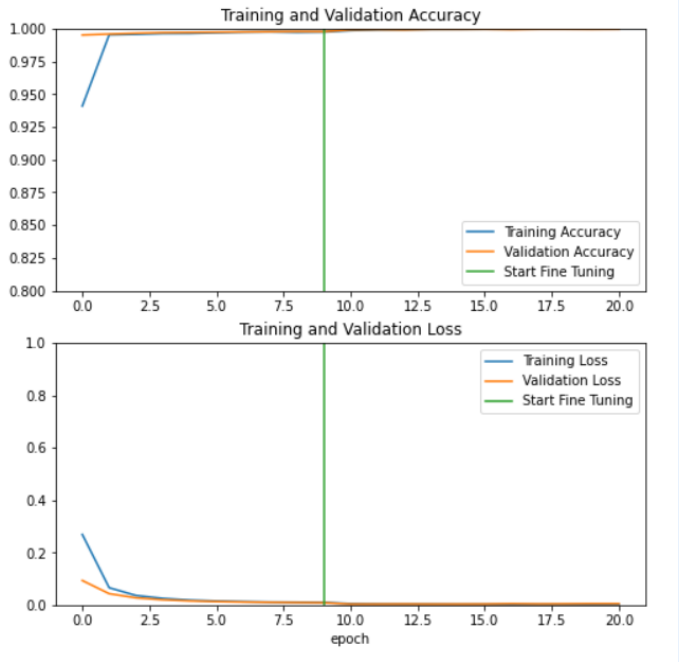
Where TP and TN mean correctly classified images with crack and without crack.

Results:

For *Mendeley* dataset:



After fine tuning:

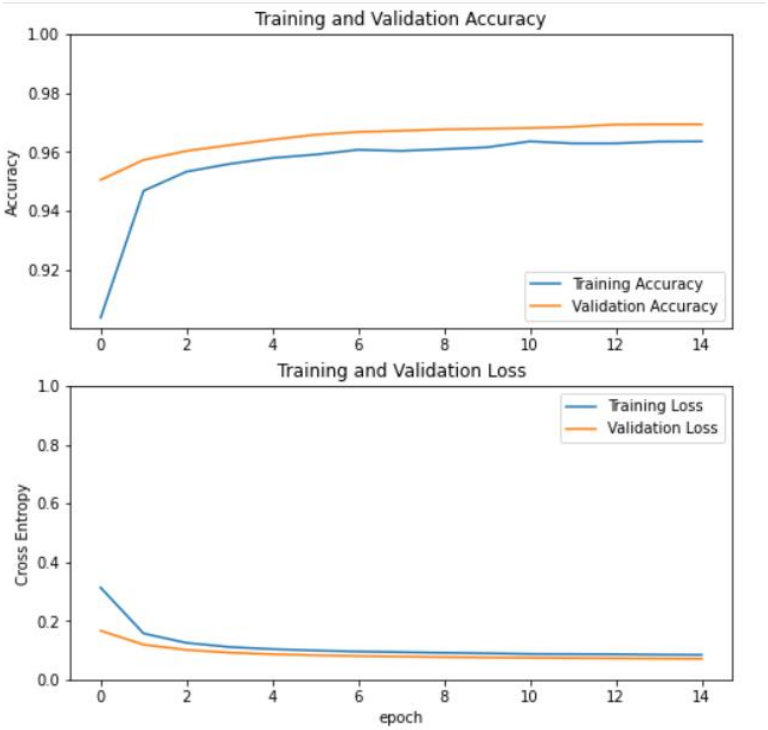


```
Test accuracy : 0.9995518326759338
Test precision : 0.9994412660598755
Test recall : 0.9998602867126465
```

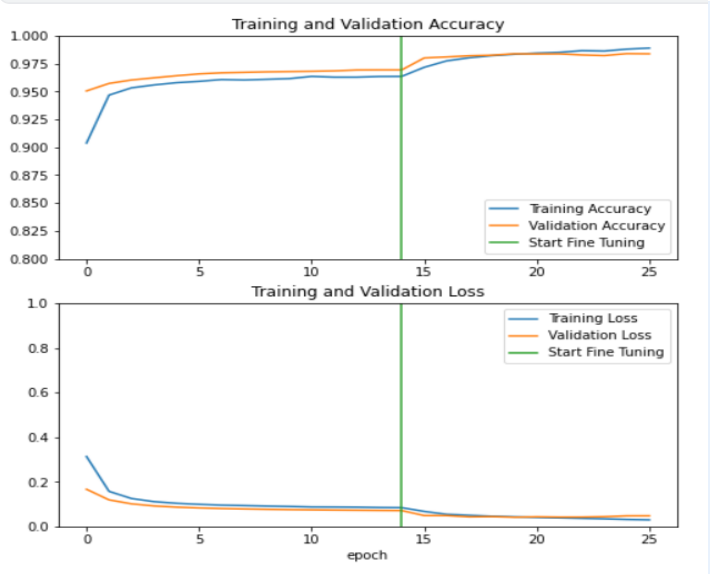
```
F1 Score :  0.9870558478284374
```

For *MendeleySDadded* dataset

Before fine tuning stage:



After fine tuning:



```
_________________________________________________________________
Total params: 4,050,852
Trainable params: 1,281
Non-trainable params: 4,049,571
_________________________________________________________________
```

```
Test accuracy : 0.9839319586753845
Test precision : 0.9928516745567322
Test recall : 0.9813272953033447
```

```
F1 Score :  0.9870558478284374
```

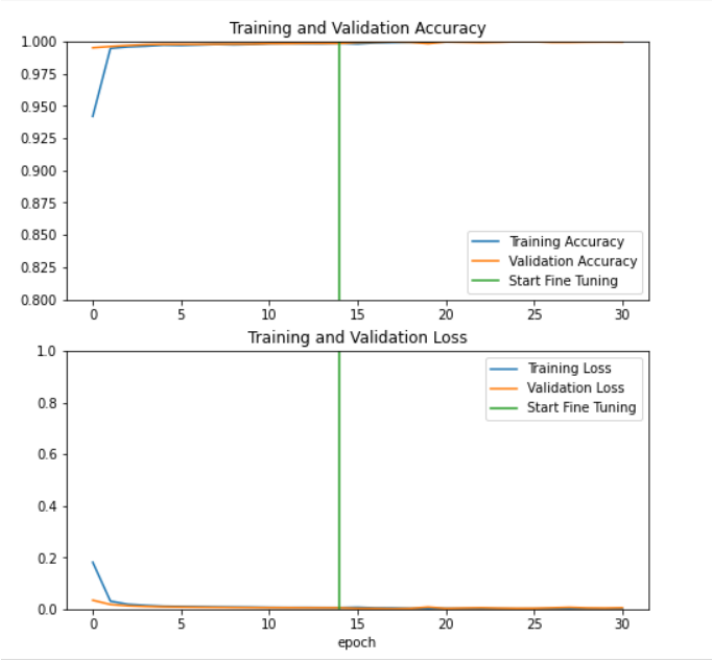
In addition to these results, we deemed that it was required, for a basis of comparison, how the EfficientNet is holding up against the other networks. We chose inception v3 and mobilenet v2 for our purposes. Both inception v3 and mobilenet have been known to be excellent models for image processing tasks

MobilenetV2 for *Mendeley* dataset:

Parameter count:

```
=================================================================
Total params: 2,259,265
Trainable params: 1,862,721
Non-trainable params: 396,544
```

Training and validation accuracy and losses:



Results:

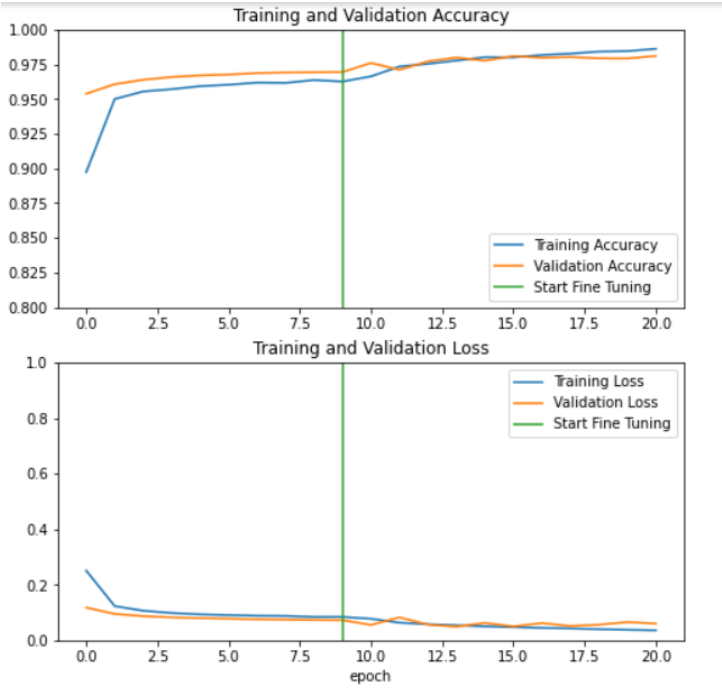
```
loss, accuracy, precision, recall = model.evaluate(TEST_IT)
print('Test accuracy :', accuracy)
print('Test precision :', precision)
print('Test recall :', recall)
```

```
349/349 [==============================] - 928s 3s/step - loss: 0.0054 - accuracy: 0.9996 - precision: 0.9993 - recall: 1.0000
Test accuracy : 0.9995518326759338
Test precision : 0.9993017911911011
Test recall : 1.0
```

```
f1_score = (2*precision*recall)/(precision+recall)
print('F1 Score : ', f1_score)
```

```
F1 Score :  0.9996507736791037
```

Mobilenet V2 on *MendeleySDadded* dataset:



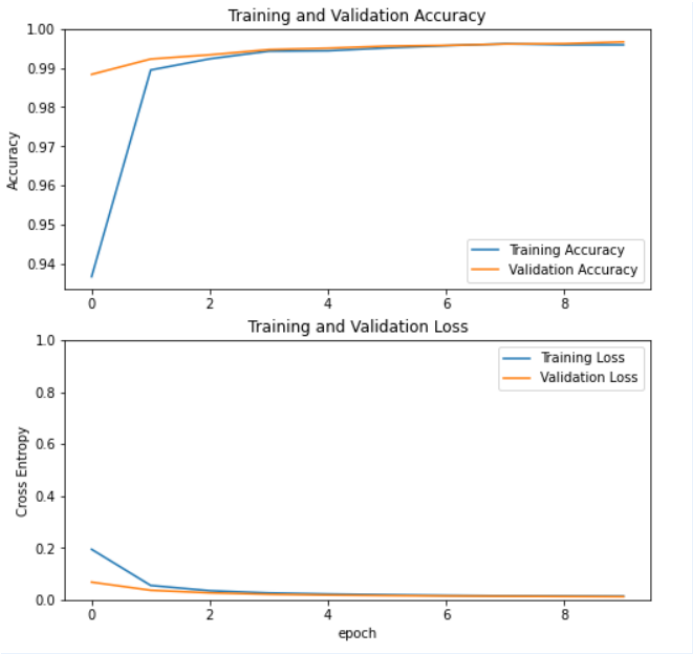

```
loss, accuracy, precision, recall = model.evaluate(TEST_IT)
print('Test accuracy :', accuracy)
print('Test precision :', precision)
print('Test recall :', recall)
```

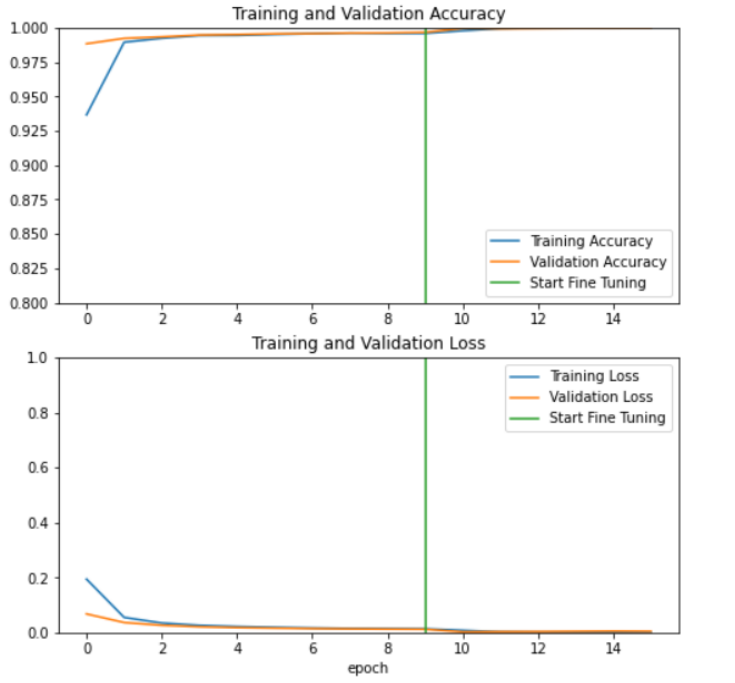
```
386/386 [==============================] - 1950s 5s/step - loss: 0.0607 - accuracy: 0.9798 - precision: 0.9915 - recall: 0.9769
Test accuracy : 0.9798070192337036
Test precision : 0.9915482401847839
Test recall : 0.9769114255905151
```

```
f1_score = (2*precision*recall)/(precision+recall)
print('F1 Score : ', f1_score)
```

```
F1 Score :  0.9841754156330845
```

InceptionV3 for *Mendeley* dataset:

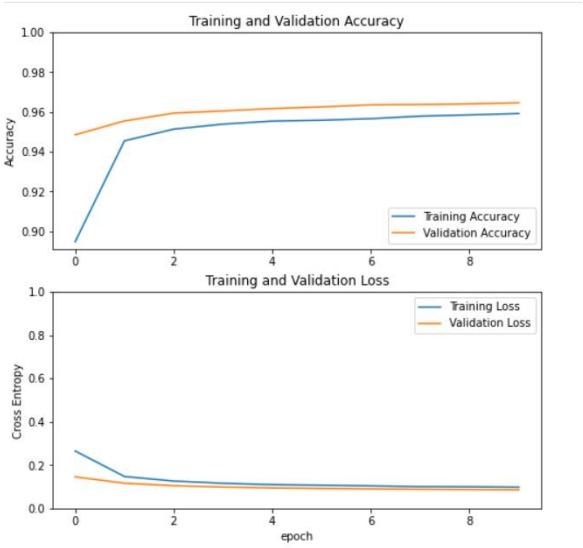


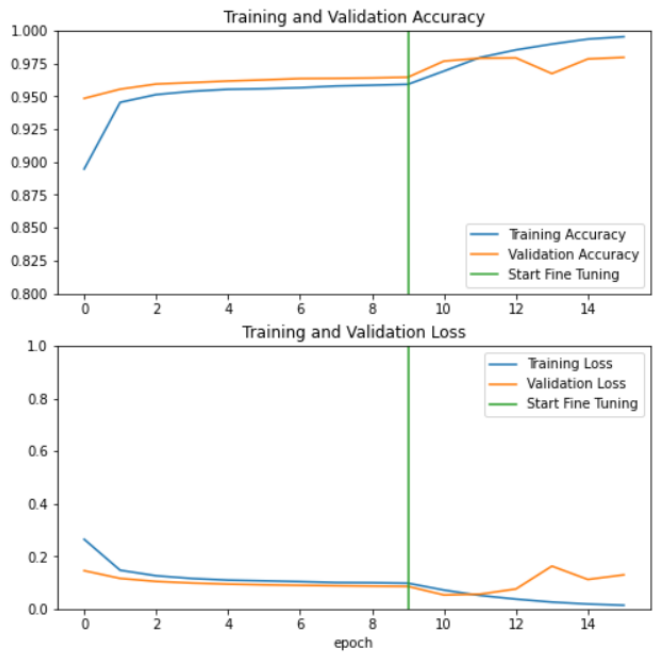



```
Test accuracy : 0.9998207092285156
Test precision : 0.9998602867126465
Test recall : 0.9998602867126465
```

```
F1 Score :  0.9998602867126465
```

InceptionV3 for *MendeleySDadded* dataset:






```
Test accuracy : 0.981647789478302
Test precision : 0.9921944737434387
Test recall : 0.9782992601394653

 F1 Score :  0.9851978749152963
```

For *Mendeley* dataset (Low dataset difficulty and easier discernible recognition possible)

Recognition model	Accuracy	Precision	Recall	F1 score
EfficientNetB0	0.99955	0.99944	0.99986	0.98706
MobileNetV2	0.99955	0.99930	0.99999	0.99965
InceptionV3	0.99982	0.99986	0.99986	0.99986

For *MendeleySDadded* dataset (a more difficult batch has been mixed into initial dataset)

Recognition model	Accuracy	Precision	Recall	F1 score
EfficientNetB0	0.98393	0.99285	0.98133	0.98706
MobileNetV2	0.98220	0.99334	0.97805	0.98563
InceptionV3	0.98165	0.99219	0.97830	0.98519

Model	Epoch count	Parameter count
EfficientNetB0	15+15	4,050,852
MobileNetV2	15+15	2,259,265
InceptionV3	15+15	21,802,784

Conclusion:

The paper we were trying to emulate and surpass had an accuracy on mixed dataset testing of 97.37%. Our use of the same model yielded an accuracy of 98.4%, more than a 1% rise. In addition, we used other networks for comparison and came to realize that in *Mendeley* datasets (easy) EfficientNet is not the best in terms of parameters as the Inception and MobileNet beat it largely. But the finesse of the EfficientNet is spectacled when it comes to varieties of data, and it shines out more accurately than other models. It also holds a good balance between precision, recall and the F1 scores. We think that this model is fairing pretty well in its current state, but considering the time constraints of a project work, it has not been possible to modify it further: like its avenues in analyzing the severity and magnitude of the crack, which we want to work on in the future.

Links we have used in writing the report:

1. https://www.tensorflow.org/guide/keras/transfer_learning

2. https://www.tensorflow.org/api_docs/python/tf/keras/applications/efficientnet/EfficientNetB0

3. https://www.kaggle.com/micajoumathematics/fine-tuning-efficientnetb0-on-cifar-100

4.https://www.tensorflow.org/api_docs/python/tf/keras/metrics?fbclid=IwAR1j_BjCKJcIJ395k2I7TX98KPfOgadX1Li-SAdibHZWIKGdrG2ctgKL0R4

5.https://stackoverflow.com/questions/43345909/when-using-mectrics-in-model-compile-in-keras-report-valueerror-unknown-metr

6. https://towardsdatascience.com/an-in-depth-efficientnet-tutorial-using-tensorflow-how-to-use-efficientnet-on-a-custom-dataset-1cab0997f65c

7.https://colab.research.google.com/github/keras-team/keras-io/blob/master/examples/vision/ipynb/image_classification_efficientnet_fine_tuning.ipynb

8. https://learnopencv.com/efficientnet-theory-code/

Dataset links:

https://data.mendeley.com/datasets/5y9wdsg2zt/2

https://digitalcommons.usu.edu/all_datasets/48/