Facial non-threatening skin disease detection using Convolution Neural Networks.

Team members

Komal Jaiswal(kq2587)
Ponmalar Rajan(zy5227)
Bhargavi Sankula(qv4668)
Madhumitha Munirathinam(br8494)

CONTENTS

1. Narrative description of our project and Problem description

We may come across many reddit posts or questions in online forums constantly asking for advice related to facial skin conditions, also many of us might have experienced some kind of non-threatening skin condition like Acne which at often times can be cured with simple home remedies and doesn't really need any professional medical diagnosis.

Our project is aimed to help us identify such non-threatening skin disorders without going to a skin clinic or consulting a dermatologist.

With future scope of developing into an app which scans your face and helps users by instantly detecting the type of skin condition and also offers advice , our project's objective is to build a good architecture for Facial skin disease detection using Keras and Dense convolution Neural network.

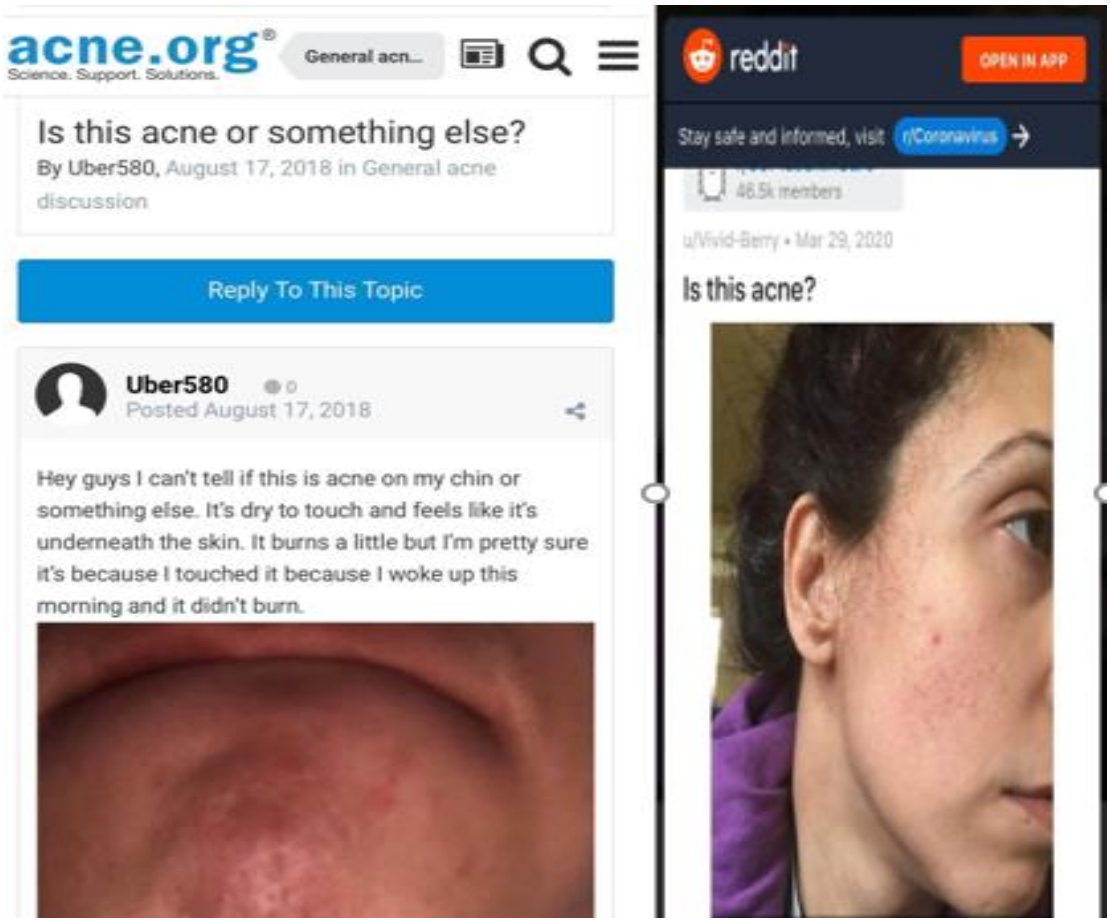

Figure 1-1 snapshots of people looking for advice on what type of skin condition they might possibly have in online forums

2. Data description

When we first came with this idea of facial skin disease detection model, we were looking for active Kaggle or any existing dataset, but we could not find any online which possibly means this area of study does not have much active projects. Affirmative, our idea is fresh.

2.1. Building our own dataset

With no dataset readily available online, we had to custom build our own dataset. We finalized on five types of skin diseases which forms five classes or categories of our project and they are Acne, Rosacea, Eczema, Melasma, Hives. And on careful study and research these five categories of skin disease are confirmed to be non-life threatening that does not always need medical help or any immediate medical diagnosis.

Figure 2-1 types of non-threatening skin diseases that forms five classes of our project

2.2. Data preprocessing and cleaning

Using chrome extension, we downloaded approx. 450 images per class so total of approx. 2250 images in total. But many images were not clear and watermarked, so we had to manually clean the datasets since the images are all taken from internet .

After cleaning, we got the following number of images per each class, on total the sample size is 1462 images collectively, which we thought was a decent sample size for our dataset. Also, by thorough manual cleaning we made it fit for our model building.

- *ACNE (283 images)*
- *ECZEMA (267 images)*
- *MELASMA (333 images)*
- *HIVES (273 images)*
- *ROSACEA (306 images)*

Our goal is to train a Convolutional Neural Network using Keras and deep learning to recognize and classify each of these Facial skin diseases. Out of 1462 images, training and validation set was split on 80-20 by random sampling method. We also randomized our training dataset to prevent any bias during the training and to prevent the model from learning the order of the training for better efficiency.

```
[ ]  # Randomize the order of training set
     SEED = 42
     train = train.sample(frac=1, random_state=SEED)
     train.index = np.arange(len(train)) # Reset indices
     train.head()
```

	File	DiseaseId	Disease Type
0	Rosacea/images11.jpg	2	Rosacea
1	Hives/images202.jpg	3	Hives
2	Melasma/images278.jpg	1	Melasma
3	Melasma/images116.jpg	1	Melasma
4	Hives/images697.jpg	3	Hives

Figure 2-2 showing how we randomized our training dataset for better model building

3. Base Model CNN from scratch

3.1. Convolution Neural Network(CNN) from scratch description

Now that our dataset is cleaned and preprocessed, we built a Convolutional Neural Network (CNN) from scratch. CNN is go-to model on every image related problem. Since the main advantage of CNN compared to its predecessors is that it automatically detects the important features without any human supervision, CNN model built from scratch is our base model.

Our input images are randomized training images and we performed a series convolution and pooling operations, followed by several fully connected layers. Since we are doing a multiclass classification the output function used is Softmax.

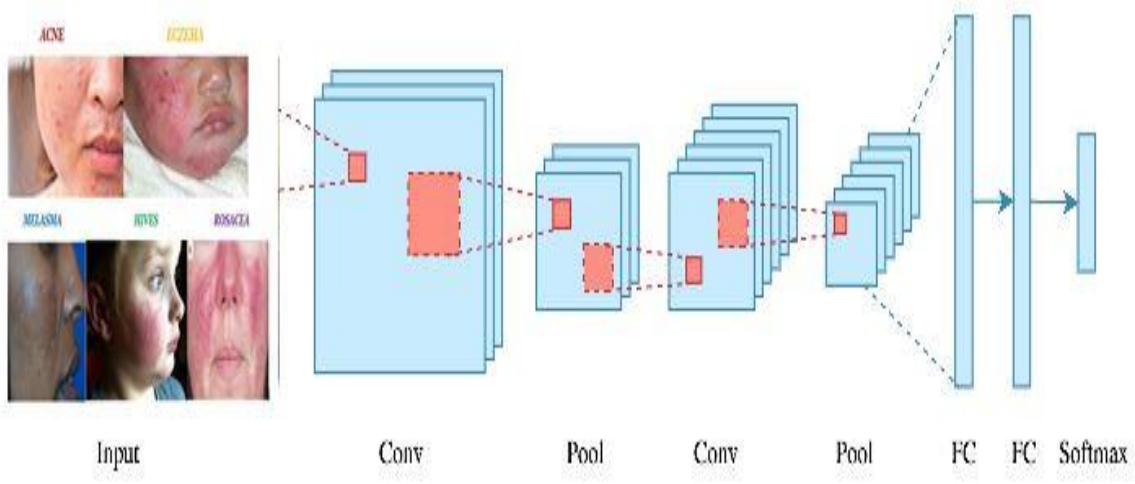

Figure 3-1 architecture of our base model CNN

3.2. CNN Model building

CNN Model from Scratch

```
[ ]  model = models.Sequential()
     model.add(layers.Conv2D(32, (3, 3), activation='relu',
     input_shape=(64, 64, 3)))
     model.add(layers.MaxPooling2D((2, 2)))
     model.add(layers.Conv2D(64, (3, 3), activation='relu'))
     model.add(layers.MaxPooling2D((2, 2)))
     model.add(layers.Conv2D(128, (3, 3), activation='relu'))
     model.add(layers.MaxPooling2D((2, 2)))
     model.add(layers.Flatten())
     model.add(layers.Dropout(0.4))
     model.add(layers.Dense(512, activation='relu'))
     model.add(layers.Dense(5, activation='softmax'))

     model.summary()
```

Figure 3-2 CNN model from scratch model building code block

CNN model was built from scratch which has following structure,

- *Conv2D*: this method creates a convolutional layer. The first parameter is the filter count, and the second one is the filter size. In the first convolution layer we created 32 filters of size 3x3. We used *relu* non-linearity as activation. Stride is 1 for convolution layers by default so we did not change that.
- *MaxPooling2D*: creates a maxpooling layer. We used a 2x2 window as it is the most common. By default, stride length is equal to the window size, which is 2 in our case, so we did not change that.
- *Flatten*: After the convolution and pooling layers we flattened their output to feed into the fully connected layers as we discussed above.
- *Dropout*: As you can see from the above code snippet, we also used dropout in our network architecture. Dropout works by randomly disconnecting nodes from the *current layer* to the *next layer*. This process of random disconnects during training batches helps naturally introduce redundancy into the model

3.3. CNN Model fitting and optimization

Since our training sample size is 1172 images and validation size are 290 images, overfitting happens because of having too few examples to train on, resulting in a model that has poor generalization performance. Data augmentation is a way to generate more training data from our current set and acts as a regularization technique.

```
[ ]  history = model.fit_generator(
                train_generator,
                steps_per_epoch=25,
                epochs=50,
                validation_data=validation_generator,
                validation_steps=50)
```

Figure 3-3 CNN from scratch model fitting

For optimization, we did hyper parameter fine tuning by using different optimizer and added hidden layers with different dropout rates and finalized the one shown in model building part which works better.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 62, 62, 32)        896
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 31, 31, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 29, 29, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 12, 64)        36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 4, 4, 128)         73856
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 2, 2, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_2 (Dense)              (None, 5)                 2565
=================================================================
Total params: 395,397
Trainable params: 395,397
Non-trainable params: 0
_________________________________________________________________
```

Highest Accuracy

```
[17] #Model from scratch using CNN
     print(max(history.history["acc"]))
     print(max(history.history["val_acc"]))

     0.46487868
     0.4048442840576172
```

Figure 3-4 CNN from scratch hyperparameters fine tuning

3.4. CNN model accuracy and performance

Our calculated baseline accuracy is 22% .With hyper parameters and fine tuning, CNN model built from scratch gave a validation accuracy of approximate 44% which is greater than baseline accuracy.

▾ Baseline Accuracy of the Model

```
[ ] a=333 #max
    b=283+333+306+273+267
    print("Baseline_Accuracy=",a/b*100)

    Baseline_Accuracy= 22.77701778385773
```

▾ Highest Accuracy

```
[ ] #Model from scratch using CNN
    print(max(history.history["acc"]))
    print(max(history.history["val_acc"]))

    0.49365482
    0.44413793087005615
```

Figure 3-5 Baseline accuracy and CNN model accuracy

Figure 3-6 CNN model from scratch performance graphs

4. Alternative models and analysis

Apart from our base model CNN built from scratch we decided to build two more alternative models as follows,

- DenseNet121
- MobileNet

4.1. DenseNet121 model building

For alternative model we wanted to build a Dense Convolutional Network (DenseNet) model, which connects each layer to every other layer in a feed-forward fashion .

DenseNet have several other compelling advantages,

- alleviates the vanishing-gradient problem,
- strengthen feature propagation,
- encourage feature reuse and substantially reduce the number of parameters,
- requires less memory and computation to achieve high performance.

Building DenseNet121 Model

```
[ ]  def build_densenet():
         densenet = DenseNet121(weights='imagenet', include_top=False)

         input = Input(shape=(SIZE, SIZE, N_ch))
         x = Conv2D(3, (3, 3), padding='same')(input)

         x = densenet(x)

         x = GlobalAveragePooling2D()(x)
         x = BatchNormalization()(x)
         x = Dropout(0.5)(x)
         x = Dense(256, activation='relu')(x)
         x = BatchNormalization()(x)
         x = Dropout(0.5)(x)

         output = Dense(5,activation = 'softmax', name='root')(x)
         model = Model(input,output)
         optimizer = Adam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=0.1, decay=0.0)
         model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
         model.summary()
         return model
```

Figure 4-1 DenseNet121 model building code block

4.2. DenseNet121 model hyperparameters fine tuning

For optimization, we again did trial and error hyper parameter fine tuning by using different optimizer and added/deleted layers with different dropout rates and changed different epochs at 30,50,100 out of which 50 epochs gave us good accuracy.

➢ Epochs 30

- We did hyperparameter fine tuning at 30 epochs DenseNet model gave accuracy of *41%*
- Total execution time was *2.5 minutes*. Even though computational time was lesser accuracy was not that great it was lesser than our base model built from scratch. So, we further increased our epochs to 50.

Figure 4-2 DenseNet model at epochs 30 with accuracy 41% and exec time of 2.5 mins

➢ Epochs 50

- We increased from 30 to 50 epochs, DenseNet model gave accuracy of *59%* with *highest history accuracy of 64%.*
- Total execution time was 3 minutes. Computational time was efficient with better accuracy that is far greater thana baseline accuracy and accuracy of base model built from scratch.

```
Epoch 1/50
 - 56s - loss: 2.7926 - accuracy: 0.1783 - val_loss: 2.7247 - val_accuracy: 0.1877

Epoch 00001: val_loss improved from inf to 2.72475, saving model to model.h5
Epoch 2/50
 - 5s - loss: 2.6158 - accuracy: 0.2100 - val_loss: 2.3491 - val_accuracy: 0.1604

Epoch 00002: val_loss improved from 2.72475 to 2.34909, saving model to model.h5
Epoch 3/50
 - 5s - loss: 2.4604 - accuracy: 0.2326 - val_loss: 2.0572 - val_accuracy: 0.1706

Epoch 00003: val_loss improved from 2.34909 to 2.05724, saving model to model.h5
Epoch 4/50
 - 5s - loss: 2.3640 - accuracy: 0.2489 - val_loss: 2.0835 - val_accuracy: 0.1672

Epoch 00004: val_loss did not improve from 2.05724
Epoch 5/50
 - 5s - loss: 2.3925 - accuracy: 0.2371 - val_loss: 1.7598 - val_accuracy: 0.2423

Epoch 00005: val_loss improved from 2.05724 to 1.75977, saving model to model.h5
Epoch 6/50
 - 5s - loss: 2.2968 - accuracy: 0.2380 - val_loss: 1.7672 - val_accuracy: 0.2355

Epoch 00006: val_loss did not improve from 1.75977
Epoch 7/50
 - 6s - loss: 2.2220 - accuracy: 0.2751 - val_loss: 1.6614 - val_accuracy: 0.2423
```

Figure 4-3 DenseNet model at epochs 30 and total exec time of approx. 3 minutes

➢ Epochs 100

– We increased from 50 to 100 epochs, DenseNet model gave accuracy of *65%*

– Total execution time was approx. *15 minutes which is thrice the execution time of 50 epochs*. Computational time is inefficient with increase in one percent of accuracy when compared to 30 epochs so 100 epochs must be avoided

```
Epoch 00095: val_loss did not improve from 0.95117
Epoch 96/100
 - 8s - loss: 0.7022 - accuracy: 0.7464 - val_loss: 0.9672 - val_accuracy: 0.6199

Epoch 00096: val_loss did not improve from 0.95117
Epoch 97/100
 - 9s - loss: 0.7821 - accuracy: 0.7005 - val_loss: 1.8081 - val_accuracy: 0.4521

Epoch 00097: val_loss did not improve from 0.95117
Epoch 98/100
 - 8s - loss: 0.8486 - accuracy: 0.7052 - val_loss: 2.4822 - val_accuracy: 0.5514

Epoch 00098: val_loss did not improve from 0.95117
Epoch 99/100
 - 8s - loss: 0.7250 - accuracy: 0.7336 - val_loss: 1.5756 - val_accuracy: 0.6130

Epoch 00099: ReduceLROnPlateau reducing learning rate to 0.001.

Epoch 00099: val_loss did not improve from 0.95117
Epoch 100/100
 - 8s - loss: 0.7257 - accuracy: 0.7409 - val_loss: 1.1063 - val_accuracy: 0.6370

Epoch 00100: val_loss did not improve from 0.95117
```

▾ Highest Accuracy

```
[18] print(max(hist.history["accuracy"]))
     print(max(hist.history["val_accuracy"]))

     0.75545454
     0.6541095972061157
```

Figure 4-4 DenseNet model at epochs 100 with accuracy 65% and exec time of 15 mins

4.3. DenseNet121 model accuracy and performance

With all hyperparameter and fine tuning, we found DenseNet model we ran at 50 epochs gave better accuracy and time efficient as it took only 3 minutes for execution.

Our calculated baseline accuracy is 22% .With hyper parameters and fine tuning, DenseNet model we built gave a validation accuracy of approximate 59% which is greater than baseline accuracy.

Data Augmentation and Model Fitting

```
[ ] model = build_densenet()
    annealer = ReduceLROnPlateau(monitor='val_accuracy', factor=0.5, patience=5, verbose=1, min_lr=1e-3)
    checkpoint = ModelCheckpoint('model.h5', verbose=1, save_best_only=True)
    # Generates batches of image data with data augmentation
    datagen = ImageDataGenerator(rotation_range=360, # Degree range for random rotations
                            width_shift_range=0.2, # Range for random horizontal shifts
                            height_shift_range=0.2, # Range for random vertical shifts
                            zoom_range=0.2, # Range for random zoom
                            horizontal_flip=True, # Randomly flip inputs horizontally
                            vertical_flip=True) # Randomly flip inputs vertically

    datagen.fit(X_train)
    # Fits the model on batches with real-time data augmentation
    hist = model.fit_generator(datagen.flow(X_train, Y_train, batch_size=BATCH_SIZE),
                   steps_per_epoch=X_train.shape[0] // BATCH_SIZE,
                   epochs=EPOCHS,
                   verbose=2,
                   callbacks=[annealer, checkpoint],
                   validation_data=(X_val, Y_val))
```

```
Model: "model_1"
_________________________________________________________________
```

Figure 4-5 DenseNet Model fitting

Baseline Accuracy of the Model

```
[ ] a=333 #max
    b=283+333+306+273+267
    print("Baseline_Accuracy=",a/b*100)
```

```
Baseline_Accuracy= 22.77701778385773
```

Highest Accuracy

```
[ ] print(max(hist.history["accuracy"]))
    print(max(hist.history["val_accuracy"]))
```

```
0.5902778
0.5904436707496643
```

Figure 4-6 Baseline accuracy and last run DenseNet model accuracy

Also, highest accuracy from history is 64% which means the DenseNet model's maximum accuracy is 62% which is far greater than our baseline accuracy and CNN model accuracy.

Figure 4-7 DenseNet model's highest history accuracy of 64%

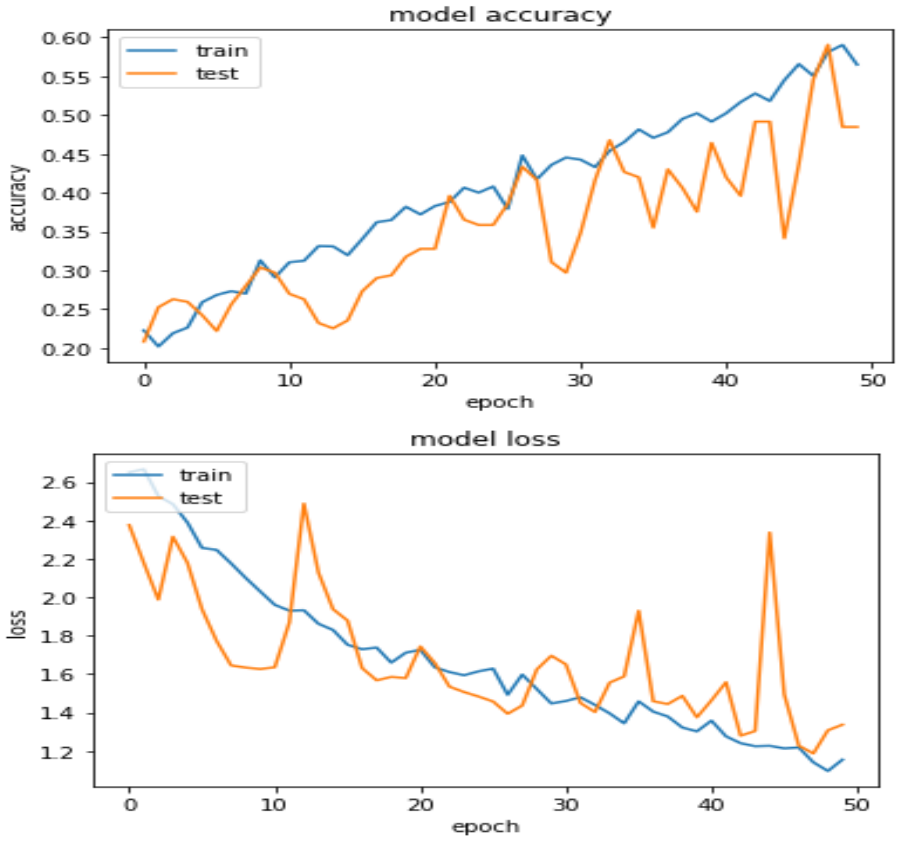

Figure 4-8 DenseNet model performance graphs

5. Advanced transfer learning model – MobileNet model

5.1. MobileNet model building

Future scope of our model is to develop our model further into an app, so we wanted to try MobileNet model for our advanced alternative model as it is the first *mobile-first* computer vision models for TensorFlow, designed to effectively maximize accuracy while being mindful of the restricted resources for an on-device or embedded application.

MobileNet have several other compelling advantages,

- Small and low latency,
- low-power models parameterized to meet the resource constraints of a variety of use cases
- they can be built upon for classification, detection, embeddings, and segmentation

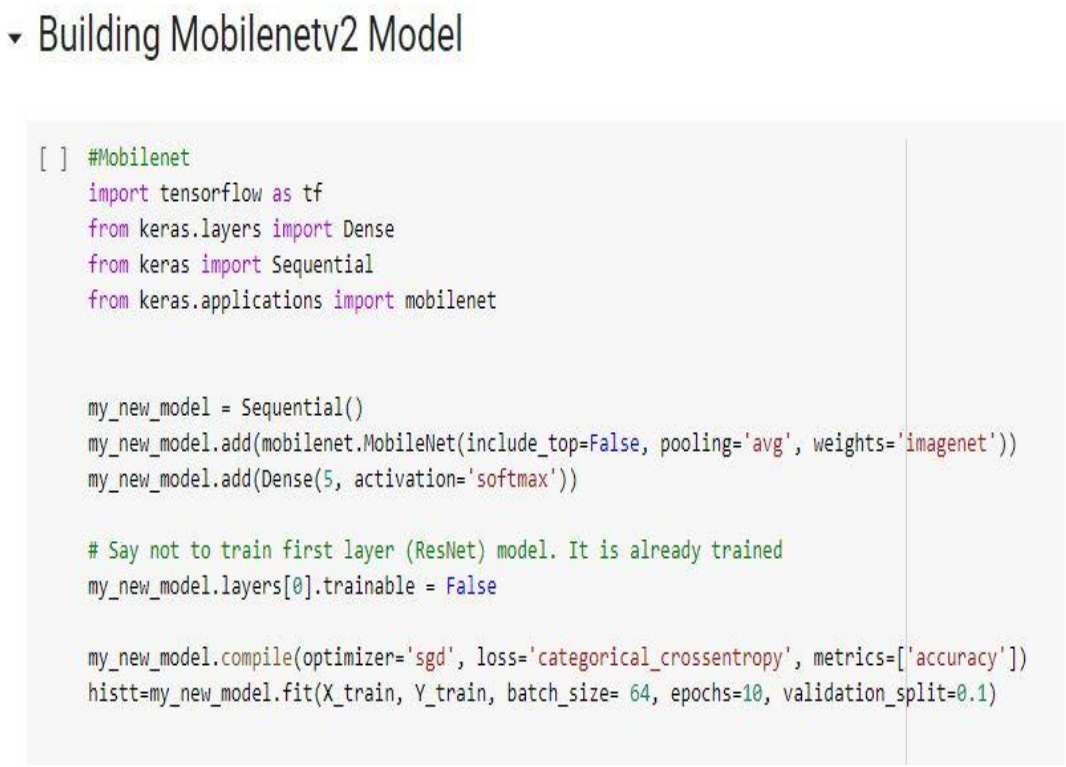

Figure 5-1 MobileNet model building code block

5.2. MobileNet model accuracy and Performance

- MobileNet model we built gave a validation accuracy of approximate 34% after fine tuning. Even though this accuracy is greater than baseline accuracy, lesser than other two models.
- Total execution time taken is 5 seconds which is much faster than other models and accuracy is decent.

▾ Baseline Accuracy of the Model

```
[ ] a=333 #max
    b=283+333+306+273+267
    print("Baseline_Accuracy=",a/b*100)

    Baseline_Accuracy= 22.77701778385773
```

▾ Highest Accuracy

```
[ ] print(max(histt.history["accuracy"]))
    print(max(histt.history["val_accuracy"]))

    0.6207224
    0.34188035130500793
```

Figure 5-2 Baseline accuracy and last run MobileNet model accuracy

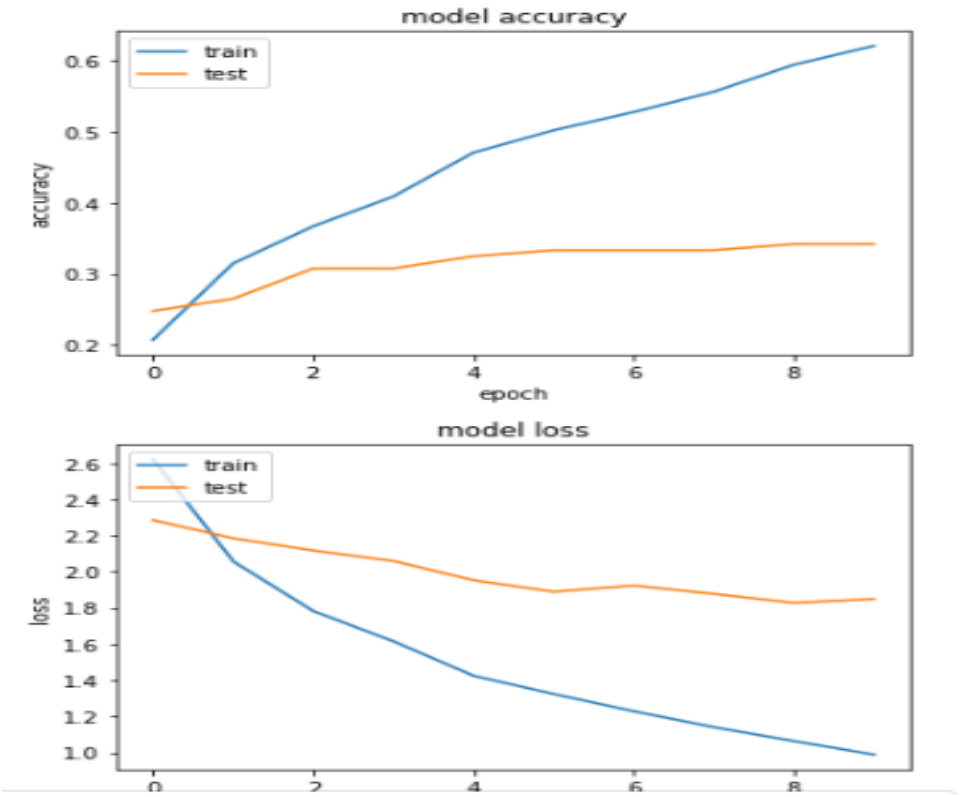

Figure 5-3 MobileNet model performance graphs

6. Comparative analysis of all three models- CNN from scratch, DenseNet, MobileNet

- From the comparative analysis of three models we built for our project, DenseNet121 alternative model performs better in terms of accuracy and time efficiency

- So, we finalized DenseNet model for our project implementation and real time testing.

CNN model from scratch	DenseNet 121 model	MobileNet model
• *Accuracy of 44% > baseline accuracy of 22%*	• *Accuracy of 59% with highest history 62% > baseline accuracy of 22%*	• *Accuracy of 35% > baseline accuracy of 22%*
• *15.8 mins execution time*	• *3.98 mins execution time*	• *5 seconds execution time*
• *Time consuming and decent accuracy*	• *Good accuracy and time efficient*	• *Time efficient but bad accuracy*

Table 6-1 Comparative analysis of all three models

7. DenseNet model implementation and Testing

We implemented our project real time , we took images both from outside our dataset and from validation set to test DenseNet model which seem to classify most of those fed images accurately, we even tested some hand images which were classified correctly, however if you notice the whole dataset is built on facial images, so that is a win.

At 64% max accuracy, our model could able to detect most of the images, recognize and classified the type of facial skin disease correctly.

➢ Image taken outside dataset (not even a facial image) and tested-> Classified correctly by our model

```
from skimage import io
from keras.preprocessing import image
#path='imbalanced/Scratch/Scratch_400.jpg'
img = image.load_img('58.jpg', grayscale=False, target_size=(64, 64))
show_img=image.load_img('58.jpg', grayscale=False, target_size=(200, 200))
skindisease_types=['Acne','Melasma','Rosacea','Hives','Eczema']
x = image.img_to_array(img)
x = np.expand_dims(x, axis = 0)
x /= 255
custom = model.predict(x)
print(custom[0])
x = np.array(x, 'float32')
plt.imshow(show_img)
plt.show()
a=custom[0]
ind=np.argmax(a)
print('Prediction:',skindisease_types[ind])
```

```
[0.2571444  0.04905971 0.02494301 0.63049215 0.03836072]
```


```
Prediction: Hives
```

Figure 7-1 Correct classification of 'hives' hand image that was randomly taken from outside of our dataset

- A facial Image taken from validation dataset and tested -> Classified correctly by our model

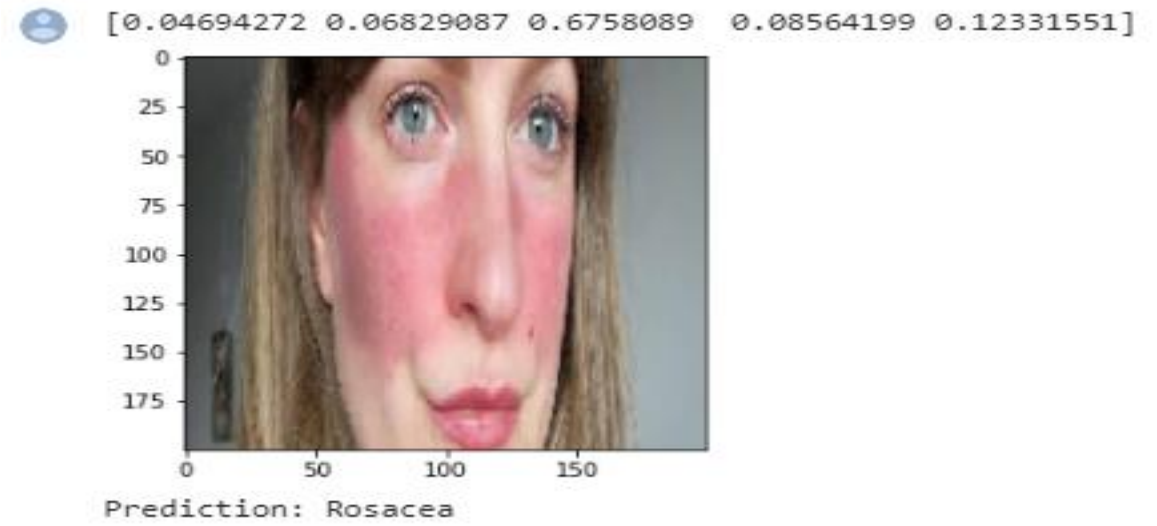

Figure 7-2 Correct classification of 'Rosacea' facial image that was randomly taken from outside of our dataset

- A facial image taken validation dataset and tested -> Classified incorrectly by our model because our model is only 64% efficient

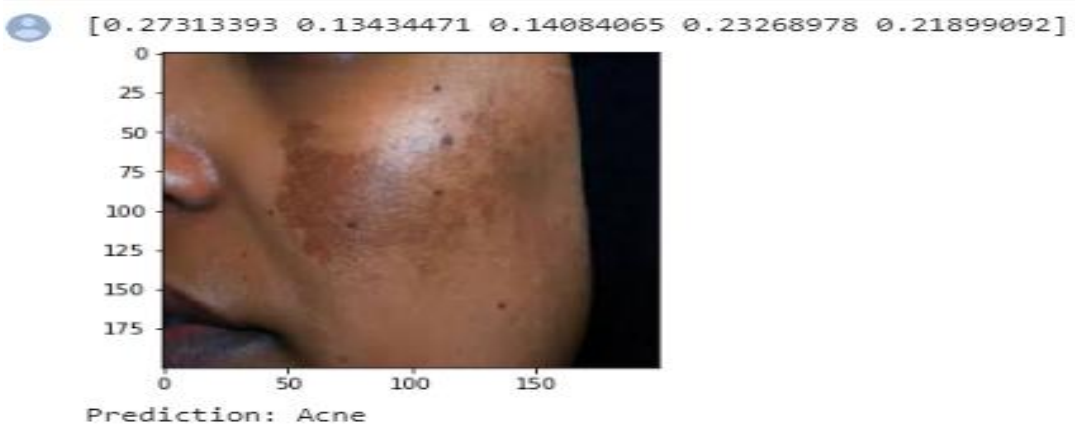

Figure 7-3 Incorrect classification of 'Melasma' facial image as 'Acne'

8. Challenges faced

- One of the main challenges we faced is to custom built our dataset since there are not projects available in similar area , also with no active Kaggle dataset or any dataset on online for that matter, building an approximate 1500 images dataset was quite challenging.

- Cleaning the images was other issue we faced since we had to manual clean it, we had to spend quite some time on it to clean, preprocess and make it fit for our project.

- One other challenge we faced was, since our dataset is custom built, we could come up with approx. 1500 images as our best, but this sample size is relatively smaller to build an solid model however we could achieve 65% accuracy.

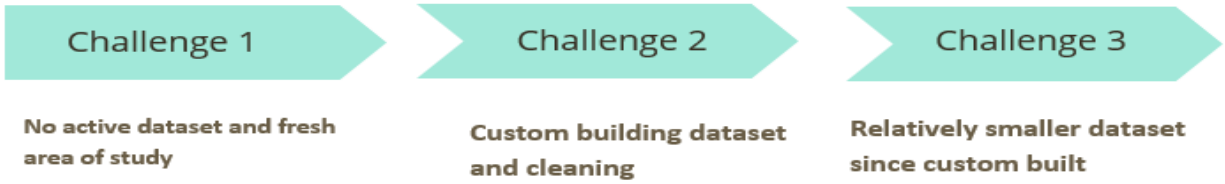

Figure 8-1 A quick snippet of challenges we faced

9. Conclusions

- Our Facial skin disease detection model at 64% efficiency performs well and could able to detect even non facial images correctly.
- Future scope of our model is it can be deployed into an iPhone and iOS app using CoreML framework. Also, MobileNet model with bigger sample size with further hyperparameter and fine tuning can be used as an architecture for Android app development
- From business perspective, we strongly believe our model could be developed into an interesting app that is highly useful for users who can instant scan and get advice on non-threatening skin diseases at one click away.
- And we believe cost to develop our model as an app and pilot it would be relatively low. Also, highly time and resources efficient.

REFERENCES

- Deep Learning (http://www.deeplearningbook.org/) by Ian Goodfellow, Yoshua Bengio and Aaron Courville MIT Press
- https://towardsdatascience.com/
- https://www.medicalnewstoday.com/articles/316622
- https://www.onhealth.com/content/1/adult_skin_diseases
- https://www.healthline.com/health/skin-disorders
- Google.com for all day to day references and internet searches
- Dr. Surendra Sarnikar for valuable inputs and feedback throughout the project execution