KOLEJ UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF COMPUTING AND INFORMATION TECHNOLOGY

Assignment Title

BMCS2114 MACHINE LEARNING

2021/2022

Student's name/ ID Number	:	Tay Xue Hao / 21WMR04790
Student's name/ ID Number	:	Lee Ming Yi / 21WMR04782
Student's name/ ID Number	:	Khoo Chyi Ze / 21WMR04778
Programme	:	RDS2 Y2S3
Tutorial Group	:	Group 2
Tutor's name	:	Dr Sandy Lim Siew Mooi

Human Emotion Recognition System

Dr Lim Siew Mooi, Mr Lee Ming Yi, Mr Tay Xue Hao, Mr Khoo Chyi Ze

Faculty of Computing and Information Technology, Tunku Abdul Rahman University College, Kuala Lumpur, Malaysia

siewmooi@tarc.edu.my, leemy-wm19@student.tarc.edu.my, tayxh-wm19@student.tarc.edu.my, khoocz-wm19@student.tarc.edu.my

Abstract:

Communicating using Language often inhibits the communication of humans' emotions, whereas faces are more honest. We can lie using our words, but not with our face. We propose a solution using a Facial Emotion Recognition(FER) system which can be used in various different domains such as sentiment analysis or panic attack detection. Our Exploratory Data Analysis showed us that the dataset we are using is imbalanced and has image classes that overlap, such as the angry and sad class. In this work, we have performed image augmentation such as shifting, and rotating. We also split the dataset into train, validation and test sets with the ratio of 64:16:20. In the model building phase, we have trained our custom built CNN and 4 pretrained models which are VGG16, ResNet50V2, MobileNetV2, and DenseNet169 using transfer learning. All the models are trained using Adam Optimizer, and softmax activation. The best model is our custom CNN, whereas the other models performed terribly with approximately 40% accuracy. We then performed hyperparameter tuning on the custom CNN by adding hidden layers and neurons, decreasing learning rate and batch size, increasing the number of filters in the last Convolution block, and adding more epochs. Furthermore, we also experienced different optimization algorithms like, Stochastic Gradient Descent (SGD), SGD + Nesterov, Adam + Nesterov (Nadam) , and RMSprop. The best optimizer was Nadam which achieved an accuracy of 72% on the training set and 65% on the test set. We concluded the assignment picking Custom CNN with 4 blocks of Convolution layers and 3 hidden layers + Nadam optimizer as our final and best model.

Keywords: Emotions, Convolutional Neural Networks, Exploratory Data Analysis, Image Preprocessing, Transfer Learning, Optimization algorithms

Table of Contents

1.0 Introduction

1.1 Problem Statement

Humans often face problems when trying to understand emotions of people through language because a lot of information is lost when we try to convert and compress thoughts into words (Elon Musk 2020). Whereas the face is more honest given they are made subconsciously . Facial expression is a sophisticated tool we humans utilize everyday to communicate our emotions and opinions towards an object, event or person. Based on reliable sources, they have done a statistic which is relevant to emotions. People form automatic judgments about the PLOS(Public Library of Science) trustworthiness of a new face as quickly as 50-100ms, according to research on facial perception. Hence, there are many possible permutations of the face, but we express 7 main emotions in our daily lives, which can be categorized as anger, disgust, fear, happiness, neutral, sadness, and surprise.

To solve the problem, we propose a solution using an AI emotion detection model, we can recognize if a human is feeling sad or angry, etc, based on the probability of emotions calculated by the model. With this model, we can potentially revolutionize the paradigms of multiple domains, such as business, where we can understand clients' feelings about a product. Or domains like healthcare, where a psychologist can detect and record changes in emotions of a patient. Other domains like security, entertainment, etc can find value in this system as well. Consequently, we strongly believe that this is a powerful tool that can be used to understand human emotions and the pragmatic value of the system, hence we will try to build a Human Emotion Recognition System that can accurately classify input faces into the 7 emotion classes.

1.2 Objective

We aim to build an AI models which is able to identify 7 different emotions which are angry, disgusted, fearful, happy, neutral, sad and surprised with an accuracy of 65% or above in emotion detection which is the human level of accuracy in classifying this dataset (Yousif Khaireddin).

2.0 Literature Review

2.1 Convolutional Neural Network (CNN)

In deep learning, a convolutional neural network (a.k.a. CNN or ConvNet) is a class of artificial neural networks that commonly implement visual imagery. Image classification is critical, because it has a big impact on our education, jobs, and daily lives. Image pre-processing, segmentation, extraction of critical features, and matching identification are all part of picture classification. We use the most up-to-date image classification techniques to solve scientific problems.We use the most up-to-date image classification algorithms in scientific investigations, traffic detection, security, medical equipment, face recognition, and other fields, not merely to collect image data faster than before. During the development of deep learning, feature extraction and classifiers were merged into a learning framework, overcoming the traditional approach of feature selection problems. The purpose of deep learning is to find multiple layers of representation.

In the 1950s and 1960s, Hubble and Wiesel recommended a visual structure model based on cat visual cortices. For the first time, the concept of receptive field has been proposed. Next in 1980, Neocognition was introduced by Kunihiko Fukushima. It was a more improved version of Hubel and Wiesel. The convolutionary and downsampling layers of the CNN are introduced by Neocognition. Then, in the year of 1989, Yann LeCun applied back-propagation to study the convolution kernel coefficient directly from photographs of handwritten numbers. As a result, learning was totally automatic, outperformed manual coefficient creation, and was applicable to a wider range of image identification challenges and image kinds.

RNNs, SVMs, ANNs, and other image classification algorithms have all been developed, but the Convolutional Neural Network, a Machine Learning approach, has excelled them all. They employed such CNNs on GPU to win an image recognition competition in 2011, the first time they attained superhuman performance. Their CNN Projects has won no less than 4 image competitions. These several characters show that the algorithm and technologies of using image processing are getting more and more advanced. We are able to see more and more of CNN footprints in our daily life in the future

A CNN employs a technology similar to a multilayer perceptron that is optimized for low processing requirements.(TechTarget Contributor, 2022) An input layer, an output layer, and a hidden layer with several convolutional layers, pooling layers, fully connected layers, and normalizing layers make up CNN's layers (TechTarget Contributor, 2022). A typical CNN can be divided into two parts, the Feature Learning layers, and the Classification Layers. The feature learning part contains multiple blocks, each block consists of the convolution layers, an activation function like ‘relu’ , a pooling layer (max,average, etc), and a Batch Normalization layer. Whereas the Classification Layer will typically have the flatten layer, fully connected layers with an activation function like ‘relu’, and a last layer with a probability conversion activation function like softmax or sigmoid. The removal of constraints and improvements in image processing efficiency result in a system that is significantly more effective and easier to train for image processing and natural language processing. (TechTarget Contributor, 2022).

For example, in the figure 2.1.1, there’s an image of a boat on the left as an input. The image will undergo 2 main processes in CNN, convolution and pooling process. Convolutional process is when the process is to extract the high-level features such as edges from the input image (Saha, 2018). The Pooling layer, like the Convolutional Layer, is responsible for shrinking the Convolved Features spatial size.(Saha, 2018) This is done to reduce the amount of computer power needed to process the data by reducing the dimensionality of the data. (Saha, 2018)

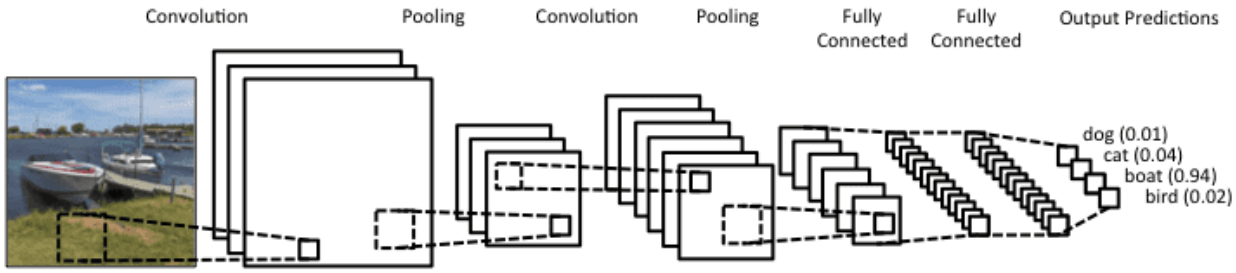


Figure 2.1 CNN Architecture

Rectified Linear Unit (ReLU) is another non-linear activation function which has gained prominence in the deep learning sector. The key benefit of employing ReLU functions over other activation functions is that ReLU does not simultaneously stimulate all neurons. Transfer learning is a machine learning technique in which a model created for one job is utilized as the

basis for a model on a different task. (Brownlee, 2017) It is a popular approach in deep learning where pre-trained models are used as the starting point on computer vision and natural language processing tasks. (Brownlee, 2017)

2.2 VGG16

VGG16 is considered as one of the convolutional neural networks (CNN). This CNN model has won the 2014 ILSVRC(Imagenet) competition. The most distinctive feature of VGG16 is that, rather than having a large number of hyper-parameters, they focused on having 3x3 filter convolution layers with a stride 1 and always used the same padding and maxpool layer of 2x2 filter stride 2. (Thakur, 2019)

Karen Simonyan and Andrew Zisserman of the University of Oxford created and launched the VGG16 Architecture in their essay "Very Deep Convolutional Networks for Large-Scale Image Recognition" in 2014. (John Perumanoor, 2021) The term 'VGG' stands for Visual Geometry Group, a group of scholars at the University of Oxford who designed this architecture, and the number '16' indicates that there are 16 layers in this architecture. (John Perumanoor, 2021)

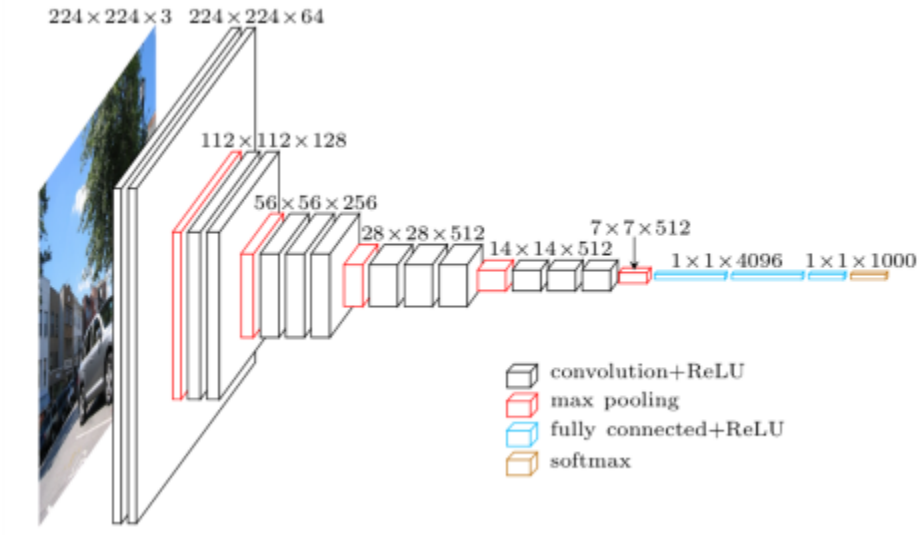


Figure 2.2 VGG16 Architecture

2.3 MobileNetV2

MobileNetV2 is a convolutional neural network architecture that aims to be mobile-friendly. It is built on an inverted residual structure, with residual connections between bottleneck layers. (Papers with Code - MobileNetV2 Explained, n.d.) As a basis of non-linearity, the intermediate expansion layer filters features with lightweight depthwise convolutions. (Papers with Code - MobileNetV2 Explained, n.d.)Overall, MobileNetV2's architecture includes a fully convolutional layer with 32 filters, followed by 19 residual bottleneck layers. (Papers with Code - MobileNetV2 Explained, n.d.)

Google announced MobileNets, a collection of TensorFlow-based computer vision models, in 2017 that are meant to maximize accuracy while keeping in mind the limited resources available for on-device or embedded applications. (Rodriguez, 2021)MobileNets are low-latency, low-power models that have been parameterized to match the resource restrictions of various use cases. The launch of MobileNetV2 in April 2018 and MobileNetV3 in May 2019. (Rodriguez, 2021)

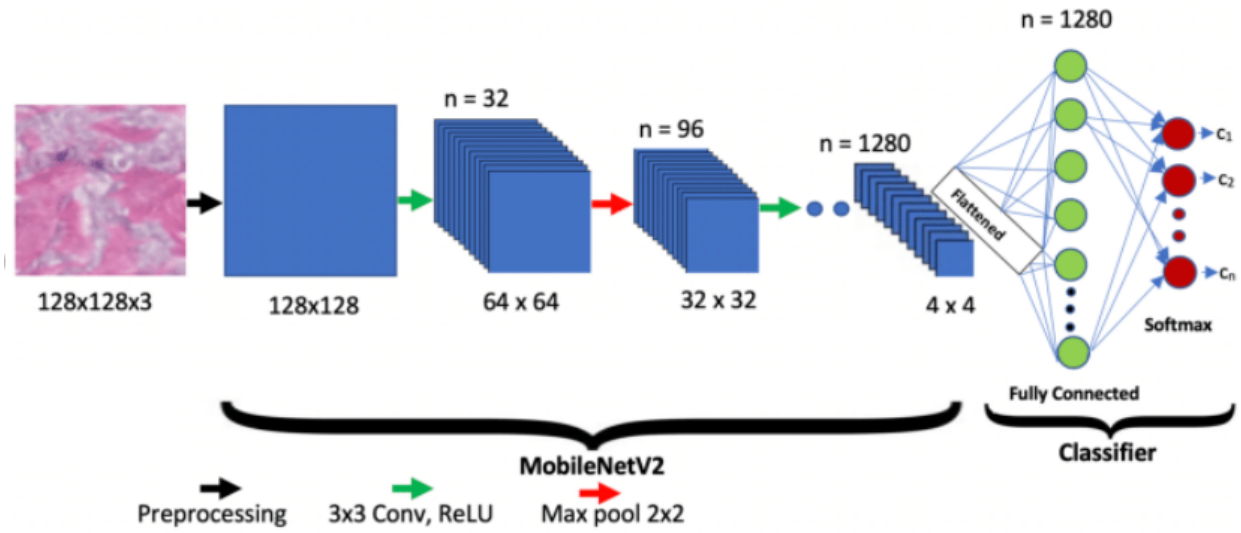


Figure 2.3 MobileNetV2 Architecture

2.4 DenseNet169

The name DenseNet comes from Densely Connected Convolutional Network in the fact that each layer in a DenseNet design is connected to every other layer. L(L+1)/2 direct connections exist for L layers. (DenseNet Architecture Explained with PyTorch Implementation from TorchVision, 2020)The feature maps of all previous layers are utilized as inputs for each layer, and its own feature maps are used as inputs for subsequent layers. (DenseNet Architecture Explained with PyTorch Implementation from TorchVision, 2020)

DenseNets can make the connectivity pattern introduced in earlier architectures easier to understand. DenseNets make use of the network's potential by reusing features. DenseNets require fewer parameters than regular CNNs because they eliminate the burden of learning redundant feature maps.

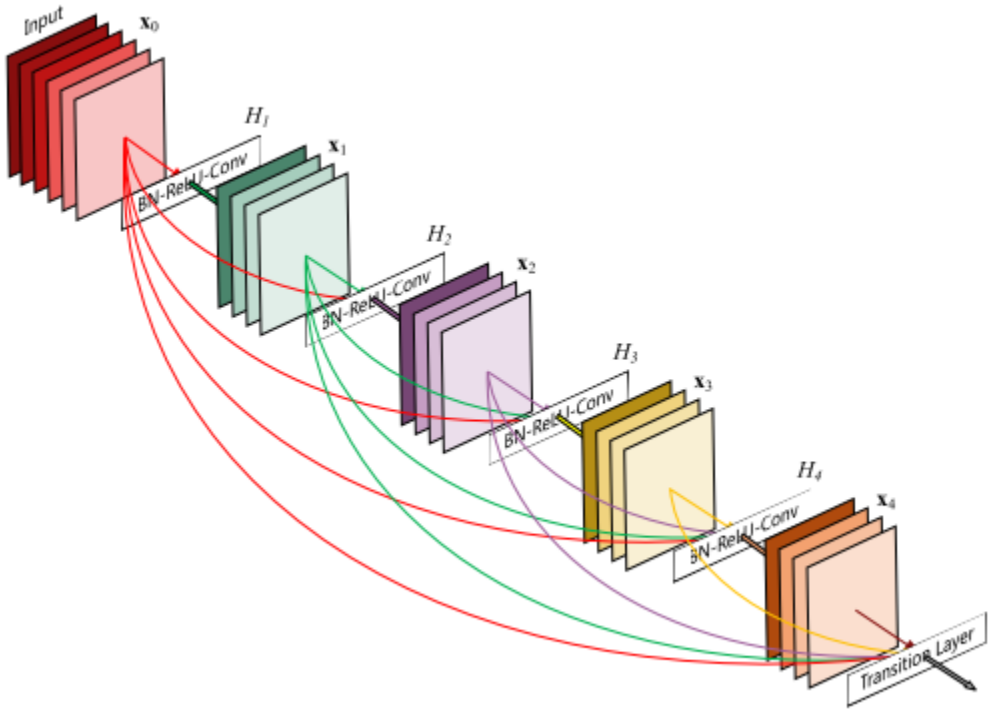


Figure 2.4 DenseNet169 Architecture

2.5 ResNet50V2

He et al. proposed ResNet, which won the 2015 ImageNet competition. Deeper networks can be trained using this strategy. The accuracy becomes more saturated as the network grows deeper. It's not even due to the presence of a high number of parameters, but due to a reduction in the training error.This scenario happens due to the gradients' inability to be back propagated.

The ResNet-50 model is divided into five stages, each with its own convolution and identity block. (Deep Residual Learning for Image Recognition, 2015) There are three convolution layers in each convolution block, and three convolution layers in each identity block. There are around 23,000,000 trainable parameters in the ResNet-50. (Deep Learning for Computer Vision, n.d.)

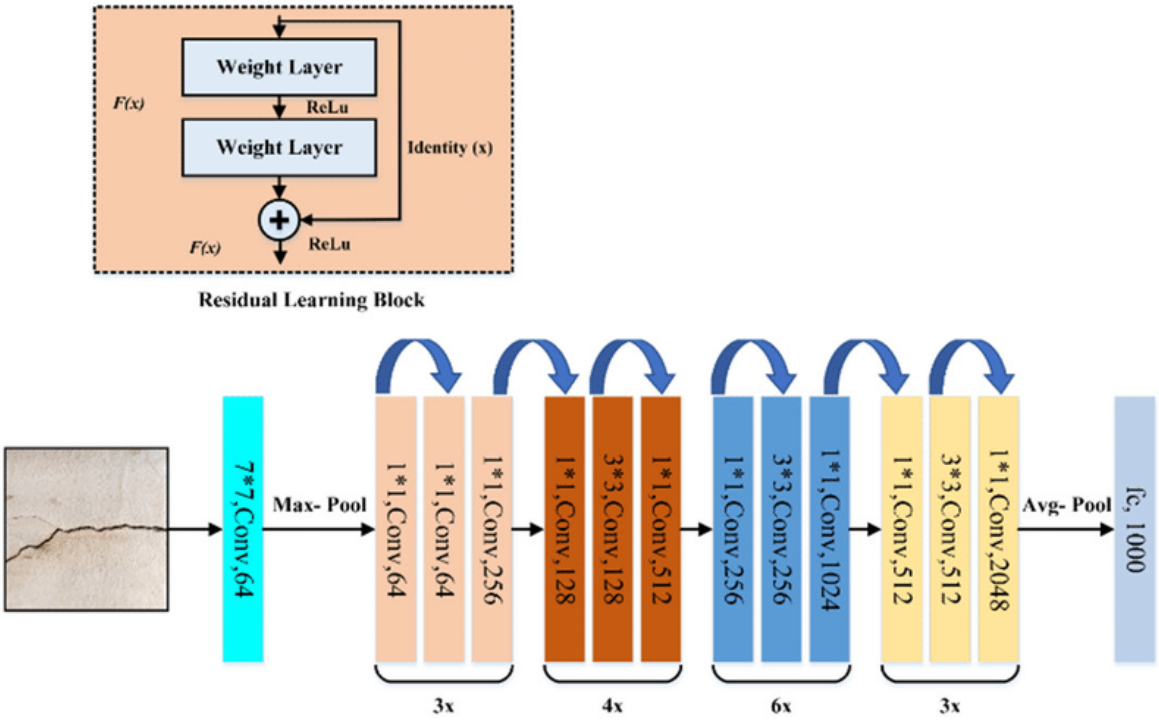


Figure 2.5 ResNet50V2 Architecture

3.0 Methodology

3.1 Exploratory Data Analysis

The data set of our project was retrieved from Kaggle. Our problem is the Emotion Recognition system which is a type of multiclass classification problem in which we have 7 different classes to classify using CNN. Our data has been split by the publisher of the dataset into test and train images. The testing data of the Emotional dataset consist of 7 classes with a total of 7178 images. There are 958 images for angry, 111 images for disgusted, 1024 images for fearful, 1774 images for happy, 1233 images for neutral, 1247 images for sad and 831 images for surprised. On the other hand, the training data for the Emotional dataset also consist of 7 classes with a total of 28709 images. There are 3995 images for angry, 436 for disgusted, 4097 for fearful, 7215 for happy, 4965 for neutral, 4830 for sad and 3171 for surprised.

Step 1: We perform label encoding to every emotion and class labels to ready them for our algorithms. Every category of emotion are differentiated as below:

Emotion	Angry	Disgusted	Fearful	Happy	Neutral	Sad	Surprise
Index Numbering	0	1	2	3	4	5	6

```
# Initialize class labels
expressions = ['angry', 'disgusted', 'fearful', 'happy',
        'neutral', 'sad', 'surprised']
```

Step 2: We load the dataset and store them in a 2D array using Numpy. We are able to store these images in 2D numpy arrays / in the form of a matrix. The first array describes image pixel (48 x 48) while the second array displays the class label/expression of the image

```
# Load dataset function

# Some images may be in different size, we will standardize
# our images to 48*48
pic_size = 48

def load_dataset(path):
    dataset = [] # Multidimensional array containing [image_array, class]
    for expr in expressions:
        full_path = os.path.join(path, expr)
        class_idx = expressions.index(expr) # Label encode the images using the expressions list we have created above
        for img in os.listdir(full_path):
            try:
                img_arr = cv2.imread(os.path.join(full_path, img))
                standardized_arr = cv2.resize(img_arr, (pic_size, pic_size))
                dataset.append([img_arr,class_idx])
            except Exception as e:
                print(e)

    return np.array(dataset)
```

Step 3: Loads the data into train and test sets

```
# specify the path to the dataset folder
path = "C:\\Users\\Acer\\Desktop\\ML Assignment\\"

# pass the arguments to the load_dataset function to get our data
train_set = load_dataset(path + 'train')
test_set = load_dataset(path + 'test')
```

Step 4: Identify the total amounts of test and training data in our datasets. From the output, we can see that for train and test set, there are:

```
: # Dimensions of our data
  print('Dimension of train_set array: ' , train_set.shape)
  print('Dimension of test_set array : ' , test_set.shape)
```

```
Dimension of train_set array:  (28709, 2)
Dimension of test_set array :  (7178, 2)
```

To test whether our matrix store in the exact way as we want:

```
# Examine the first row of the matrix
train_set[0]
```

```
array([array([[[ 70,  70,  70],
               [ 80,  80,  80],
               [ 82,  82,  82],
               ...,
               [ 52,  52,  52],
               [ 43,  43,  43],
               [ 41,  41,  41]],

              [[ 65,  65,  65],
               [ 61,  61,  61],
               [ 58,  58,  58],
               ...,
               [ 56,  56,  56],
               [ 52,  52,  52],
               [ 44,  44,  44]],

              [[ 50,  50,  50],
               [ 43,  43,  43],
               [ 54,  54,  54],
               ...,
               [ 49,  49,  49],
               [ 56,  56,  56],
               [ 47,  47,  47]],
```

```
# Examine the shape of a single image ___can change to plot pixel size
train_set[0][0].shape
```

```
(48, 48, 3)
```

The two outputs above show that our image has resolution of 48x48 and our image has 3 channels, although three channels have the same value since our image is in grayscale.

```
# Examine the class of the first image
train_set[0][1]
```

```
0
```

The output above tells us the class of the first image. Index 0 means it is from the class 'angry'

Step 5 : Merging Train and Test set for the time being

As we are not training our model yet, we are not required to split our dataset into a train and test set. WIth that, we can merge both sets together into one. There are two reasons why we are merging them. The first reason is that we are going to further split our dataset into train, validation and test sets.

Then, the second reason is that we only have to deal with one dataset in the preprocessing phase. So, we have to merge 2 dataset into 1 in order to perform the preprocessing successfully.

```
# Row-wise concatenation should give us 28709 + 7178 = 35887 rows.
data = np.concatenate((train_set, test_set), axis = 0)
```

```
data.shape
```

```
(35887, 2)
```

After merging the Train and Test set, there are a total of 35887 rows and 2 columns as we can see from the data shape. The column remains the same as before.

Step 6: Image visualization

The next step is to display our image using the matplotlib to test our dataset. But before that, we had created 2 functions to help in the image displaying. The first function is 'plot_class_image'. This function is to display the first 9 images in the given class of the dataset and shows the number of images in the class. To use this function, we need to pass in the image dataset, and the expression class.

Next, the second function is 'plot_class_images' that plots the first 9 images of a class in the dataset and also shows the total number of images of that set. To use this function, we only need to pass in the images set, we will use this to plot our images only after we subset the images.

Displaying Images:

Then we use the functions that we created to display the images for each class.

1. **Angry**

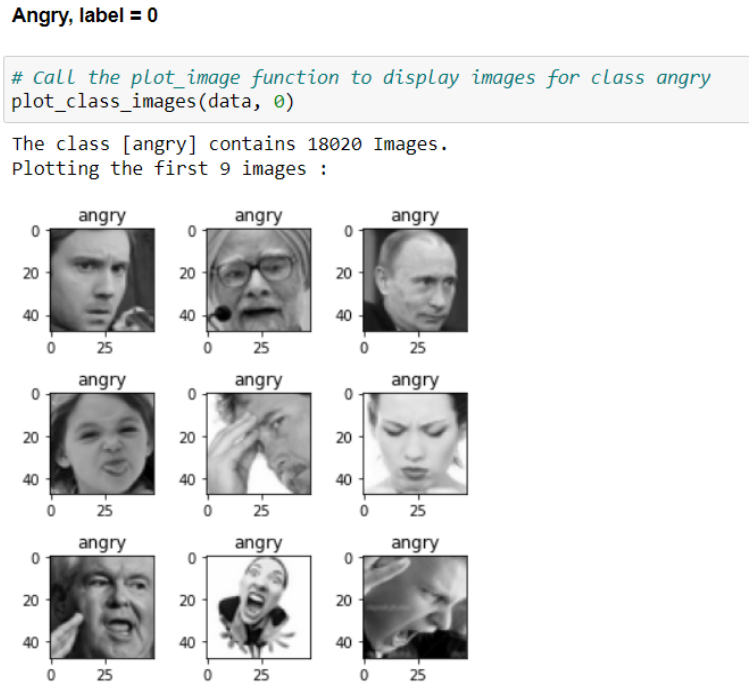


Figure 3.1.1 Angry images

2. Disgusted

Disgusted, label = 1

```
# Call the plot_image function to display images for class disgusted
plot_class_images(data, 1)
```

```
The class [disgusted] contains 18020 Images.
Plotting the first 9 images :
```

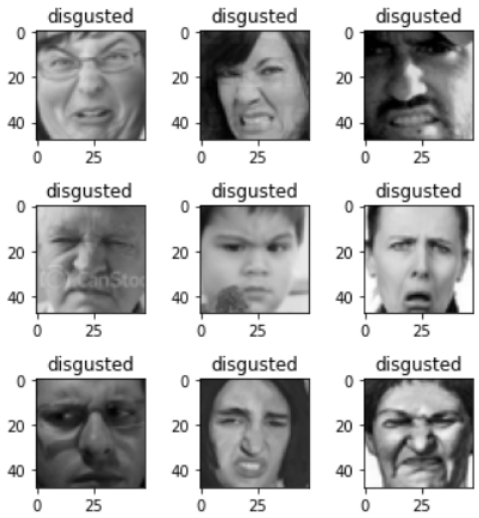


Figure 3.1.2 Disgusted images

3. Fearful

```
# Call the plot_image function to display images for class fearful
plot_class_images(data, 2)
```

```
The class [fearful] contains 18020 Images.
Plotting the first 9 images :
```

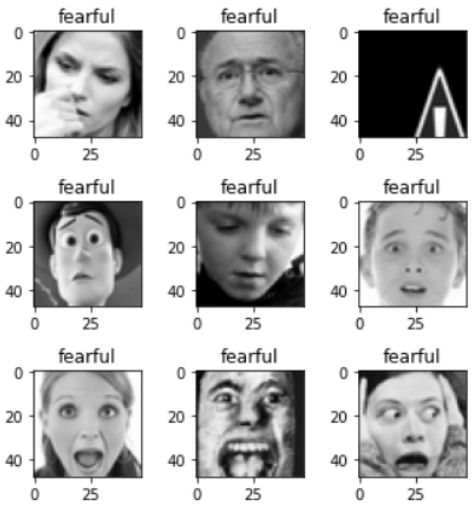


Figure 3.1.3 Fearful images

There is an outlier in the fearful class.

4. Happy

```
# Call the plot_image function to display images for class happy
plot_class_images(data, 3)
```

```
The class [happy] contains 18020 Images.
Plotting the first 9 images :
```

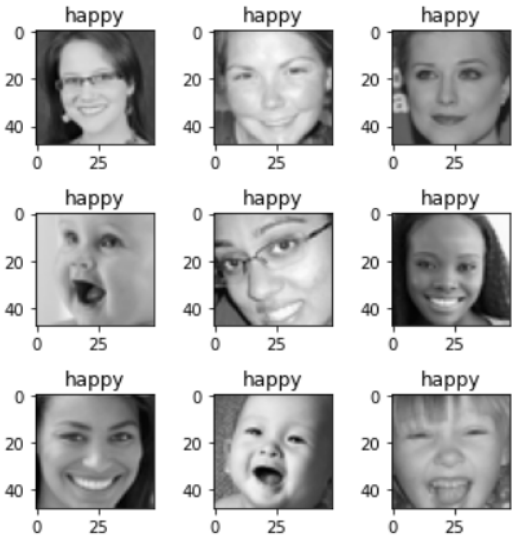


Figure 3.1.4 Happy images

5. Neutral

Neutral, label = 4

```
# Call the plot_image function to display images for class neutral
plot_class_images(data, 4)
```

```
The class [neutral] contains 18020 Images.
Plotting the first 9 images :
```

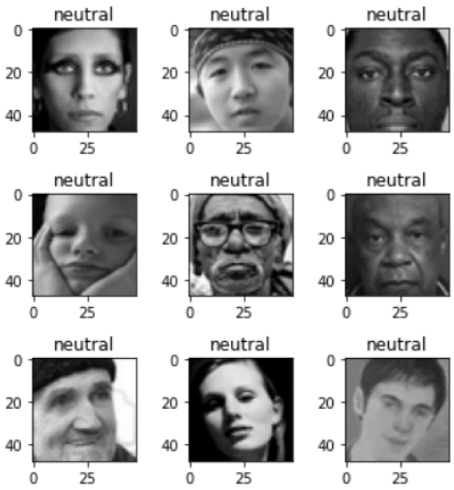


Figure 3.1.5 Neutral images

6. Sad

Sad, label = 5

```
# Call the plot_image function to display images for class sad
plot_class_images(data, 5)
```

```
The class [sad] contains 18020 Images.
Plotting the first 9 images :
```

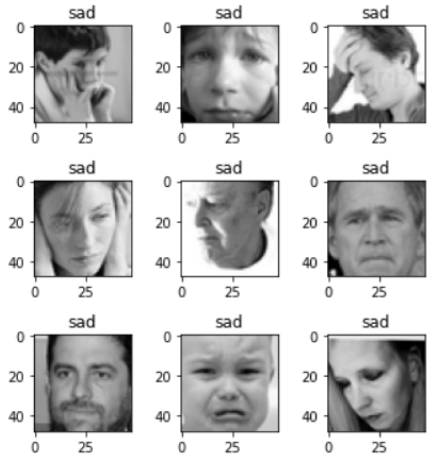


Figure 3.1.6 Sad images

7. Surprised

Surprised, label = 6

```
# Call the plot_image function to display images for class surprised
plot_class_images(data, 6)
```

```
The class [surprised] contains 18020 Images.
Plotting the first 9 images :
```

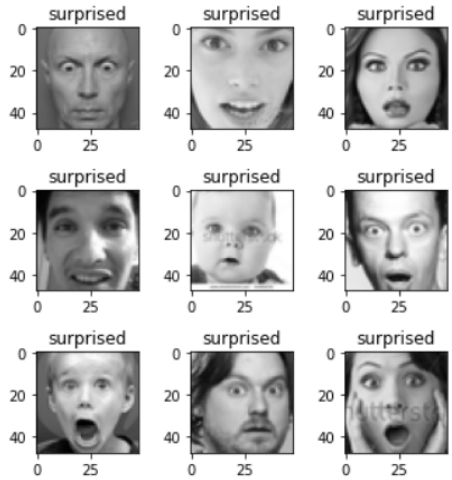


Figure 3.1.7 Surprised images

Some images look very similar, for example some angry images look very similar to the disgusted or sad images. Angry images and disgusted images are kinda similar because the people show and clench their teeth when they are angry or disgusted. Angry and sad images are similar as well because people facepalm when they are angry or sad. These facts could mean that the model will have trouble separating these classes from each other.

Step 7: Plot the frequency distribution of the classes

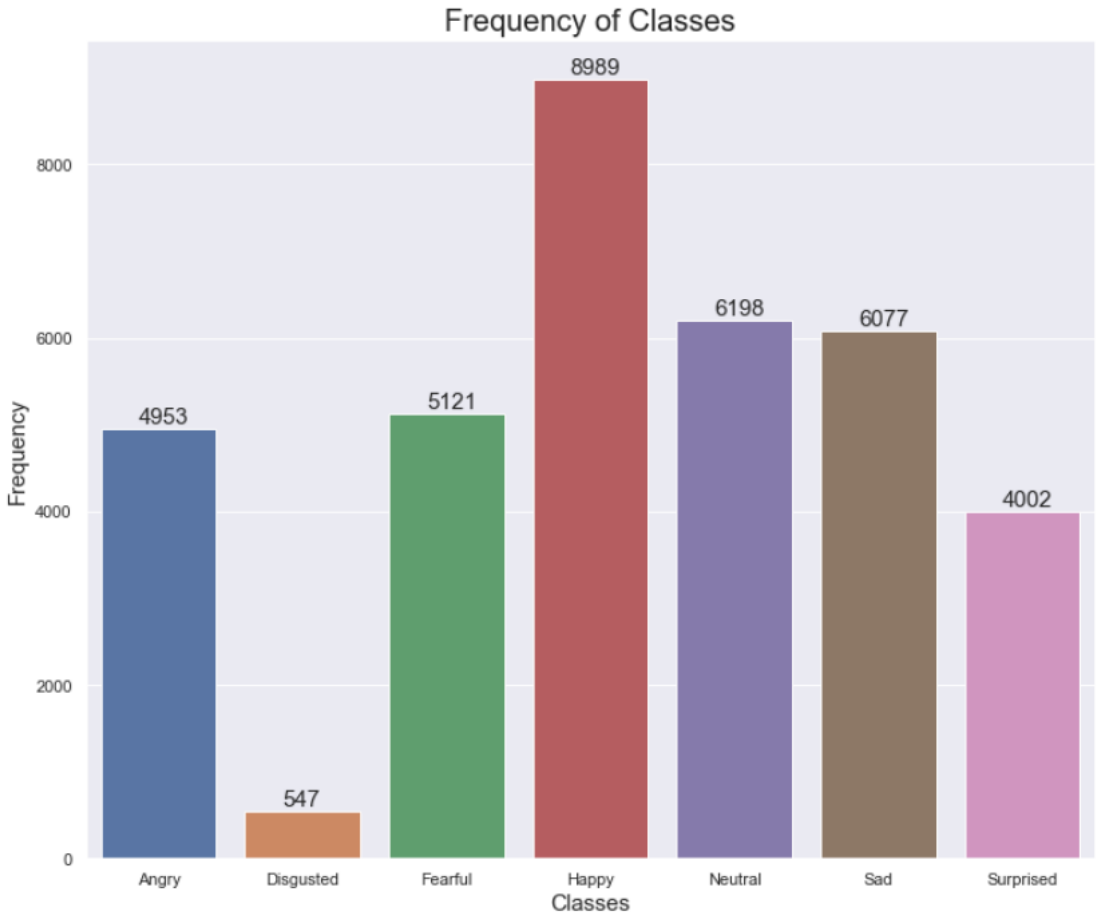


Figure 3.1.8 Class distribution

Figure 3.1.8 shows that our dataset classes are not equally distributed(Imbalanced) as there are significantly less 'Disgusted' images with only 547 examples. Other classes except 'Happy' are quite balanced with approximately 4002 to 6198 examples. Whereas the class 'Happy' has significantly more examples at 8989 examples.

Hypothesis:

H1. Our models will have trouble separating angry images from sad, and disgusted classes as they have similar features.

H2. Our models will have trouble classifying disgusted classes as there are too little examples belonging to that class.

3.2 Image Processing

Step 1: Image Data Generator

```
imheight, imwidth = 48, 48
batch_size = 64
num_class = 7
train_datagen = ImageDataGenerator(rescale = 1./255,
                                   rotation_range=30, fill_mode='nearest',
                                   width_shift_range=[0,5],
                                   height_shift_range=[0,5],
                                   validation_split = 0.2)
test_datagen = ImageDataGenerator(rescale = 1./255)

# train_datagen = ImageDataGenerator(validation_split = 0.3)
# test_datagen = ImageDataGenerator()

train_set = train_datagen.flow_from_directory(
                  train_dir,
                  target_size = (imheight, imwidth),
                  color_mode = 'grayscale',
                  batch_size = batch_size,
                  class_mode = 'categorical', # multiclass
                  subset = 'training', # set as training data
                  shuffle = True)

validation_set = train_datagen.flow_from_directory(
                        train_dir,
                        target_size=(imheight, imwidth),
                        color_mode = 'grayscale',
                        batch_size=batch_size,
                        class_mode = 'categorical',
                        subset='validation',
                        shuffle=False) # set as validation data

test_set = test_datagen.flow_from_directory(
                    test_dir,
                    target_size =(imheight, imwidth),
                    color_mode = 'grayscale',
                    batch_size = batch_size,
                    class_mode ='categorical',
                    shuffle=False)
```

```
Found 22968 images belonging to 7 classes.
Found 5741 images belonging to 7 classes.
Found 7178 images belonging to 7 classes.
```

In this part, we generate images for train_set, validation_set, and test_set using ImageDataGenerator. For the generator in train datagen, rescale(1./255) is done to reduce the pixel to the range of 0-1 value since the original value is too big. Then we also perform the rotation for the train_datagen image randomly from 0 to 30 degrees. Then, we also try to shift the image. For the width of the image, it will be shifted left or right randomly for 0 or 5 pixels while the height of the image will be shifted up or down randomly for 0 or 5 pixels. Lastly, we split the train set and validation set for the radio of 80:20 respectively. As a whole, our dataset is split into train, validation and test set with the ratio of 64:16:20 respectively. Next, the generator for the test set also needs to be rescale to 1/255.

Furthermore, we fetch the images from our train and test folder by calling flow_from_directory function of the ImageDataGenerator class to generate train, validation, and test set. The target size for the image is 48,48 and the colormode for the image that we use in the train, test and validation set is ‘grayscale’. Then the batch size is set to 64, which means that a batch of 64 images will be feedforward in each iteration, and we will have approximately ceil(22968/64) = 359 backpropagation to update the weights of our model in a single epoch. It will make the training time faster. Since our target label is multinomial, we have to use the ‘categorical’ class_mode to get the result. Last but not least, we need to shuffle our data after each epoch for the train set. It is because the gradient estimation is incorrect because of the risk of creating a batch that does not represent the entire dataset. By shuffling the data after each epoch, we will not get "stuck" with too many bad batches. Shuffling is not required for validation and testing.

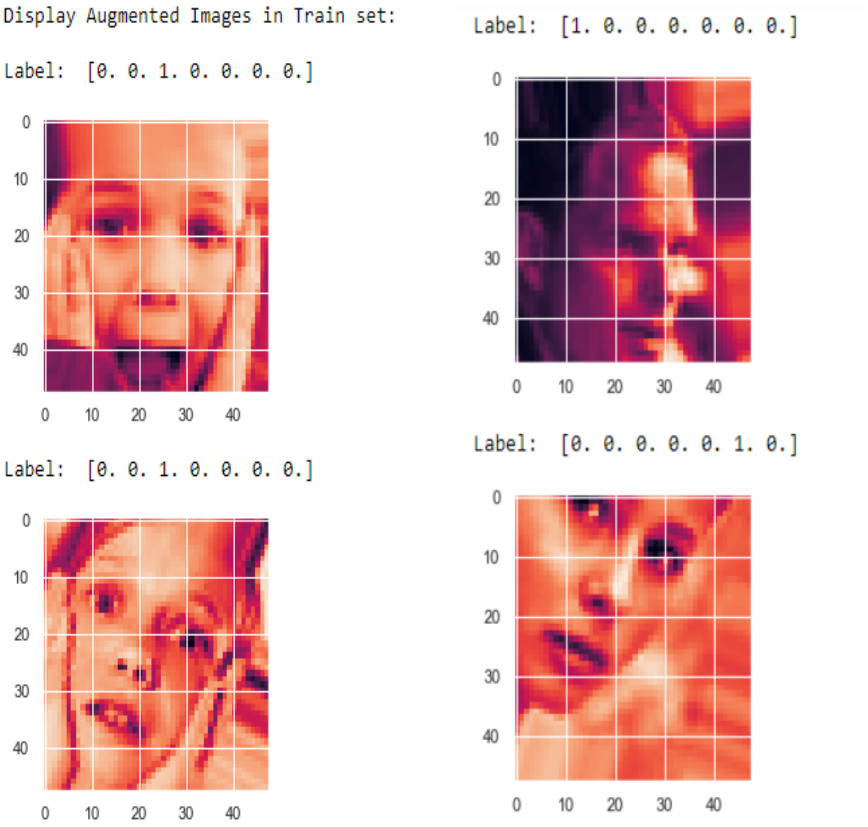


Figure 3.2.1 Augmented Images

Figure 3.2.1 shows that only the first image remains original and the other 3 images have been shifted or rotated. The reason why we need augmentation for the train and validation set is because we need to train our model to recognize our data even if it has been rotated or shifted in order to increase the generalization ability of our model.

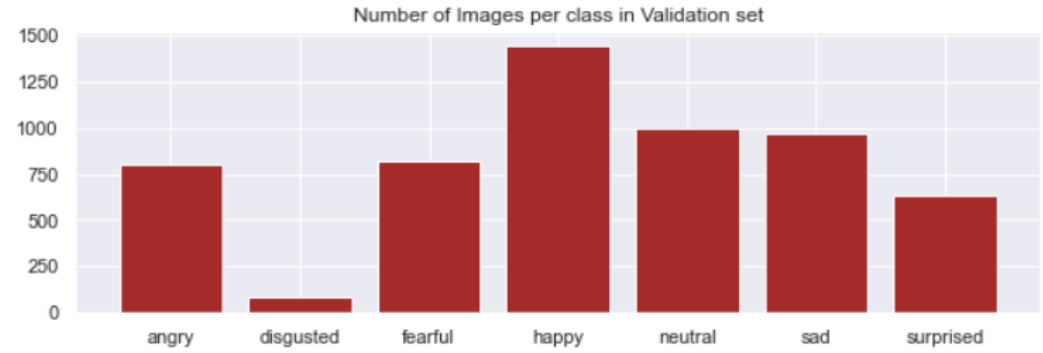
Step 2: Class Distribution of the different sets

- **Train set**


```
[3196, 349, 3278, 5772, 3972, 3864, 2537]
```

Figure 3.2.2 Class distribution of train set

- **Validation Set**




```
[799, 87, 819, 1443, 993, 966, 634]
```

Figure 3.2.3 Class distribution of validation set

- **Test Set**


```
[958, 111, 1024, 1774, 1233, 1247, 831]
```

Figure 3.2.4 Class distribution of test set

Among these 3 distributions, we are most interested in the one in train set. From the train set distribution, it has shown a heavily imbalanced distribution especially in disgusted class and happy class. A huge difference among them will affect the model's Precision and Recall. In order to counter the issue, we will do weight adjusting for the model so that the model can equally penalize under or over sampled classes in the set.

Step 3: Adjusting Class Weights

```
: # Setting class weights to handle imbalanceness of our dataset
import sklearn
from sklearn.utils import compute_class_weight

class_weights = compute_class_weight(class_weight = "balanced",
                                     classes = np.unique(train_set.classes),
                                     y = train_set.classes)
class_weights = dict(zip(np.unique(train_set.classes), class_weights))
class_weights
```

```
: {0: 1.0266404434114071,
 1: 9.401555464592715,
 2: 1.0009587727708533,
 3: 0.5684585684585685,
 4: 0.826068191627104,
 5: 0.8491570541259982,
 6: 1.2933160650937552}
```

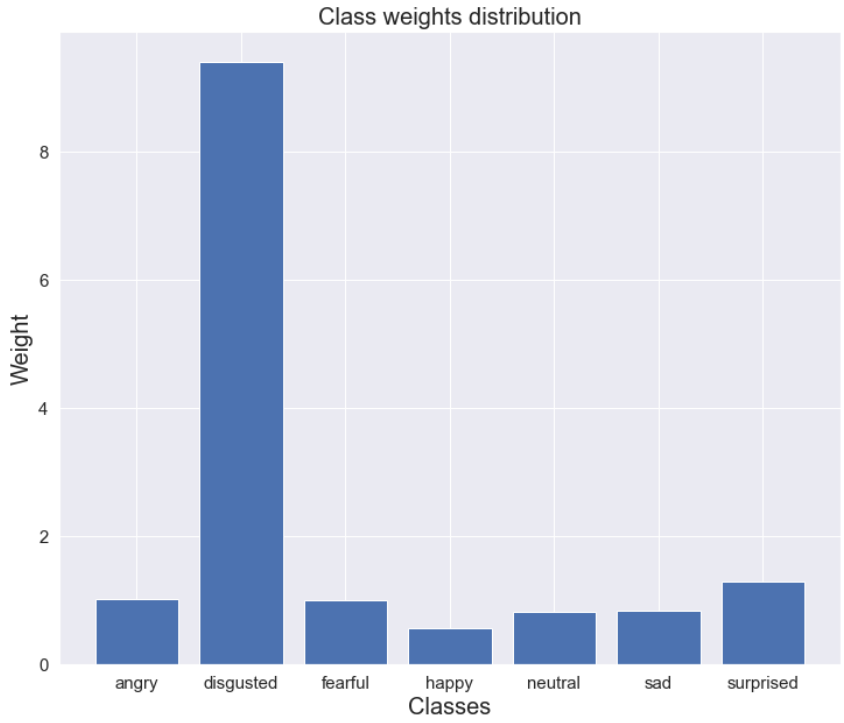


Figure 3.2.5 Class weights

Since our training set is extremely imbalanced, it is imperative that we initialize class weights using scikit-learn.util.class.compute_class_weights to make sure we calibrate the weights for each class. According to the graph above, the weight of the disgusted class will be higher since we have significantly less ‘disgusted’ class and vice versa for the ‘happy’ class. The approach makes sure the weights update is more significant during backpropagation for the disgusted class and less significant for the other classes. The compute_class_weight function automatically assigns weights based on the distribution of the classes.

3.3 Model Building and Classification Error Metrics

This project will involve 5 models to classify our dataset. Which are 1 custom built CNN and 4 pre-trained models using Transfer Learning. The models are Custom CNN, VGG16, ResNet50v2, MobileNetV2, and DenseNet169. We will train all the aforementioned models using mostly the same hyperparameters in the first iteration, which are :

Hyperparameters	Values	Explanation
Batch Size	64	- 64 is generally a good batch size. The weights of the model will only need to be updated $\frac{22968}{64} = 3595$ times per epoch. Making training time much faster.

Optimizer	Adam	- Adam is a variance of gradient descent algorithm used to minimize a loss function. Adam optimizers are very robust as they include a momentum term and have adaptive learning rate. Making it the best first choice for all deep learning projects.
		
		Adam weight updating formula:
		Repeat Until Convergence {
		$\nu_j \leftarrow \eta * \nu_j - \alpha * \nabla_w \sum_1^m L_m(w)$
		$\omega_j \leftarrow \nu_j + \omega_j$
		}
		$\eta * \nu_j$
		Coefficient of Momentum — Retained Gradient

Learning Rate	0.01	- We will experiment with 0.01 LR first and then decide if we want to increase or decrease it in the hyperparameter tuning phase.
Loss Function	Categorical Cross Entropy	- This loss function is used when we are doing a multiclass project.
Epochs	Custom CNN = 60 All Pretrained Models = 40	- Since our custom CNN is trained and built from scratch, is it fair that we let it train for more epochs

		than pretrained models which were trained for millions of epochs.
Hidden Layer Activation Function	ReLU	- ReLU is a great first choice as well for deep learning projects. It gives the model the ability to learn non linear function and prevents exponential growth in computation needed for training.
Output Layer Activation Function	Softmax	- Softmax is an activation function for predicting the target class in multiclass classification. It calculates the probability of each class in the last layer and outputs the maximum probability class. Formula : $$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$
Early Stopping	monitor = ‘val_accuracy’ patience=7 restore_best_weights= True	- This hyperparameter allows the model to stop the training if the model did not improve on the validation set. Validation accuracy of the model will determine the stopping decision. If the model did not improve for 7 epochs, training will stop. This is for time saving purposes. Once training finishes, it will restore the best weights of the model and save it.
Reduce Learning Rate on Plateau	monitor= 'val_loss' factor = 0.2 patience=3 min_delta=0.0001	- This hyperparameter reduces the learning rate if validation loss did not decrease after 3 epochs. It will reduce by a factor of 0.2. This helps the model to get out of local minima.
Checkpoint	monitor = ‘val_accuracy’ save_best_only = True mode = ‘max’	- Checkpoints monitor the models’ validation accuracy and save the best version of the model (max

		accuracy) to a h5 file.
Callbacks	Early stopping Reduce LR on Plateau Checkpoint	- Callbacks is just defined so that we can pass the Early stopping, Reduce Learning Rate on Plateau, and Checkpoint to the models

Basic CNN Model:

Model 1 : Custom CNN

Step 1: Specifying the Architecture of our Custom CNN:

Just like a typical CNN architecture, our model is divided into two parts, the Feature Learning part where we use convolutions, and the Classification part where we use hidden layers for classifying the emotions.

i. Feature Learning Part:

```
# Specifying the model architecture

CNN = Sequential()

# Feature Learning layers
# Block-1
CNN.add(Conv2D(32,(3,3),padding='same',kernel_initializer='he_normal',input_shape=(48,48,1)))
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(Conv2D(32,(3,3),padding='same',kernel_initializer='he_normal',input_shape=(48,48,1)))
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(MaxPooling2D(pool_size=(2,2)))
CNN.add(Dropout(0.2))

# Block-2

CNN.add(Conv2D(64,(3,3),padding='same',kernel_initializer='he_normal'))
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(Conv2D(64,(3,3),padding='same',kernel_initializer='he_normal'))
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(MaxPooling2D(pool_size=(2,2)))
CNN.add(Dropout(0.2))

# Block-3

CNN.add(Conv2D(128,(3,3),padding='same',kernel_initializer='he_normal'))
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(Conv2D(128,(3,3),padding='same',kernel_initializer='he_normal'))
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(MaxPooling2D(pool_size=(2,2)))
CNN.add(Dropout(0.2))

# Block-4

CNN.add(Conv2D(256,(3,3),padding='same',kernel_initializer='he_normal'))
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(Conv2D(256,(3,3),padding='same',kernel_initializer='he_normal'))
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(MaxPooling2D(pool_size=(2,2)))
CNN.add(Dropout(0.2))
```

The feature learning part of our model has 4 blocks of layers performing convolutions and pooling. Each block has 7 layers. The layers used and the sequence are the same for all 4 blocks. The layers in sequence are:

Layer Number	Layer	Parameters
1	2D Convolutional layer	Number of Filters: - Block 1: 32 - Block 2: 62 - Block 3: 128 - Block 4: 256 Filter Size : 3 x 3 Padding : Same Activation: ReLU
2	Batch Normalization	-
3	2D Convolutional layer	Number of Filters: - Block 1: 32 - Block 2: 62 - Block 3: 128 - Block 4: 256 Filter Size : 3 x 3 Padding : Same Activation: ReLU
4	Batch Normalization	-
5	2D Max Pooling	Pool Size : 2 x 2
6	Dropout	Probability : 0.2

Table 3.3.1.1 The layers in each block of Feature Learning part in CNN

The number of filters increases each block because CNN models can learn more complex features by combining the simpler features in the earlier blocks. The first few blocks will be able to learn simple features like edges, corners, and so on. The last block having 256 number of filters means it will learn 256 complex features about the emotions like the shape of the mouth and eyes. Activation function we used is ReLu. Batch Normalization layers help us to reduce training time by re-centering and re-scaling images. Max Pooling layer helps us to find the most

significant pixels in each 2 x 2 matrix of convolution layer, this helps us to remove weak pixels that do not give us much insight about the features of an image. This also reduces the training time.

After the last Feature Learning layer, there will be a Flatten layer to convert the feature matrix into a one-dimensional vector. This is so that we can pass into the fully connected layers for classification.

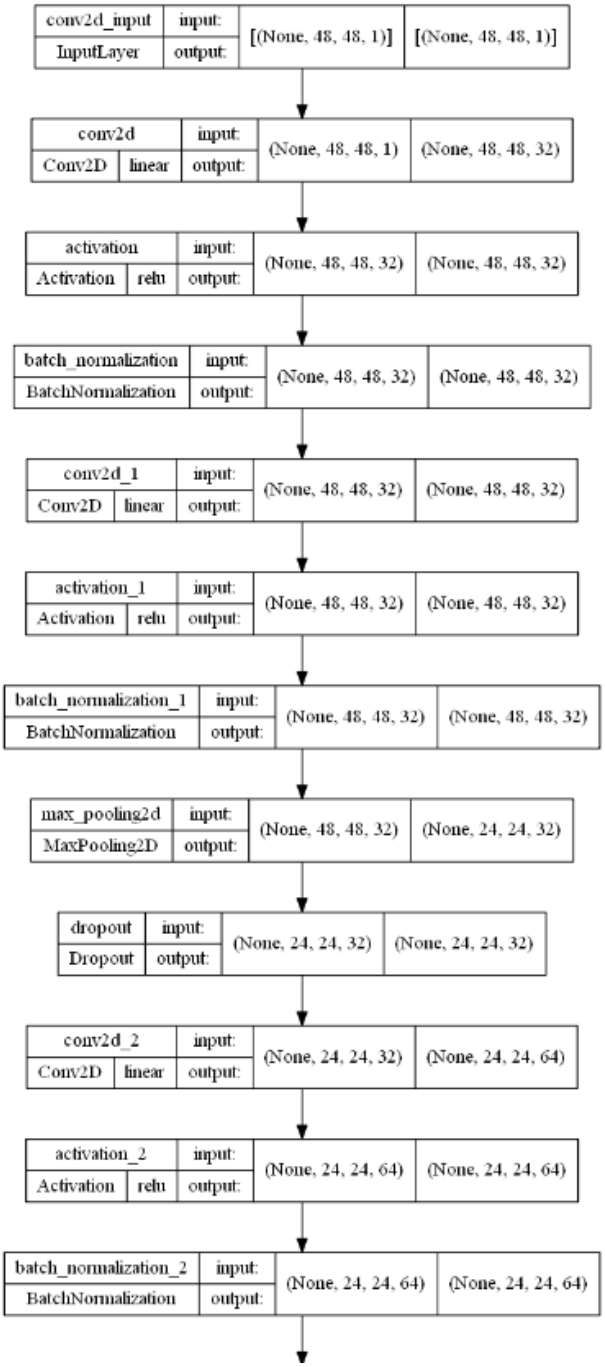
ii. Classification Part

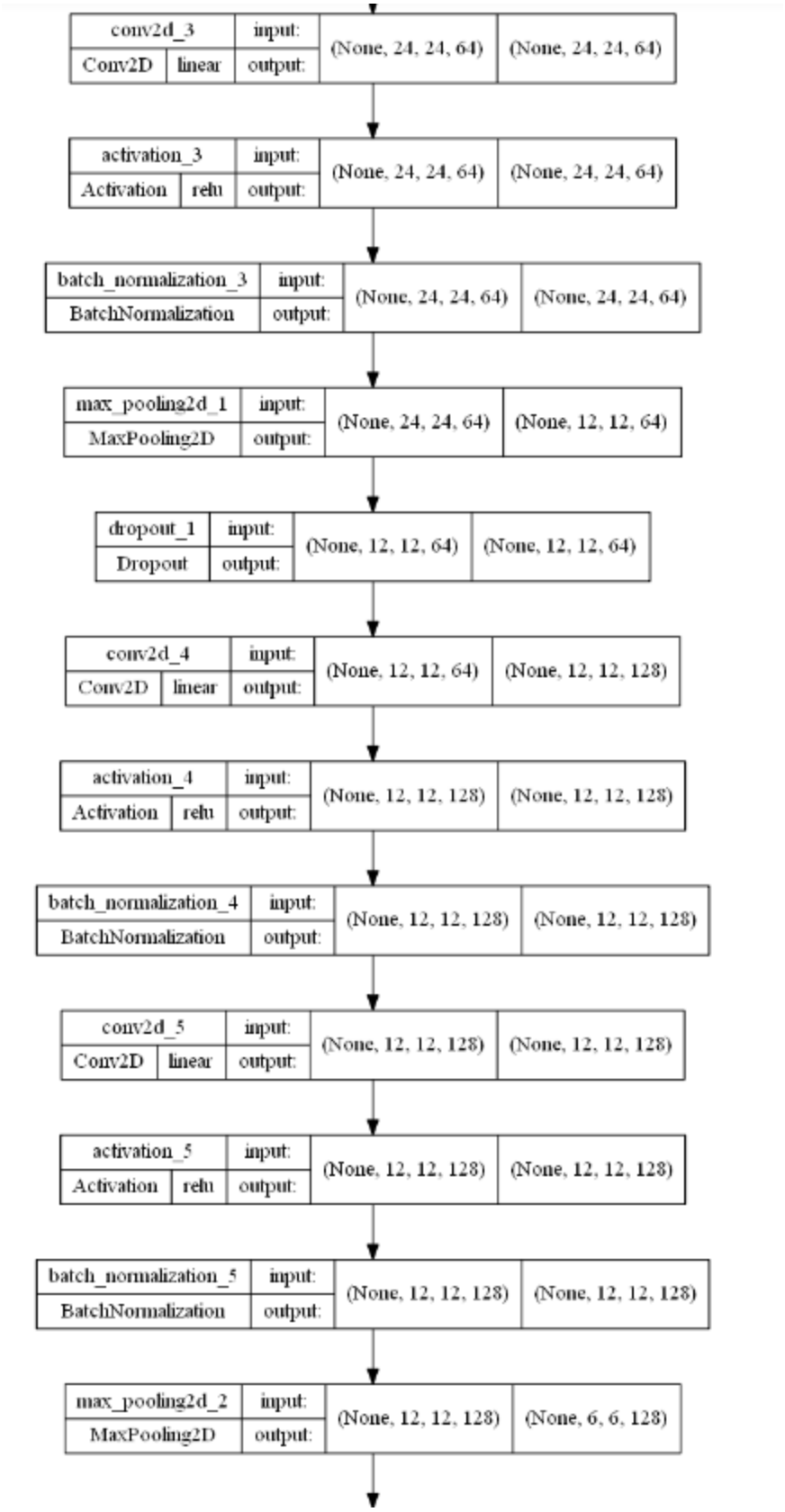
```
# Classification Layers
# Block-1

CNN.add(Flatten())
CNN.add(Dense(128,kernel_initializer='he_normal'))
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(Dropout(0.2))

# Block-2

CNN.add(Dense(128,kernel_initializer='he_normal'))
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(Dropout(0.2))

# Block-3

CNN.add(Dense(7,kernel_initializer='he_normal'))
CNN.add(Activation('softmax')) # Softmax activation since we are doing multiclass classification
```

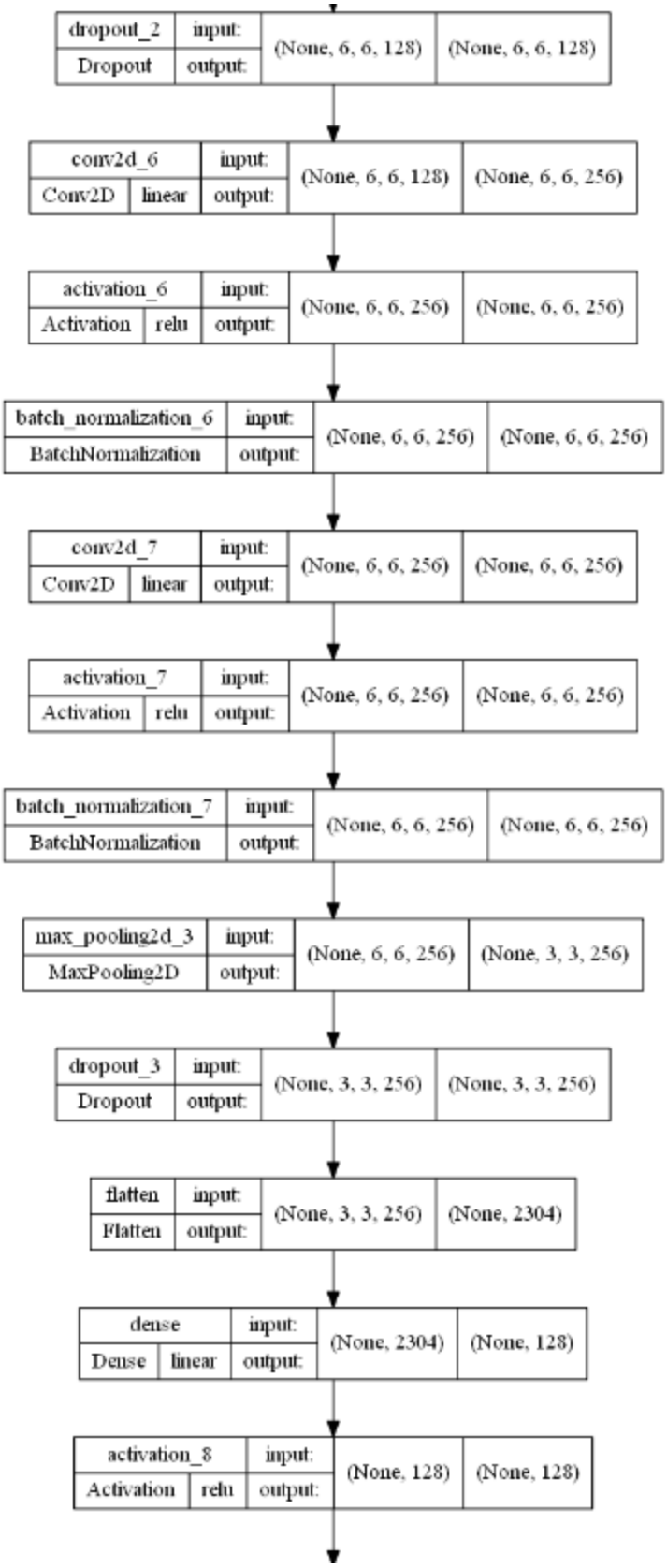
The Classification part is simpler compared to the Feature Learning part. This part consists of 3 blocks. The first 2 blocks are identical, the first layer of these blocks is a Fully Connected Hidden Layer with 128 neurons. The activation function we used is ReLu. Then the output from the activation function is passed into a Batch Normalization layer and subsequently a Dropout layer with 0.2 probability of neurons being dropped.

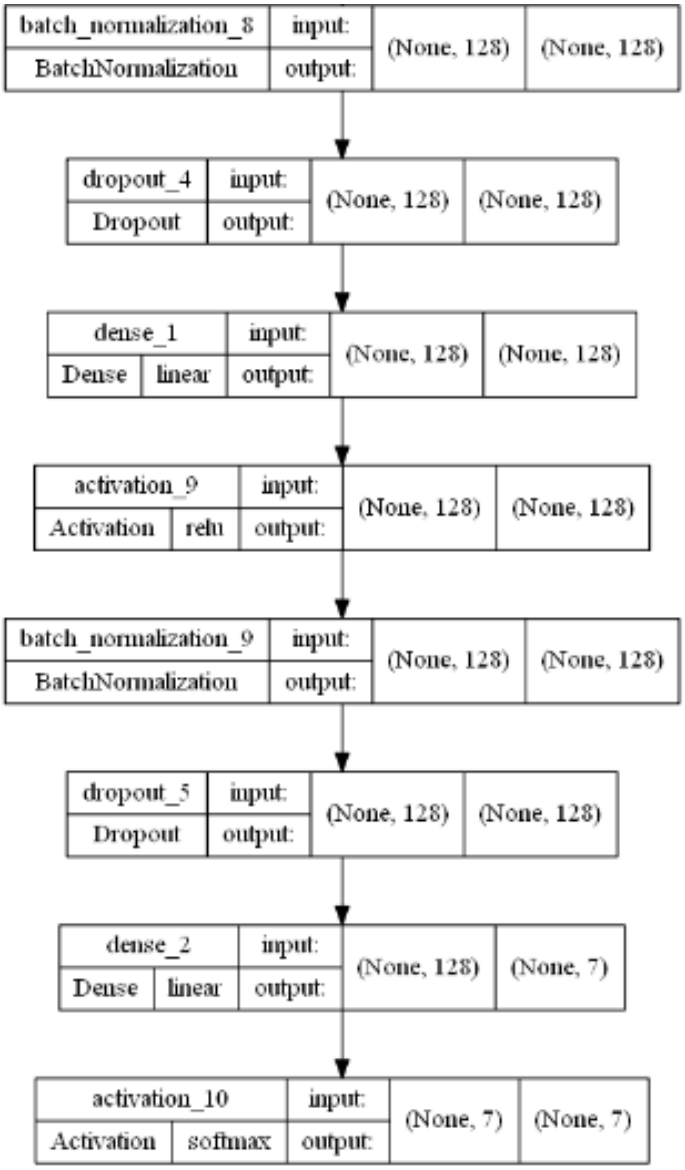
Finally, the last layer is a Dense Layer with 7 neurons, because we have 7 classes in our dataset. Then the activation function is softmax, which is used for multiclass classification.

Summary of our Architecture :




conv2d_3 | input: | (None, 24, 24, 64) | (None, 24, 24, 64)
Conv2D | linear | output:
activation_3 | input: | (None, 24, 24, 64) | (None, 24, 24, 64)
Activation | relu | output:
batch_normalization_3 | input: | (None, 24, 24, 64) | (None, 24, 24, 64)
BatchNormalization | output:
max_pooling2d_1 | input: | (None, 24, 24, 64) | (None, 12, 12, 64)
MaxPooling2D | output:
dropout_1 | input: | (None, 12, 12, 64) | (None, 12, 12, 64)
Dropout | output:
conv2d_4 | input: | (None, 12, 12, 64) | (None, 12, 12, 128)
Conv2D | linear | output:
activation_4 | input: | (None, 12, 12, 128) | (None, 12, 12, 128)
Activation | relu | output:
batch_normalization_4 | input: | (None, 12, 12, 128) | (None, 12, 12, 128)
BatchNormalization | output:
conv2d_5 | input: | (None, 12, 12, 128) | (None, 12, 12, 128)
Conv2D | linear | output:
activation_5 | input: | (None, 12, 12, 128) | (None, 12, 12, 128)
Activation | relu | output:
batch_normalization_5 | input: | (None, 12, 12, 128) | (None, 12, 12, 128)
BatchNormalization | output:
max_pooling2d_2 | input: | (None, 12, 12, 128) | (None, 6, 6, 128)
MaxPooling2D | output:


dropout_2 | input: | (None, 6, 6, 128) | (None, 6, 6, 128)
Dropout | output:
conv2d_6 | input: | (None, 6, 6, 128) | (None, 6, 6, 256)
Conv2D | linear | output:
activation_6 | input: | (None, 6, 6, 256) | (None, 6, 6, 256)
Activation | relu | output:
batch_normalization_6 | input: | (None, 6, 6, 256) | (None, 6, 6, 256)
BatchNormalization | output:
conv2d_7 | input: | (None, 6, 6, 256) | (None, 6, 6, 256)
Conv2D | linear | output:
activation_7 | input: | (None, 6, 6, 256) | (None, 6, 6, 256)
Activation | relu | output:
batch_normalization_7 | input: | (None, 6, 6, 256) | (None, 6, 6, 256)
BatchNormalization | output:
max_pooling2d_3 | input: | (None, 6, 6, 256) | (None, 3, 3, 256)
MaxPooling2D | output:
dropout_3 | input: | (None, 3, 3, 256) | (None, 3, 3, 256)
Dropout | output:
flatten | input: | (None, 3, 3, 256) | (None, 2304)
Flatten | output:
dense | input: | (None, 2304) | (None, 128)
Dense | linear | output:
activation_8 | input: | (None, 128) | (None, 128)
Activation | relu | output:




```
=================================================================
Total params: 1,488,999
Trainable params: 1,486,567
Non-trainable params: 2,432
```

Figure 3.3.1 Summary of Custom CNN Architecture

1,488,999 parameters/weights will be trained, therefore we have defined quite a complex architecture as our dataset is also very complex and hard to learn.

Step 2 : Train our model

```
# Training the model

CNN_checkpoint = ModelCheckpoint("./CNN.h5", monitor='val_accuracy',verbose=1,save_best_only=True, mode='max')
CNN_callbacks = define_callbacks(CNN_checkpoint)

CNN_history = train_model(CNN, train_set, validation_set, test_set, CNN_callbacks, class_weights, epochs=60)
```

We train our model by passing in our defined model, train, validation, and test set, as well as callbacks and class_weights (for handling imbalance) to the train_model function that we have created ourselves. We pass epoch = 60 as the default epoch we have defined is 40, since other models will use 40 epoch.

```
Epoch 53: val_accuracy did not improve from 0.62881

Epoch 53: ReduceLROnPlateau reducing learning rate to 1.2799998785339995e-07.
359/359 [==============================] - 54s 151ms/step - loss: 0.9025 - accuracy: 0.6302 - precision: 0.7583 - recall: 0.485
8 - auc: 0.9183 - val_loss: 1.0000 - val_accuracy: 0.6272 - val_precision: 0.7372 - val_recall: 0.5027 - val_auc: 0.9169 - lr:
6.4000e-07
Epoch 53: early stopping
Time taken to train: 0:44:43.963661

Training scores:
============================
Loss:  0.9009
Accuracy:  0.6577
Precision:  0.7844
Recall:  0.519
AUC:  0.932
============================

Validation scores:
============================
Loss:  0.9994
Accuracy:  0.6288
Precision:  0.739
Recall:  0.501
AUC:  0.917
============================

Test scores:
============================
Loss:  1.0043
Accuracy:  0.6205
Precision:  0.7392
Recall:  0.5035
AUC:  0.9156
============================
```

Figure 3.3.2 CNN Training output

Our custom CNN has trained for 53 epochs as it did not improve anymore after that. The time taken to train it was 44 minutes, as it is a very complex architecture and there are 1.4 million parameters to train. It has achieved a decent result on the training set, but lackluster results on validation and test set. However, the precision recall is very balanced in all three sets because we have passed in the class weights parameter into training. The AUC score is also very high at 0.9156, indicating that the model is able to separate most of the classes based on their features. Therefore, our model may not have ultra high accuracy, but it has great generalization ability.

Step 3 : Plot and Evaluate Classification Metrics

- **CNN: Training vs Validation Accuracy and Loss History**

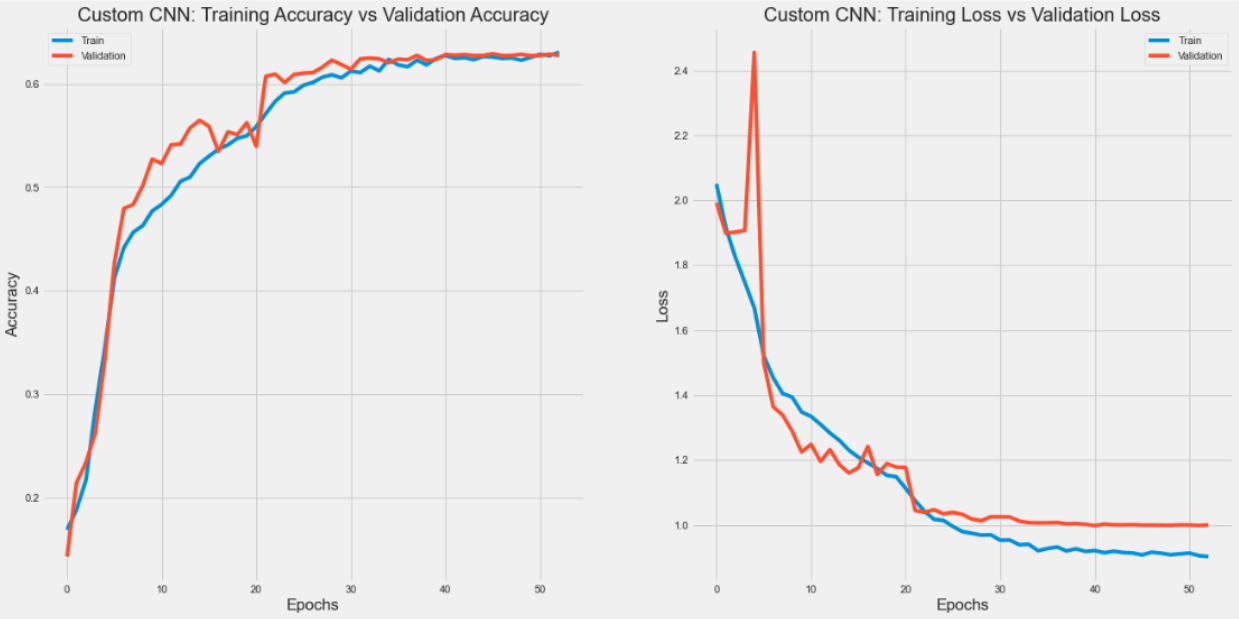


Figure 3.3.3 CNN Training History

In terms of Accuracy, the model hit a plateau at around 30 to 40 epoch on both training and validation set. In terms of Loss, the model hit a local minimum at around 21 epochs on the validation set, and around 35 epochs on the training set. These results tell us that more epochs will not improve the accuracy and loss of the model.

- **CNN: ROC AUC**

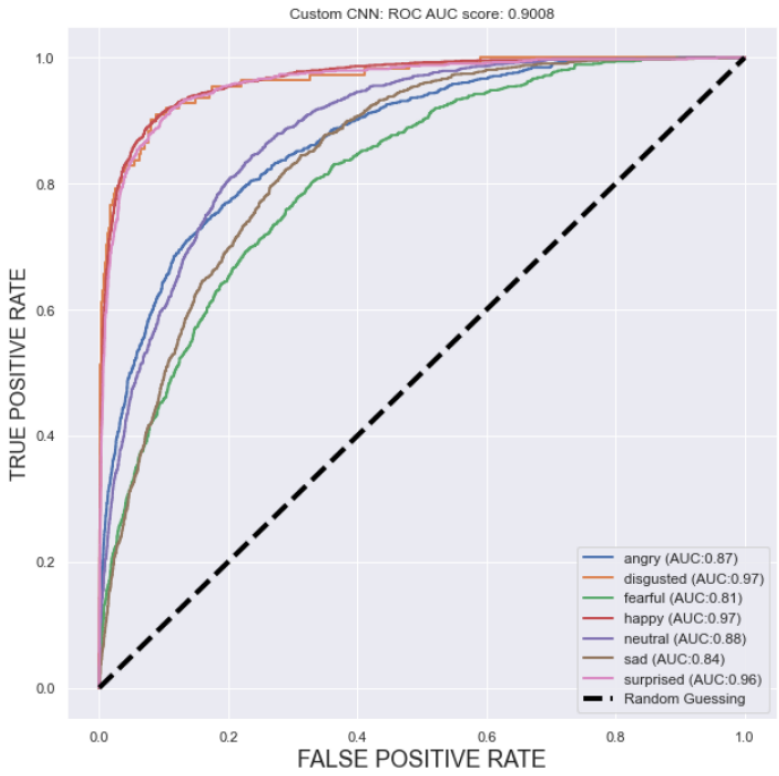


Figure 3.3.4 CNN ROC AUC

ROC AUC plot tells us the degree of separability of the model on each class. It tells us which class can the model separate easily and which one not. False Positive Rate is plotted against the True Positive Rate. We plotted the model's ROC AUC using the test set. From our ROC AUC plot, we can see that happy (red), surprised (pink), and disgusted (orange) classes are easily distinguishable by our model. However, it has some trouble distinguishing between neutral, angry, sad, and fearful in descending order of separability. It is understandable that the model performed this way because some pictures of angry, sad and fearful looks very similar (Refer to Figure 3.1.1 - Figure 3.1.7). Overall, the model can still generalize to the test set well and also handles imbalance very well with 0.9008 AUC

- **CNN : Error by class on Test set**

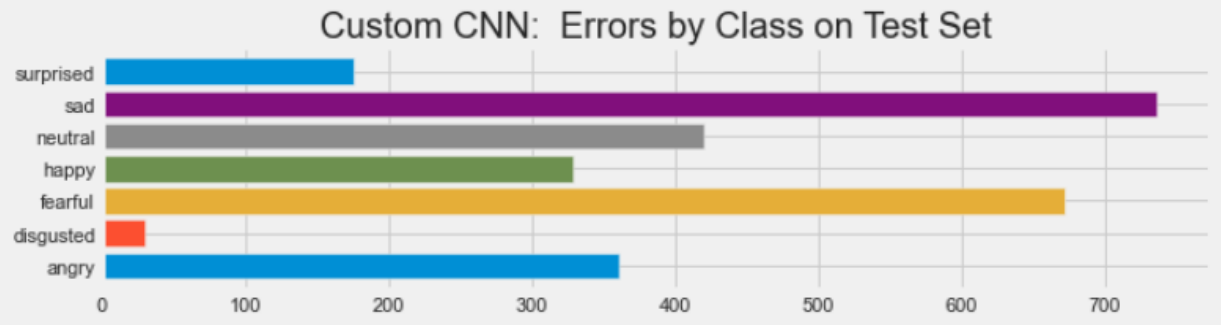


Figure 3.3.5 CNN : Error on test set

Figure 3.3.1.5 shows us how many pictures of each class have the model classified wrongly. As expected, 'sad' and 'fearful' have the most errors as they are the most similar emotions. Disgusted class has low error because of the less number of disgusted images in our dataset.

- **CNN : Confusion Matrix and Classification Report**

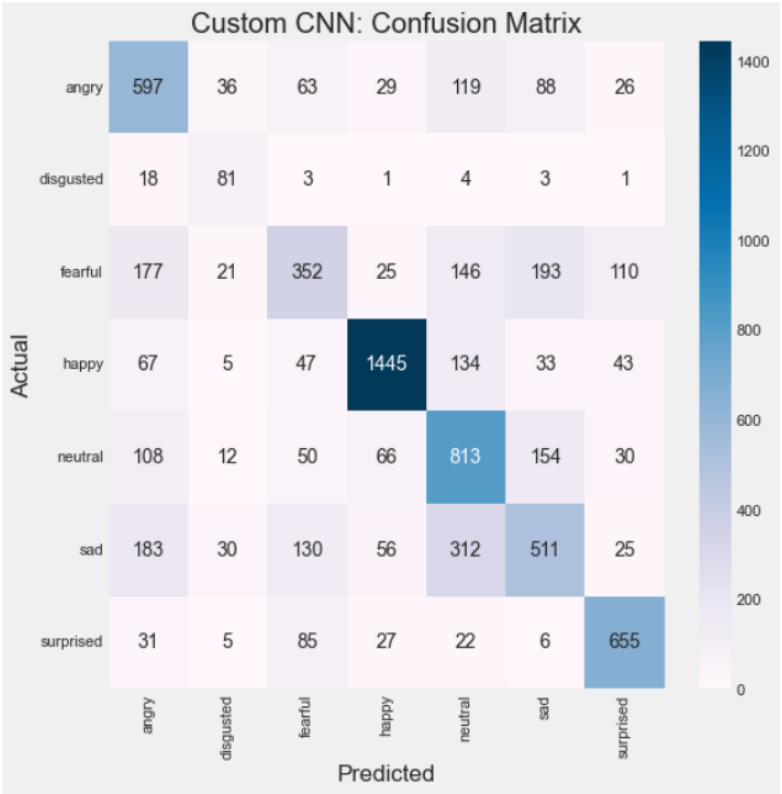


Figure 3.3.6 CNN : Confusion Matrix

Custom CNN: Classification Report on Test Set:

	precision	recall	f1-score	support
angry	0.51	0.62	0.56	958
disgusted	0.43	0.73	0.54	111
fearful	0.48	0.34	0.40	1024
happy	0.88	0.81	0.84	1774
neutral	0.52	0.66	0.58	1233
sad	0.52	0.41	0.46	1247
surprised	0.74	0.79	0.76	831
accuracy			0.62	7178
macro avg	0.58	0.62	0.59	7178
weighted avg	0.62	0.62	0.62	7178

Figure 3.3.7 CNN : Classification Report

Figure 3.3.6 and Figure 3.3.7 shows us the most summarized performance of the model on our test set. It has achieved an accuracy of 62% which can be improved. However, our custom CNN's F1 score on happy and surprise are quite high. For the rest of the classes, it performed quite decently also with more than 0.5 F1 score, except for the sad and fearful class. This means there is not much trade off between precision and recall in our custom CNN, as they are both decent or high in almost all classes.

- **CNN : Test model on a sample image**

```
classify_test(CNN_model, 'fearful')
```

```
D:\Anaconda\envs\GPUenabled\lib\site-packages\keras_preprocessing\imag
ase use color_mode = "grayscale"
  warnings.warn('grayscale is deprecated. Please use '

Predicted Class:  fearful
Probability:  tf.Tensor(
[[0.06772903 0.00265209 0.4319182  0.00495107 0.05831221 0.23324181
  0.2011955 ]], shape=(1, 7), dtype=float32)
```

Figure 3.3.8 CNN : Classify test

We have used a sample image from the fearful class and try to find out if our custom CNN can correctly classify it or not. Unsurprisingly, it was classified correctly! Even though fearful is the hardest class to separate, based on our ROC AUC, it is still able to classify correctly, however, if we look at the probability, it is only 0.4319, which means the model is not completely confident about its classification. The probability of 'sad' and 'surprised' for the image is approximately 0.2, which is quite high. Looking at the person in the image, we can tell it really does look like she's sad(crying) or surprised(screaming).

- **Summary and Insight:**

After training our first custom-built CNN, we have found out that the sad and fearful class is actually very hard to distinguish between each other, as some images look alike. Other than that, our CNN model can perform well in terms of precision and recall, but accuracy can be better. Also, our model does not overfit to the train set as the accuracy for train set and test set are 65% and 62% respectively, only 3% difference. Some key metrics of the model on the test set are :

Metrics	Score
Accuracy	62%
Averaged F1 score of all classes	$\frac{0.56 + 0.54 + 0.40 + 0.84 + 0.58 + 0.46 + 0.76}{7} = 0.59$
AUC score	0.9008

Transfer Learning Models:

We use a technique called transfer learning to train pre-trained models from keras that have been previously trained by other people using stronger hardware and more complex architecture. What we do is cut off the last layer of the pre-trained models and replace it with our own classification layers and train it. The hyperparameters we will be using for all pre-trained models is the same as in Table 3.3.1. The only difference is we will be using 40 instead of 60 epochs for pre-trained models. And we do not use class_weights parameters as it will reduce the accuracy a lot, as these models cannot handle class_weights.

- **Initialize new train, validation, test sets with 3 channels**

```
train_set_3c = train_datagen.flow_from_directory(
                  train_dir,
                  target_size = (imheight, imwidth),
                  batch_size = batch_size,
                  class_mode = 'categorical', # multiclass
                  subset = 'training', # set as training data
                  seed = 125,
                  shuffle = True)

validation_set_3c = valid_datagen.flow_from_directory(
                        train_dir,
                        target_size=(imheight, imwidth),
                        batch_size=batch_size,
                        class_mode = 'categorical',
                        subset='validation',
                        seed = 125,
                        shuffle=False) # set as validation data

test_set_3c = test_datagen.flow_from_directory(
                    test_dir,
                    target_size =(imheight, imwidth),
                    batch_size = batch_size,
                    class_mode ='categorical',
                    shuffle=False)
```

```
Found 22968 images belonging to 7 classes.
Found 5741 images belonging to 7 classes.
Found 7178 images belonging to 7 classes.
```

All transfer learning models require input images to be in 3 channels/colored. Therefore, we created a new train, validation and test set where we do not set color_mode = ‘grayscale’, like for our custom CNN model.

For every pre-trained model’s architecture we use the Global Max Pooling layer to flatten out the output of models’ convolution layers. Then we pass the flattened vector into a fully connected hidden layer with 4096 neurons, with the activation function ReLU. Then we pass the output to a Dropout layer with 0.2 probability of neurons dropped. Finally, our last layer has 7 neurons, and a softmax activation function, just like in our custom CNN.

Model 2 : VGG16

The first pre-trained model that we used in our dataset is VGG16. The most distinctive feature of VGG16 is that, rather than having a large number of hyper-parameters, they focused on having 3x3 filter convolution layers with a stride 1 and always used the same padding and maxpool layer of 2x2 filter stride 2 (Thakur, 2019). Since the name of this model is VGG16, which means that it has 16 weight layers. There are 5 blocks in the architecture and both block1 and block2 are the same. It is because they have 2 convolution layers and 1 max pooling layer. For block3 and 4, they have 3 convolution layers and a max pooling layer while block5 has only 2 convolution layers.

Step 1: Specifying the Architecture of our VGG16 :

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 1, 1, 512)         14714688

 global_max_pooling2d (Globa  (None, 512)              0
 lMaxPooling2D)

 dense_3 (Dense)             (None, 4096)              2101248

 dropout_6 (Dropout)         (None, 4096)              0

 Classifier (Dense)          (None, 7)                 28679

=================================================================
Total params: 16,844,615
Trainable params: 2,129,927
Non-trainable params: 14,714,688
_________________________________________________________________
```

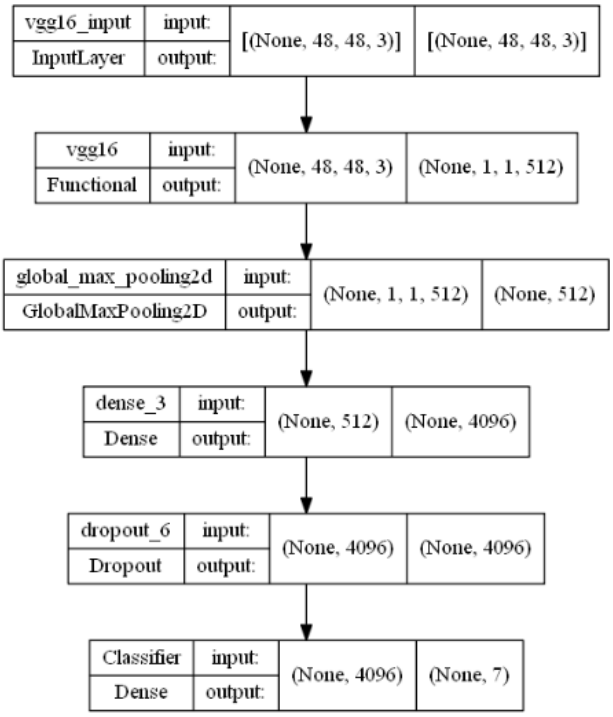


Figure 3.3.9 Summary of VGG16 Architecture

The architecture of the Classification layers of all transfer learning models are the same as explained above. VGG16 has 2,129,927 trainable parameters ready to be trained and 14,714,688 non-trainable parameters which are also known as frozen parameters. By freezing the parameters, the training time can be reduced. The total parameters of VGG16 is 16,844,615 that is defined as a complex architecture as our dataset is also very complex and hard to learn.

Step 2 : Train our VGG16

```
Epoch 40: val_accuracy did not improve from 0.39244
359/359 [==============================] - 65s 182ms/step - loss: 1.5732 - accuracy: 0.3804 - precision: 0.6252 - recall: 0.108
6 - auc: 0.7713 - val_loss: 1.5550 - val_accuracy: 0.3924 - val_precision: 0.6471 - val_recall: 0.1360 - val_auc: 0.7791 - lr:
1.6000e-05
Epoch 40: early stopping
Time taken to train: 0:50:06.902077

Training scores:
==============================
Loss:  1.5431
Accuracy:  0.3974
Precision:  0.6763
Recall:  0.1076
AUC:  0.7832
==============================

Validation scores:
==============================
Loss:  1.5558
Accuracy:  0.3924
Precision:  0.643
Recall:  0.1359
AUC:  0.7788
==============================

Test scores:
==============================
Loss:  1.5411
Accuracy:  0.397
Precision:  0.641
Recall:  0.1406
AUC:  0.7832
==============================
```

Figure 3.3.10 VGG16 training output

The VGG16 has trained for 40 epochs only since it is a pre-trained model that has been trained many times before while the custom CNN needs more epoch to train because it is a custom model that has not trained before. The time taken for training VGG16 was 50 minutes, as it is a very complex architecture with 2.1 million trainable parameters. The accuracy of the test set is lackluster, which is only 39.7%. The precision and recall is very imbalanced in all three sets because our data is imbalanced, which leads to the model ignoring some classes and cannot generalize well to all classes. The AUC score is considered low at 0.7832, indicating that the model is not able to separate well.

Step 3 : Plot and Evaluate Classification Metrics

- **VGG16 : Training vs Validation Accuracy and Loss History**

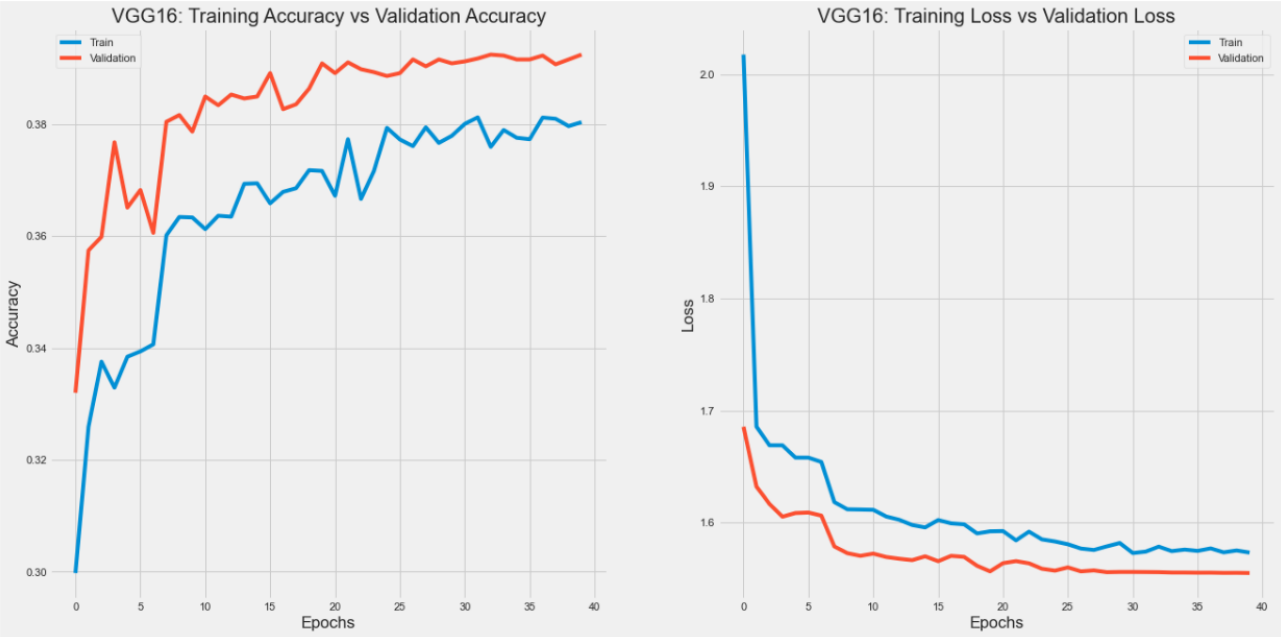


Figure 3.3.11 VGG16 Training History

In terms of Accuracy, the model hit a plateau at around 25 to 30 epoch on both training and validation set. In terms of Loss, the model hit a local minimum at around 20 epochs on both validation set and training set. These results tell us that more epochs will not improve the accuracy and loss of the model. Usually the accuracy of the training set is higher than the validation set, but this model has shown that the accuracy of the validation set is higher. It is because the training set is trained using augmented images that are harder to classify which causes lower accuracy and higher loss.

- **VGG16 : ROC AUC**

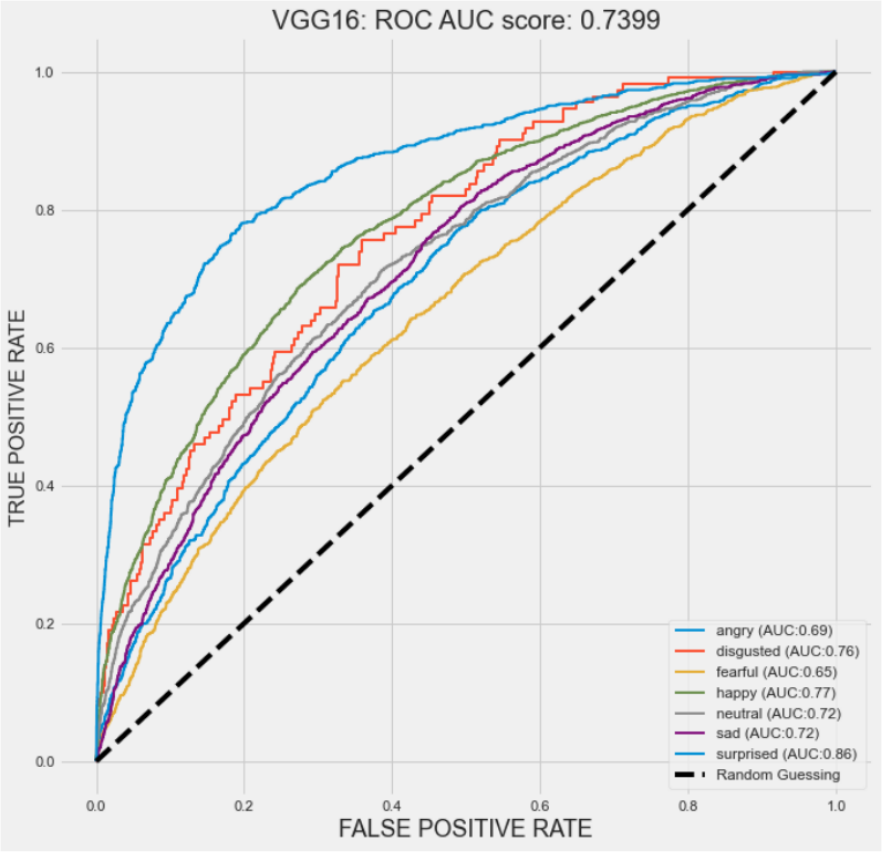


Figure 3.3.12 VGG16 ROC AUC

False Positive Rate is plotted against the True Positive Rate. We plotted the model's ROC AUC using the test set. From our ROC AUC plot, we can see that the surprised (light blue) class is the easiest distinguishable by VGG16 while the fearful(yellow) class is the hardest to distinguish. The rest of the classes have moderate separability with 0.69 to 0.77 AUC score. It is understandable that the model performed this way because the pictures of surprise are different and easiest to distinguish. Overall, the model can still generalize and handles the imbalance to the test set decently with 0.7399 AUC.

- **VGG16 : Error by class on Test set**

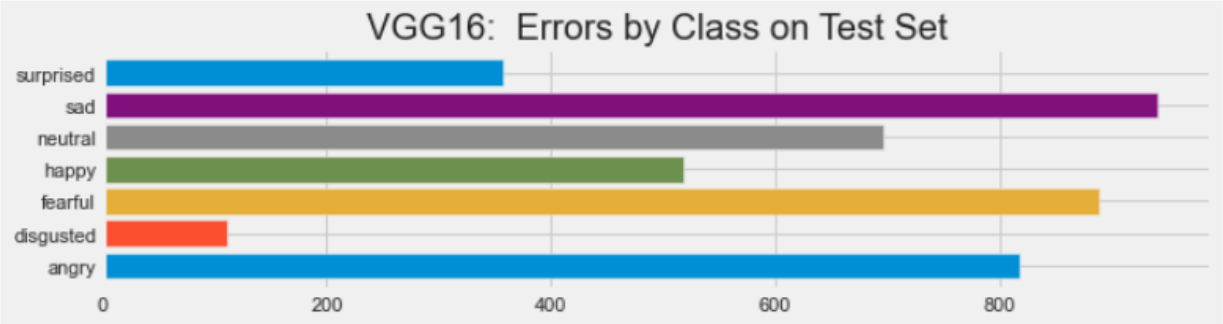


Figure 3.3.13 VGG16 : Error on test set

Figure 3.3.2.5 shows us how many pictures of each class have the model classified wrongly. As expected, 'sad', 'fearful and 'angry have the most errors as they are the most similar emotions. Disgusted class has low error because of the less number of disgusted images in our dataset.

- **VGG16 : Confusion Matrix and Classification Report**

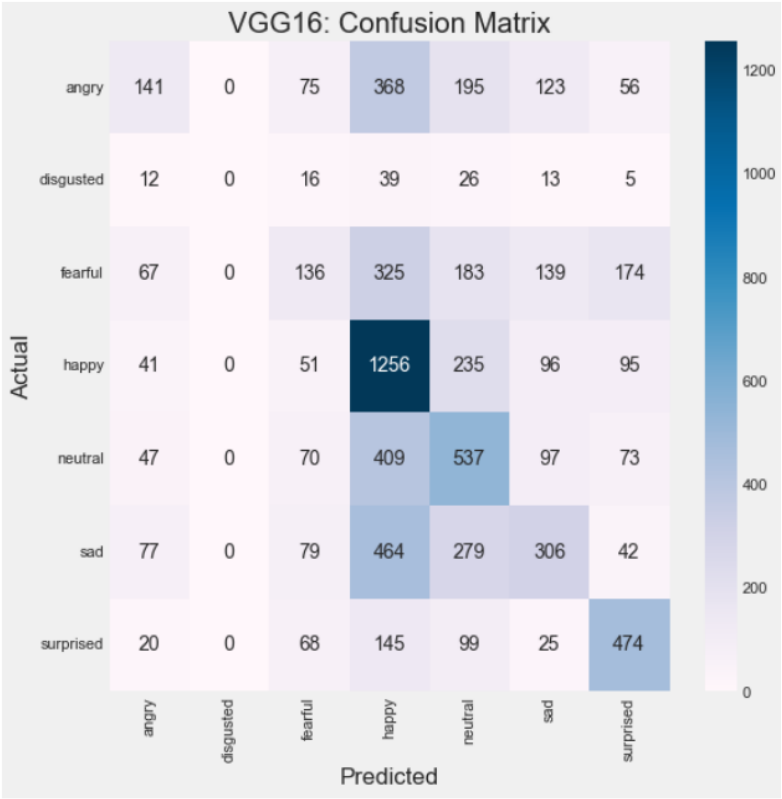


Figure 3.3.14 VGG16:Confusion Matrix

VGG16: Classification Report on Test Set:

	precision	recall	f1-score	support
angry	0.35	0.15	0.21	958
disgusted	0.00	0.00	0.00	111
fearful	0.27	0.13	0.18	1024
happy	0.42	0.71	0.53	1774
neutral	0.35	0.44	0.39	1233
sad	0.38	0.25	0.30	1247
surprised	0.52	0.57	0.54	831
accuracy			0.40	7178
macro avg	0.33	0.32	0.31	7178
weighted avg	0.37	0.40	0.36	7178

Figure 3.3.15 VGG16: Classification Report

Figure 3.3.2.6 and Figure 3.3.2.7 shows us the most summarized performance of the model on our test set. It has achieved an accuracy of 40% which can be improved. However, the VGG16's F1 score for happy and surprised are the highest among all the classes. For the rest of the classes, it performed very badly with less than 0.4 F1 score and the disgusted class performed 0 F1 score. This means that the disgusted class has been completely ignored by the model because the amount of images are too little.

- **VGG16 : Test model on a sample image**

```
# Test VGG on a sample image
classify_test(VGG_model, 'angry', channel)
```

```
Predicted Class:  angry
Probability:  tf.Tensor(
[[0.26920706 0.01372323 0.22600368 0.18250906 0.07429536 0.11308995
  0.12117161]], shape=(1, 7), dtype=float32)
```

Figure 3.3.16 VGG16 : Classify test

We have used a sample image from the angry class and try to find out if VGG16 can correctly classify it or not. Surprisingly, it was classified correctly! Even though angry class is quite hard to separate, based on our ROC AUC, it is still able to classify correctly, however, if we look at the probability, it is only 0.2692, which means the model is not completely confident about its classification. The probability of 'fearful' for the image is slightly lower than angry, which is 0.2260. It is because 'angry' and 'fearful' have similar features in the image. Looking at the person in the image, we could not tell if the person was angry or fearful .

- **Summary & Insight**:

After training the VGG16, we have found out that the angry and fearful class is actually very hard to distinguish between each other, as some images look alike. Other than that, VGG16 performed lackluster in terms of accuracy, precision, recall as well as AUC. Also, VGG16 does not overfit to the train set as the accuracy for train set and test set are the same with 39.7%. Some key metrics of the model on the test set are :

Metrics	Score
Accuracy	39.7%
Averaged F1 score of all classes	$\frac{0.21+0+0.18+0.53+0.39+0.30+0.54}{7} = 0.31$
AUC score	0.7399

Model 3 : ResNet50V2

Next, the next pre-trained model that we used in our dataset is ResNet50V2. ResNet50 is a variant of the ResNet model with 48 convolution layers plus one MaxPool layer and one Average pool layer. There are 3.8 x 10 ^ 9 floating point operations. ResNets was originally applied to image recognition, object localisation and object detection. The framework can also be used for non-computer visual tasks to improve accuracy.

Step 1: Specifying the Architecture of our [model] :

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet50v2 (Functional)      (None, 2, 2, 2048)        23564800

global_max_pooling2d_1 (Glo  (None, 2048)              0
balMaxPooling2D)

dense_4 (Dense)              (None, 4096)              8392704

dropout_7 (Dropout)          (None, 4096)              0

Classifier (Dense)           (None, 7)                 28679

=================================================================
Total params: 31,986,183
Trainable params: 8,421,383
Non-trainable params: 23,564,800
_________________________________________________________________
```

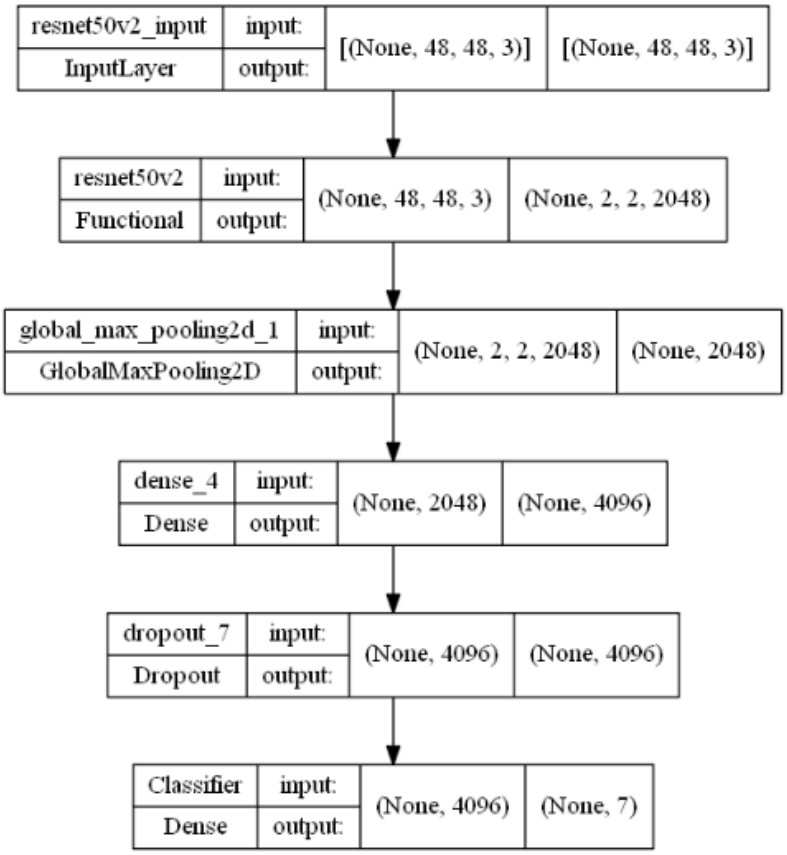


Figure 3.3.17 Summary of ResNest50V2

ResNet50V2 has 8,421,383 trainable parameters ready to be trained and 23,564,800 non-trainable parameters which are also known as frozen parameters. By freezing the parameters, the training time can be reduced. The total parameters of ResNet50V2 is 31,986,183 that is defined as a very complex architecture as our dataset is also very complex and hard to learn.

Step 2: Train ourResNet50V2 :

```
Epoch 40: val_accuracy did not improve from 0.39976
359/359 [==============================] - 62s 172ms/step - loss: 1.6106 - accuracy: 0.3652 - precision: 0.6583 - recall: 0.108
2 - auc: 0.7555 - val_loss: 1.5566 - val_accuracy: 0.3956 - val_precision: 0.7112 - val_recall: 0.1261 - val_auc: 0.7778 - lr:
8.0000e-05
Epoch 40: early stopping
Time taken to train: 0:42:38.593777

Training scores:
============================
Loss:  1.5931
Accuracy:  0.3729
Precision:  0.6872
Recall:  0.0954
AUC:  0.7652
============================

Validation scores:
============================
Loss:  1.5673
Accuracy:  0.3998
Precision:  0.7164
Recall:  0.1197
AUC:  0.7753
============================

Test scores:
============================
Loss:  1.5607
Accuracy:  0.4007
Precision:  0.7267
Recall:  0.1208
AUC:  0.7768
============================
```

Figure 3.3.18 *ResNet50V2 training output*

The ResNet50V2 has also trained for 40 epochs only since it is a pre-trained model that has been trained many times before. The time taken for training ResNet50V2 was 42 minutes, as it is a very complex architecture with 8.4 million trainable parameters. The accuracy of the test set is lackluster, which is only 40%. The precision and recall is very imbalanced in all three sets because our data is imbalanced, which leads to the model ignoring some classes and cannot generalize well to all classes. The AUC score is considered low at 0.7768, indicating that the model is not able to separate well.

Step 3 : Plot and Evaluate Classification Metrics

- **ResNet50V2 : Training vs Validation Accuracy and Loss History**

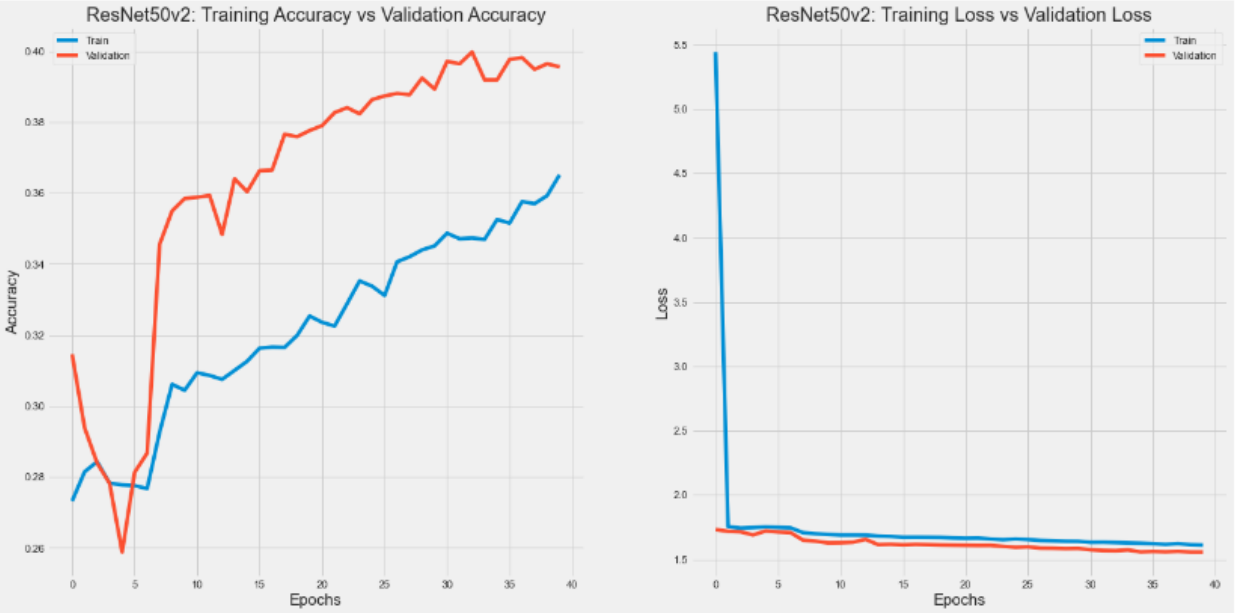


Figure 3.3.19 ResNet50V2 ROC AUC

In terms of Accuracy, the validation set is much higher than the training set due to the underfitting. The accuracy training set is very low, thus it cannot generalize to augmented images. In terms of Loss, we can see that both validation and training set hit the local minimum from about the first epoch. These results tell us that the model has not improved or improved slowly. It is because the optimization algorithm is not able to converge to minimize loss.

ResNet50V2 : ROC AUC

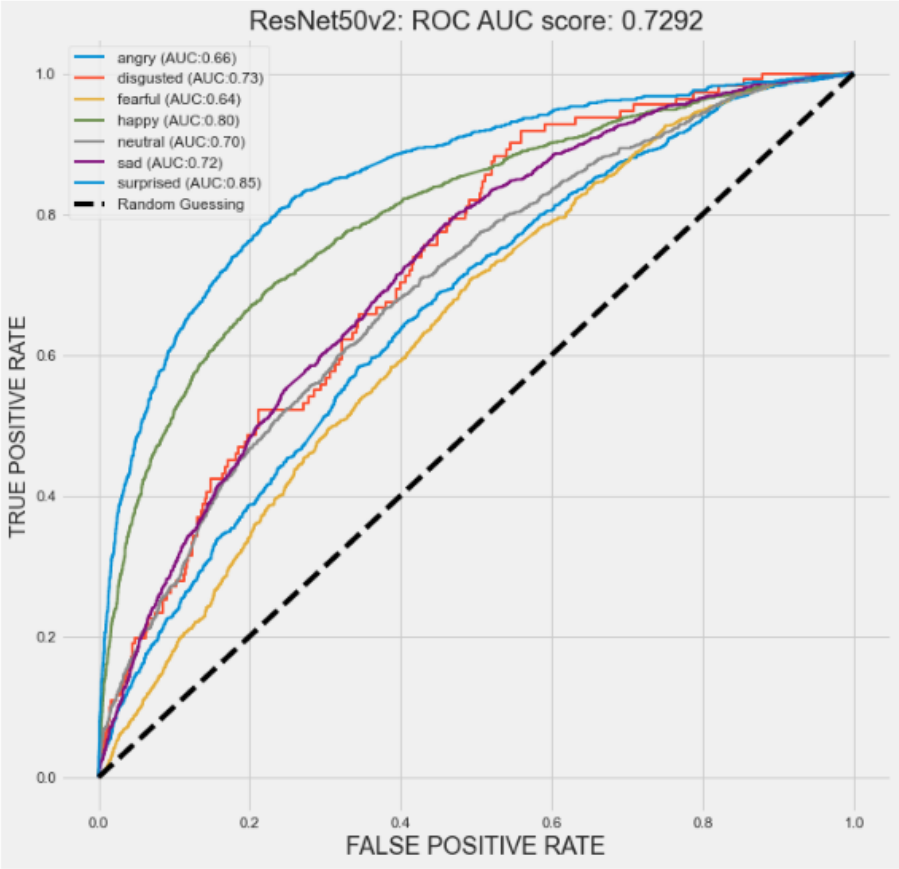


Figure 3.3.20 ResNet50V2 ROC AUC

False Positive Rate is plotted against the True Positive Rate. We plotted the model's ROC AUC using the test set. From our ROC AUC plot, we can see that the surprised (light blue) and happy (green) class are easier to distinguish by ResNet50V2 while the fearful(yellow) class is the hardest to distinguish. The rest of the classes have moderate separability with 0.66 to 0.73 AUC score. It is understandable that the model performed this way because the pictures of surprise are different and easiest to distinguish. Overall, the model can still generalize and handles the imbalance to the test set decently with 0.7292 AUC.

- **ResNet50V2 : Error by class on Test set**

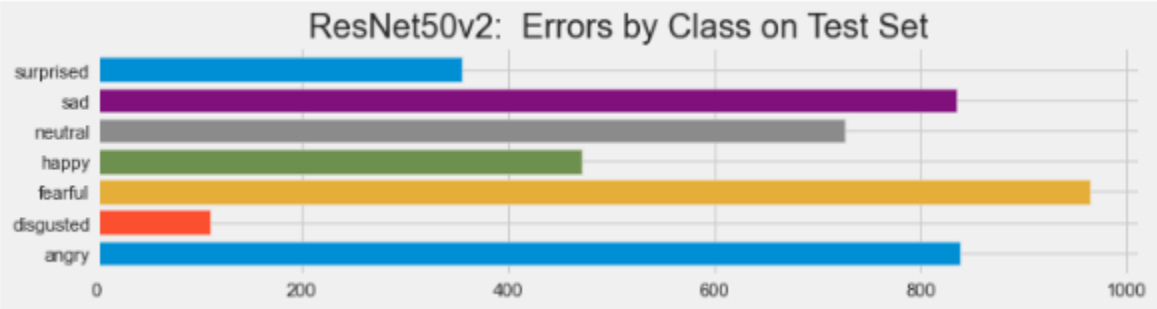


Figure 3.3.21 ResNet50V2 : Error on test set

Figure 3.3.3.5 shows us how many pictures of each class have the model classified wrongly. As expected, 'sad', 'fearful and 'angry have the most errors as they are the most similar emotions. Disgusted class has low error because of the less number of disgusted images in our dataset.

- **ResNet50V2 : Confusion Matrix and Classification Report**

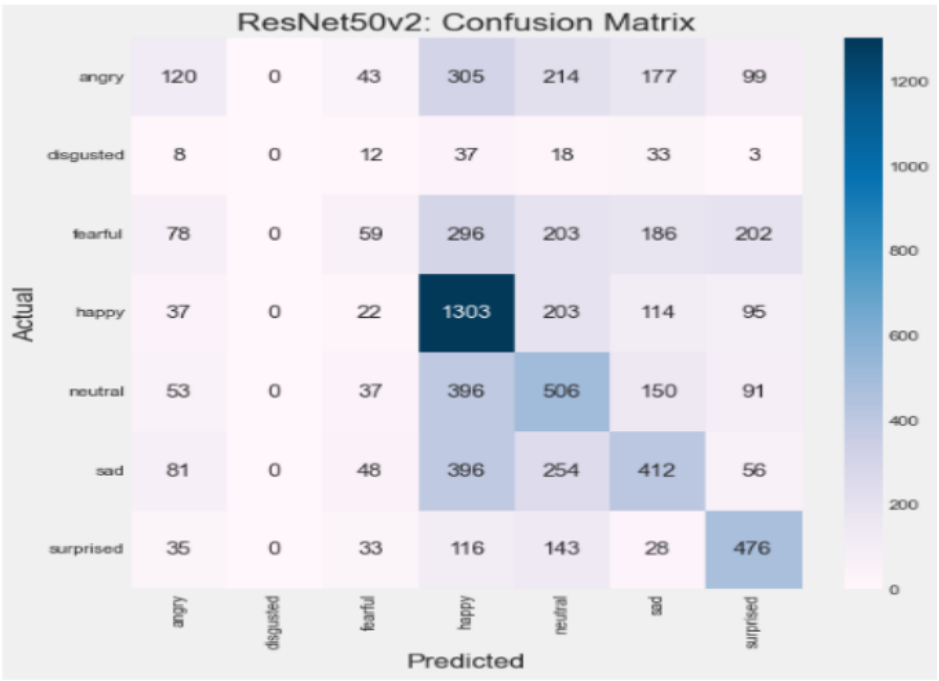


Figure 3.3.22 ResNet50V2:Confusion Matrix

```
ResNet50v2: Classification Report on Test Set:
----------------------
               precision    recall  f1-score   support

       angry       0.29      0.13      0.18       958
   disgusted       0.00      0.00      0.00       111
     fearful       0.23      0.06      0.09      1024
       happy       0.46      0.73      0.56      1774
     neutral       0.33      0.41      0.36      1233
         sad       0.37      0.33      0.35      1247
   surprised       0.47      0.57      0.51       831

    accuracy                           0.40      7178
   macro avg       0.31      0.32      0.29      7178
weighted avg       0.36      0.40      0.36      7178
```

Figure 3.3.23 ResNet50V2: Classification Report

Figure 3.3.3.6 and Figure 3.3.3.7 shows us the most summarized performance of the model on our test set. It has achieved an accuracy of 40% which can be improved. However, the ResNet50V2's F1 score for happy and surprised are the highest among all the classes. For the rest of the classes, it performed very badly with less than 0.4 F1 score and the disgusted class performed 0 F1 score. This means that the disgusted class has also been completely ignored by the model because the amount of images are too little.

- **ResNet50V2: Test model on a sample image**

```
# Test ResNet on a sample image
classify_test(ResNet_model, 'happy', channel)
```

```
Predicted Class:  happy
Probability:  tf.Tensor(
[[0.11736386 0.01416186 0.11171259 0.3623833  0.15831617 0.18438368
  0.05167856]], shape=(1, 7), dtype=float32)
```

Figure 3.3.24 ResNet50V2 : Classify test

We have used a sample image from the happy class and try to find out if ResNet50V2 can correctly classify it or not. Unsurprisingly, it was classified correctly because happy is easier to separate. Based on our ROC AUC, it is still able to classify correctly, however, if we look at the probability, it is only 0.3624, which means the model is not completely confident about its classification. The probability of other classes is quite low compared to happy since a happy image does not have much similarity with other classes' images. Looking at the person in the image, we could tell if the person was happy with a smiling face.

- **Summary and Insight:**

After training the ResNet50V2, we have found out that the happy class is actually easy to distinguish, as the images do not look alike with other classes. Other than that, ResNet50V2 performed lackluster in terms of accuracy, precision, recall as well as AUC. Also, ResNet50V2 improves very slowly. Some key metrics of the model on the test set are :

Metrics	Score
Accuracy	40%
Averaged F1 score of all classes	$\frac{0.18 + 0 + 0.09 + 0.56 + 0.36 + 0.35 + 0.51}{7} = 0.29$
AUC score	0.7292

Model 4 : MobileNetV2

MobileNetV2 is a convolutional neural network architecture aimed at improving performance on mobile devices. It is based on an inverted residual structure with residual connections between the bottleneck layers. Inverted residual is many residual blocks connecting the beginning and end of the convolution block with a skip connection. Adding these 2 states gives the network access to previous activations that have not changed in the convolution block. This approach has proven essential for building very deep networks.

Step 1: Specifying the Architecture of our MobileNet V2 :

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
mobilenetv2_1.00_224 (Funct  (None, 2, 2, 1280)       2257984
ional)

global_max_pooling2d_2 (Glo  (None, 1280)             0
balMaxPooling2D)

dense_5 (Dense)              (None, 4096)              5246976

dropout_8 (Dropout)          (None, 4096)              0

Classifier (Dense)           (None, 7)                 28679

=================================================================
Total params: 7,533,639
Trainable params: 5,275,655
Non-trainable params: 2,257,984
_________________________________________________________________
```

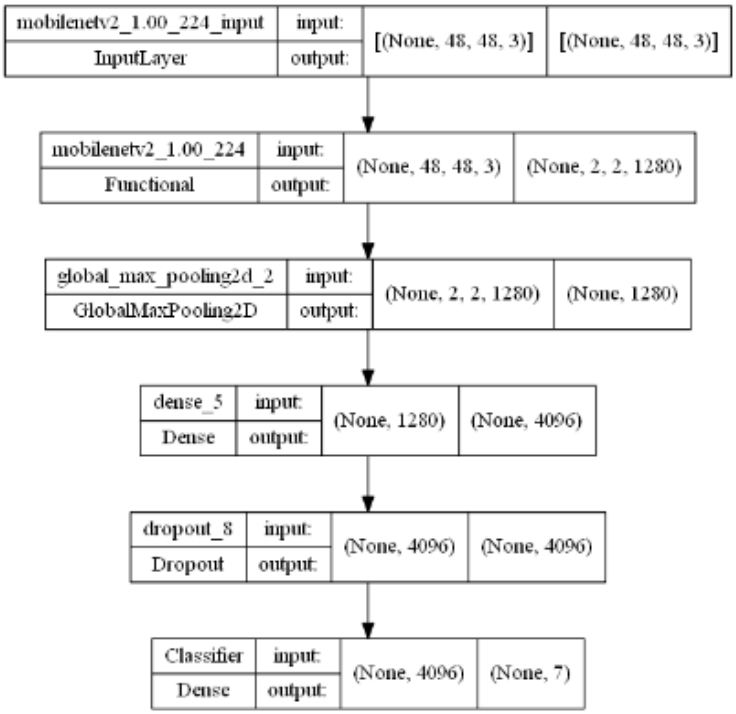


Figure 3.3.25 Summary of MobileNetV2 Architecture

There's a total of 7,533,639 parameters/weight in this architecture. From this, we can know that 5,275,655 trainable parameters and 2,257,984 non-trainable parameters A.K.A. frozen

parameters. By quarter of the session, it can reduce the training time. MobileNetV2 has complicated architecture while our dataset also shows that it is hard to learn and complex.

Step 2: Train our MobileNetV2 :

```
#Compile, train and evaluate model

MobileNet_checkpoint = ModelCheckpoint("./MobileNet.h5", monitor='val_accuracy',verbose=1,save_best_only=True, mode='max')
MobileNet_callbacks = define_callbacks(MobileNet_checkpoint)

MobileNet_history = train_model(MobileNet_full, train_set_3c, validation_set_3c, test_set_3c, MobileNet_callbacks, class_weights
                                , epochs=40)
```

We train our model by calling the train model function with our defined mode, train, validation, and test sets, as well as callbacks and class weights (for handling imbalance). We choose epoch = 40 as the default epoch.

```
Epoch 34: val_accuracy did not improve from 0.36387

Epoch 34: ReduceLROnPlateau reducing learning rate to 3.1999996628122896e-06.
359/359 [==============================] - 56s 154ms/step - loss: 1.6266 - accuracy: 0.3493 - precision_1: 0.6320 - recall_1:
0.0876 - auc_1: 0.7474 - val_loss: 1.6131 - val_accuracy: 0.3620 - val_precision_1: 0.6094 - val_recall_1: 0.1223 - val_auc_1:
0.7546 - lr: 1.6000e-05
Epoch 34: early stopping
Time taken to train: 0:29:29.511971

Training scores:
============================
Loss:  1.6023
Accuracy:  0.3732
Precision:  0.6719
Recall:  0.079
AUC:  0.7599
============================

Validation scores:
============================
Loss:  1.6136
Accuracy:  0.3639
Precision:  0.614
Recall:  0.1252
AUC:  0.7543
============================

Test scores:
============================
Loss:  1.6099
Accuracy:  0.3649
Precision:  0.6231
Recall:  0.123
AUC:  0.7548
============================
```

Figure 3.3.26 MobileNetV2 Training Output

Our custom MobileNetV2 only trained for 34 epochs before it stopped improving. It tooks 29 minutes to train it because it has a very complicated design with 1.4 million parameters to train. On the training set, it performed admirably, but on the validation and test sets, it performed poorly. However, due to the high precision and low accuracy, the system shows a weak recall among all three sets. As a result, MobileNetV2 has shown poor performance for our dataset.

Step 3 : Plot and Evaluate Classification Metrics

- **MobileNet V2: Training vs Validation Accuracy and Loss History**

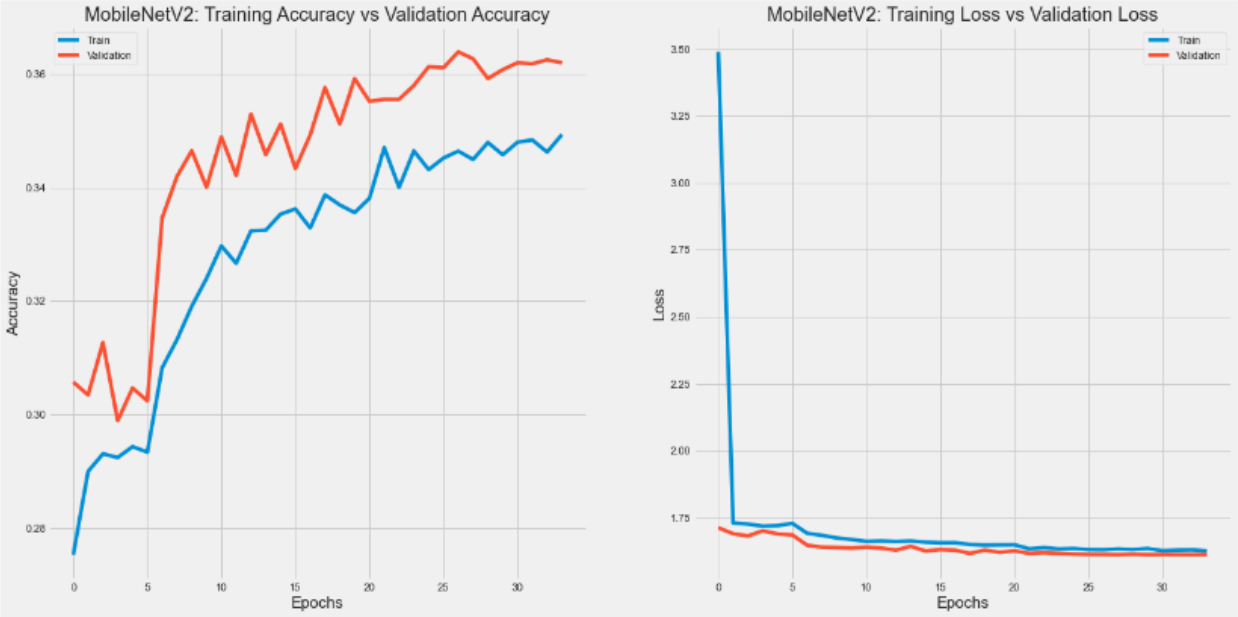


Figure 3.3.27 MobileNetV2 Training History

In terms of accuracy, the model doesn't hit a plateau as they both keep improving although the maximum of epoch is 40. On the other hand, in terms of loss, the train set hit a local minimum at approximately 1 epoch, while the validation doesn't improve since the start of the epoch. These results tell us that this model doesn't become better and raises slowly. The reason for this is that the optimization algorithm is unable to converge in order to reduce loss.

- **MobileNetV2 : ROC AUC**

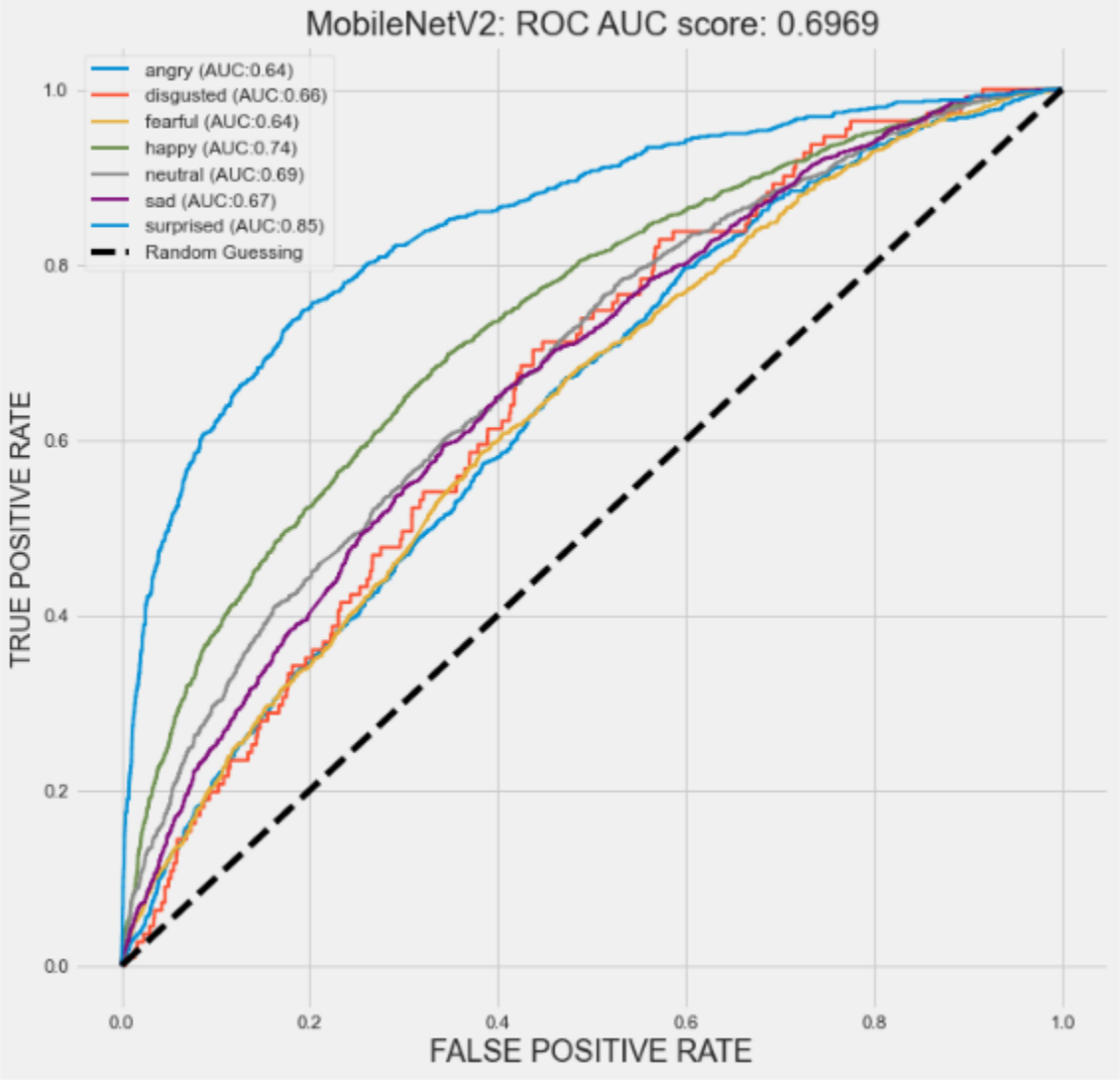


Figure 3.3.28 MobileNet V2 ROC AUC

ROC and AUC plot tells how much the model is capable of distinguishing between classes. From our ROC AUC plot, we can see that surprise(Light Blue) and Happy (Dark Green) are distinguishable easily by our model. Even so, other models such as neutral, sad, disgusted, angry and fearful dataset in descending order based on its separability. This shows that happy emotion and surprise emotion are slightly hard to classify due to its similarity. Overall, this model is still able to generalize to the test set well and control the imbalance well due to its 0.6969 AUC which is above average.

- **MobileNet V2 : Error by class on Test set**

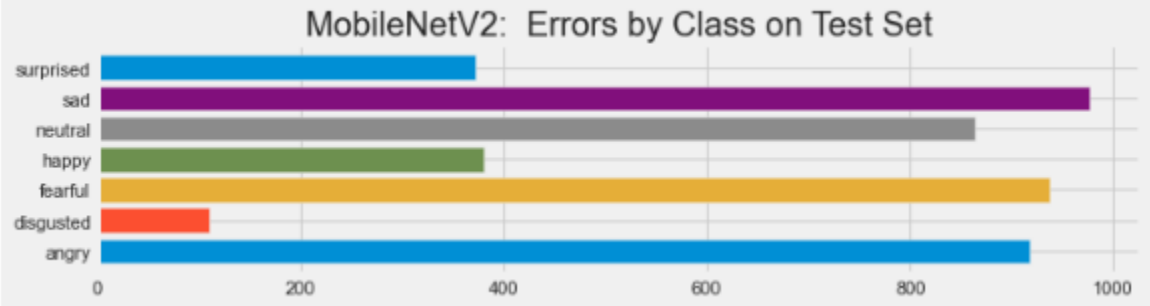


Figure 3.3.29 MobileNetV2: Error on test set

Figure above explains how many photos in each class had the model incorrectly classified. As what we can see from few other models, the sad class and disgusted class have the most errors as expected. What we didn't expect is that the neutral and the angry class are having a high error too in this model. This is because the model fails to classify the neutral and angry emotion from the image.

- **MobileNet V2 : Confusion Matrix and Classification Report**

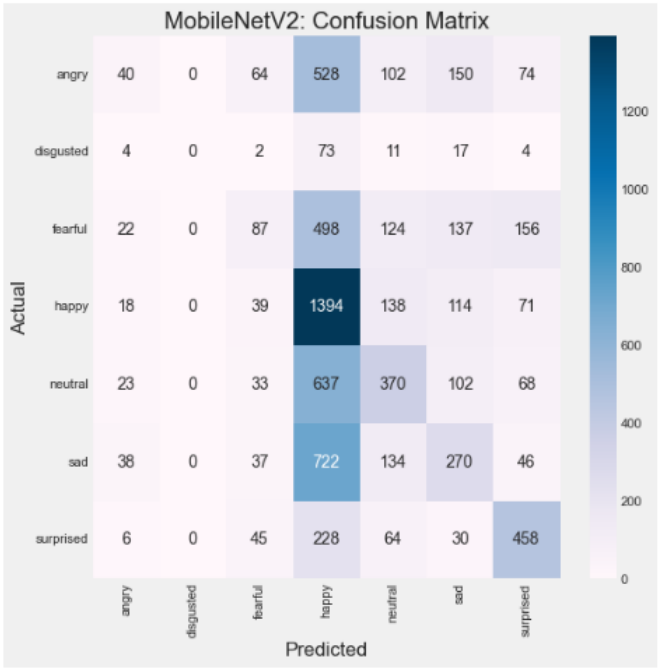


Figure 3.3.30 MobileNet V2: Confusion Matrix

MobileNetV2: Classification Report on Test Set:

	precision	recall	f1-score	support
angry	0.26	0.04	0.07	958
disgusted	0.00	0.00	0.00	111
fearful	0.28	0.08	0.13	1024
happy	0.34	0.79	0.48	1774
neutral	0.39	0.30	0.34	1233
sad	0.33	0.22	0.26	1247
surprised	0.52	0.55	0.54	831
accuracy			0.36	7178
macro avg	0.30	0.28	0.26	7178
weighted avg	0.35	0.36	0.31	7178

Figure 3.3.31 MobileNet V2: Classification Report

The figure 3.3.4.6 and figure 3.3.4.7 indicates a performance of the model evaluation metric for our model MobileNet V2. From the figure above, we can see that there's an accuracy of 36% which can be improved more. In our scenario, F1 scores for the happy and surprised class are among all of the other classes which are expected. But, the outcome of the f1-score for disgusted hits 0 followed by angry (0.07), fearful (0.13), sad (0.26) and neutral (0.34). This means that the disgusted class has also been left out due to its amount of images being too little.

- **MobileNet V2 : Test model on a sample image**

```
# Test MobileNet on a sample image

classify_test(MobileNet_model, 'disgusted', channel)
```

```
Predicted Class:  happy
Probability:  tf.Tensor(
[[0.18760392 0.01991873 0.14761579 0.202874   0.18760775 0.18829387
  0.06608592]], shape=(1, 7), dtype=float32)
```

Figure 3.3.32 MobileNet V2 : Classify test

In this process, we have used an image from disgusted to test whether MobileNet V2 is able to identify the emotion correctly. Unfortunately, it cannot identify disgusted emotion. Based on our Classification Matrix and Classification Report, the precision and accuracy of the disgusted class are 0 due to too little data compared to others. Thus, the model is confused between the happy class and disgusted class.

- **Summary & Insight:**

After training the MobileNet V2, we have found out that the disgusted class is hard to classify as it has a limited amount of data. This scenario happens when the model is confused between different classes or it is hard to separate between classes. Moreover, MobileNet V2 has performed inconsistency in terms of accuracy, precision, recall as well as AUC. Besides that, MobileNet V2 does not fit the train set as the accuracy for train set and test set are the same with 36%. Tables below shows the key metrics of the model on the test set are:

Metrics	Score
Accuracy	36%
Averaged F1 score of all classes	$\frac{0.07+0+0.13+0.48+0.34+0.26+0.54}{7} = 0.26$
AUC score	0.6969

Model 5 : DenseNet169

The name DenseNet comes from Densely Connected Convolutional Network in the fact that each layer in a DenseNet design is connected to every other layer. L(L+1)/2 direct connections exist for L layers. DenseNets can make the connectivity pattern introduced in earlier architectures easier to understand.

Step 1: Specifying the Architecture of our DenseNet 169 :

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 densenet169 (Functional)    (None, 1, 1, 1664)        12642880

 global_max_pooling2d_3 (Glo  (None, 1664)             0
 balMaxPooling2D)

 dense_6 (Dense)             (None, 4096)              6819840

 dropout_9 (Dropout)         (None, 4096)              0

 Classifier (Dense)          (None, 7)                 28679

=================================================================
Total params: 19,491,399
Trainable params: 6,848,519
Non-trainable params: 12,642,880
_________________________________________________________________
```

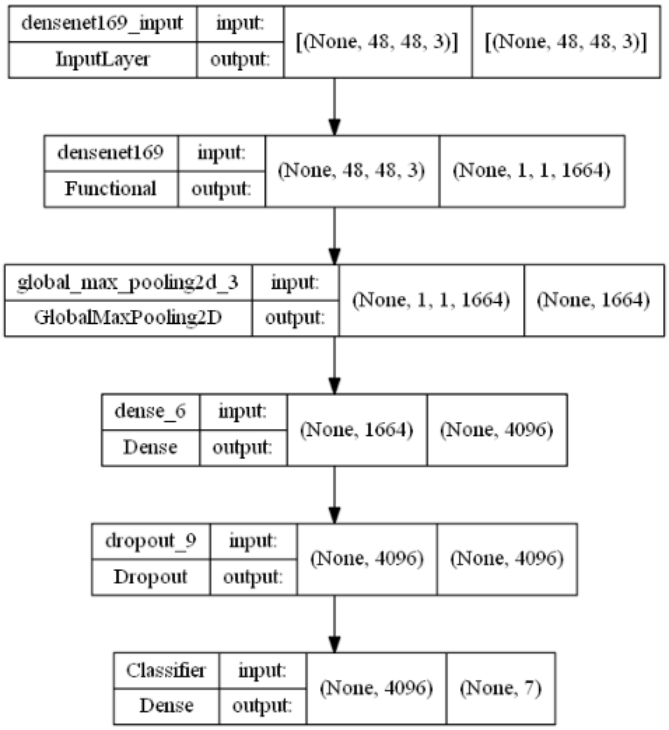


Figure 3.3.33 Summary of DenseNet 169 Architecture

There's a total of 6,848,519 trainable parameters and 12,642,880 non-trainable parameters. The non-trainable parameters which are also known as frozen parameters. By freezing the parameters, the training time can be reduced. The total parameters of DenseNet 169 is 19,491,499 parameters that are defined as a very complex architecture.

Step 2: Train our DenseNet 169 :

```
Epoch 37: val_accuracy did not improve from 0.40550

Epoch 37: ReduceLROnPlateau reducing learning rate to 3.199999628122896e-06.
359/359 [==============================] - 68s 188ms/step - loss: 1.6005 - accuracy: 0.3681 - precision_2: 0.6404 - recall_2:
0.1067 - auc_2: 0.7594 - val_loss: 1.5453 - val_accuracy: 0.4039 - val_precision_2: 0.6779 - val_recall_2: 0.1441 - val_auc_2:
0.7813 - lr: 1.6000e-05
Epoch 37: early stopping
Time taken to train: 0:46:12.373273

Training scores:
============================
Loss:  1.5617
Accuracy:  0.3894
Precision:  0.6956
Recall:  0.1047
AUC:  0.777
============================

Validation scores:
============================
Loss:  1.5455
Accuracy:  0.4055
Precision:  0.6797
Recall:  0.142
AUC:  0.7813
============================

Test scores:
============================
Loss:  1.5532
Accuracy:  0.3934
Precision:  0.6809
Recall:  0.1308
AUC:  0.778
============================
```

Figure 3.3.34 DenseNet169 training output

Similar to other models, DenseNet169 has also prepared to train 40 epochs. But unexpectedly, the DenseNet169 only trained 37 epochs. The time taken for the training of DenseNet169 was 46 minutes as it has 6,848,519 trainable parameters. The accuracy of the test set is below average, which is only 39%. The precision and recall are very imbalanced in all three sets. This will lead to the model ignoring some classes and cannot separate the class well.

Step 3 : Plot and Evaluate Classification Metrics

- **DenseNet169 : Training vs Validation Accuracy and Loss History**

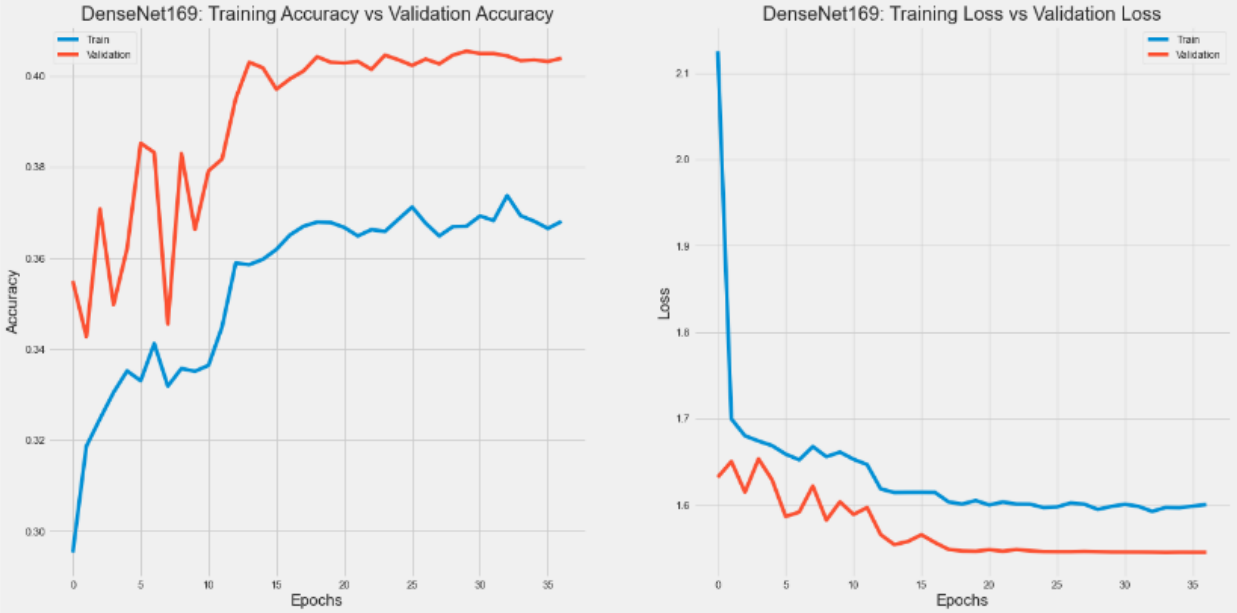


Figure 3.3.35 DenseNet169 Training History

In terms of Accuracy, the model hit a plateau at 20 epoch to 25 epoch on both training and validation set. While in terms of Loss, the model hit a local minimum at around 15 epoch onwards on both validation and train set. These results tell us that more epochs will not upgrade the accuracy and loss of the model. Normally, the training set's accuracy is higher than the validation set, but this model has proven that the validation set's accuracy is higher. It's because the training set comprises augmented photos, which are more difficult to classify, resulting in poorer accuracy and larger loss.

- **DenseNet169 : ROC AUC**

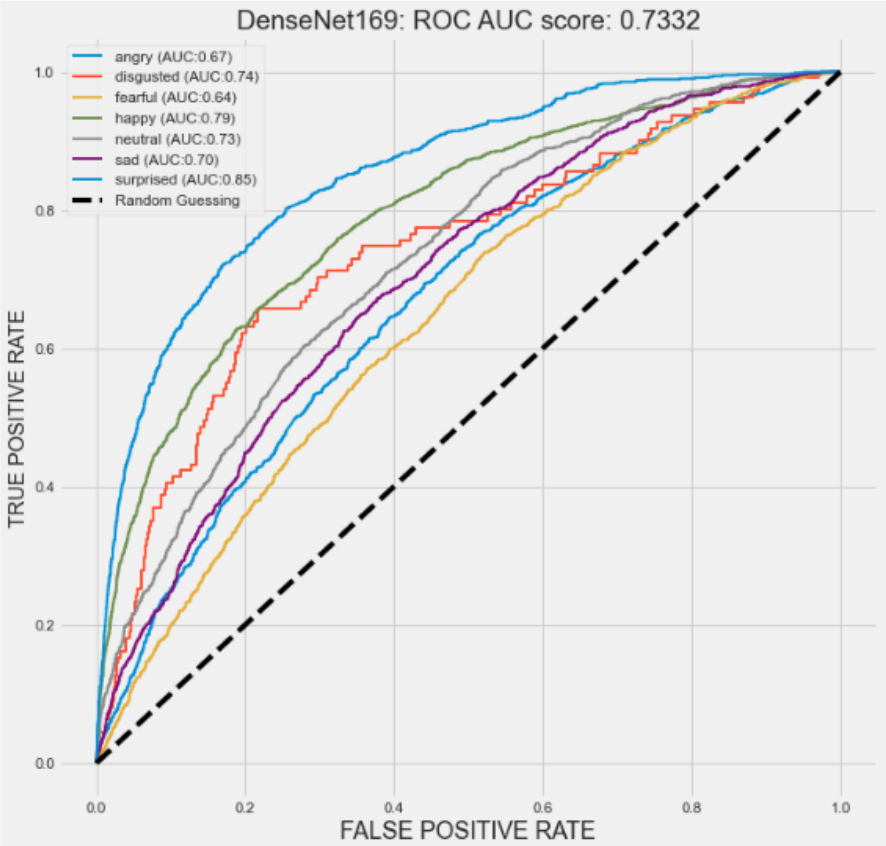


Figure 3.3.36 DenseNet ROC AUC

The False Positive Rate (FPR) is compared to the True Positive Rate (TPR). Using the test set, we plotted the model's ROC AUC. We can see from our ROC AUC plot that DenseNet169 distinguishes the surprised (light blue) class the easiest, while the fearful (yellow) class is the most difficult. The remaining classes have moderate separability, with AUC scores ranging from 0.64 to 0.79. It's reasonable that the model acted in this manner because the surprise pictures are distinct and easy to discer. With an AUC of 0.7332, the model can still generalize and handle the imbalance to the test set reasonably well.

- **DenseNet169 : Error by class on Test set**

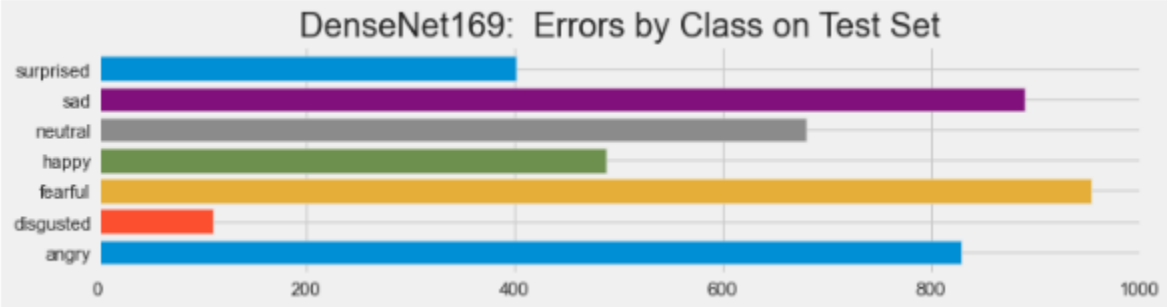


Figure 3.3.37 MobileNet169: Error on test set

Figure 3.3.6.5 illustrates how many images in each class had the model incorrectly labeled. As expcected. The emotions sad,' 'fearful,' and 'angry' contain the most inaccuracies, as expected, because they have the most errors. Because there are fewer disgusted photos in our collection, the Disgusted class has a low error rate.

- **DenseNet169 : Confusion Matrix and Classification Report**

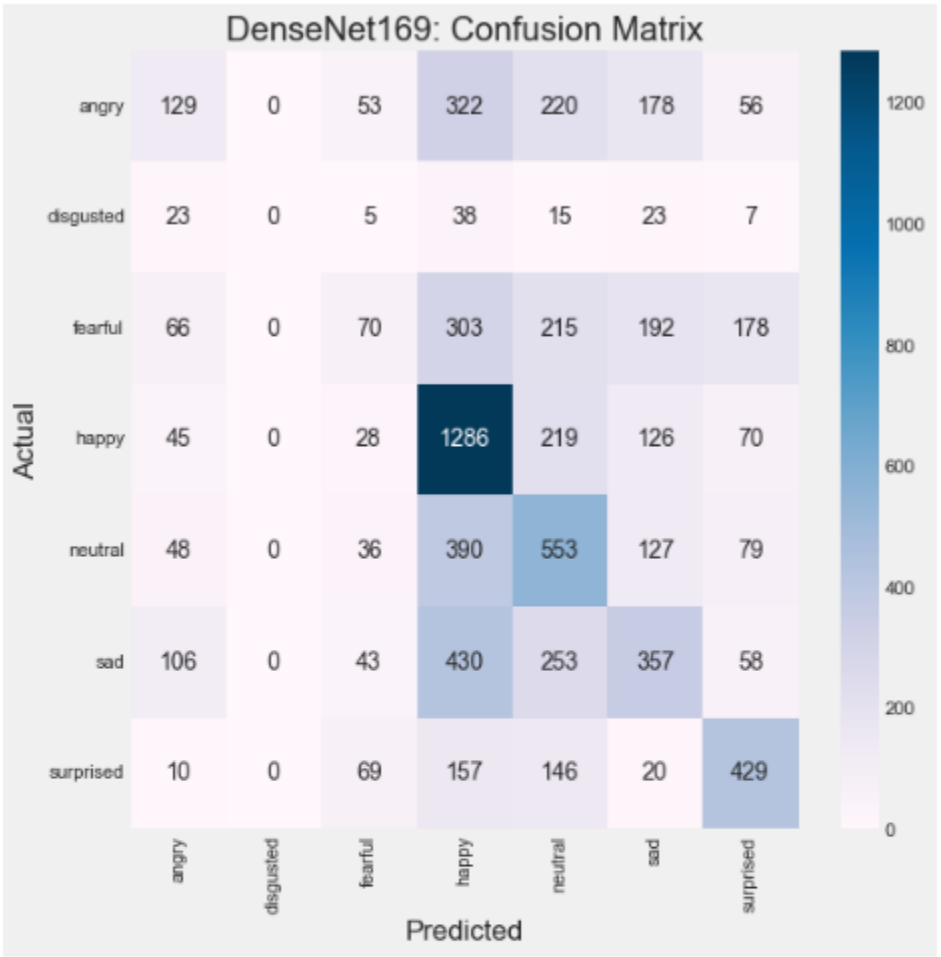


Figure 3.3.38 MobileNet169: Confusion Matrix

DenseNet169: Classification Report on Test Set:

	precision	recall	f1-score	support
angry	0.30	0.13	0.19	958
disgusted	0.00	0.00	0.00	111
fearful	0.23	0.07	0.11	1024
happy	0.44	0.72	0.55	1774
neutral	0.34	0.45	0.39	1233
sad	0.35	0.29	0.31	1247
surprised	0.49	0.52	0.50	831
accuracy			0.39	7178
macro avg	0.31	0.31	0.29	7178
weighted avg	0.36	0.39	0.35	7178

Figure 3.3.39 DenseNet169: Classification Report

The most summary performance of the model on our test set is shown in Figures 3.3.2.6 and 3.3.2.7. It has a 39 percent accuracy rate, which can be increased. The DenseNet169's f1 score for happy and surprised are more than average. The remaining classes did poorly, with less than 0.4 F1 scores, and the disgusted class received a 0 F1 score. This signifies that the model has totally ignored the disgusted class because there aren't enough photos.

- **DenseNet169 : Test model on a sample image**

```
# Test DenseNet on a sample image

classify_test(DenseNet_model, 'surprised', channel)
```

```
Predicted Class:  fearful
Probability:  tf.Tensor(
[[0.16457215 0.01544467 0.25774223 0.16074336 0.08774278 0.08881696
  0.22493786]], shape=(1, 7), dtype=float32)
```

Figure 3.3.40 DenseNet169: Classify test

We utilized an example image from the surprised class to see if DenseNet169 could identify it correctly or not. Surprisingly, it was classified wrong! Based on our ROC AUC, it is unable to classify the surprised class due to its probability only consisting of 0.1645, indicating that the model is not totally confident in its classification. The similarity between the surprised and fearful are close. We couldn't tell if the individual in the image was surprised or fearful by looking at them.

- **Summary & Insight**

We discovered that the surprised class is difficult to detect after training the DenseNet169, as the photos do resemble those of other classes. Aside from that, DenseNet169 underperformed in terms of accuracy, precision, recall, and AUC. Furthermore, DenseNet169 improves at a slow-going pace. The following are some major model metrics on the test set:

Metrics	Score
Accuracy	39%
Averaged F1 score of all classes	$\frac{0.19+0+0.11+0.55+0.39+0.31+0.50}{7} = 0.29$
AUC score	0.7332

Summary table of all the models on the test set:

Metrics:	Custom CNN	Vgg16	ResNet50V2	MobileNetV2	DenseNet169
Loss	1.0043	1.5411	1.5607	1.6099	1.5532
Accuracy	0.6205	0.397	0.4007	0.3649	0.3934
Precision	0.7392	0.641	0.7267	0.6231	0.6809
Recall	0.5035	0.1406	0.1208	0.123	0.1308
AUC	0.9156	0.7832	0.7768	0.7548	0.778

Figure 3.3.41 Summary table of all models

It is clear that our Custom CNN is the best model among all 5 models we experimented with using the same hyperparameters. The worst one is MobileNetV2. Our Custom CNN can perform well because it can handle the class_weights hyperparameters whereas the other models cannot, this makes the custom CNN way more robust than the other models. It achieved the highest accuracy, accuracy, precision, recall, as well as AUC. Therefore, we will only use Custom CNN in the next phase, Hyperparameter Tuning. We will not use other models for fine tuning as they have proven to be ineffective for our problems.

3.4 Hyperparameter Tuning of our best model: Custom CNN

To our surprise, the best model trained using the initial hyperparameter is our own custom built CNN. This is because our custom CNN can perform well even if we pass the class_weights hyperparameter into training. The other models achieved less than 20% when we pass in class_weights. And even without class_weights, they can only achieve around 40% accuracy. In this final phase, we will be tuning the hyper parameters and training the Custom CNN again. Hopefully, it will achieve higher accuracy than 62% and learn the representation of our dataset better.

Step 1: Specifying the hyperparameters to tune

	Hyperparameters	Values
1.	Learning Rate	0.001
2.	Hidden layers	2+1 layers
3.	Neurons in hidden layer	From 128 to 256
4.	Filters in last convolution block	From 256 to 512
5.	Batch Size	32
6.	Epochs	70
7.	Optimizers	Adam, SGD, SGD+Nesterov, Nadam, RMSprop

Table 3.4.1 Hyperparameters to tune table

Smaller learning rate means the model will converge slower but it will not overshoot, therefore guarantee a convergence. Next, we will use more hidden layers and add neurons to those layers to allow our model to learn more complex representation of the data. Third, we will add more filters to the last convolution layer to allow the models to learn more features of the images. Fourth, we will reduce the batch size because this allows the model to update the weights more frequently, although it will make training time slower. We use 70 epochs just in case some optimizers can improve even after 60 epochs. Finally, we will experiment with different optimization algorithms to find out which suits our problem better.

Step 2: Redefine the models' architecture

```
# Block-4

CNN.add(Conv2D(512,(3,3),padding='same',kernel_initializer='he_normal')) # From 256 to 512
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(Conv2D(512,(3,3),padding='same',kernel_initializer='he_normal')) # From 256 to 512
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(MaxPooling2D(pool_size=(2,2)))
CNN.add(Dropout(0.2))

# Classification Part
# Block-1

CNN.add(Flatten())
CNN.add(Dense(256,kernel_initializer='he_normal')) # From 128 to 256
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(Dropout(0.2))

# Block-2

CNN.add(Dense(256,kernel_initializer='he_normal')) # From 128 to 256
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(Dropout(0.2))

# Block-3 --> Added Layer

CNN.add(Dense(256,kernel_initializer='he_normal')) # From 128 to 256
CNN.add(Activation('relu'))
CNN.add(BatchNormalization())
CNN.add(Dropout(0.2))

# Block-4

CNN.add(Dense(7,kernel_initializer='he_normal'))
CNN.add(Activation('softmax')) # Softmax activation since we are doing multiclass classification
```

Figure 3.4.2 Model architecture code

The values that we have circled are the ones that have changed from the previous version. This architecture will be used by all the CNN models later on.

Step 3: Train the models using different optimizers

Optimizer 1: CNN with Adam

```
# Training CNN with Adam

# Save the tuned model to a new folder 'Tuned_models'
CNN_Adam_checkpoint = ModelCheckpoint("./Tuned_models/CNN_Adam.h5", monitor='val_accuracy',verbose=1,save_best_only=True, mode='m
CNN_Adam_callbacks = define_callbacks(CNN_Adam_checkpoint)

CNN_Adam_history = train_model(CNN_Adam, train_set, validation_set, test_set, CNN_Adam_callbacks, class_weights, epochs=70)
```

Figure 3.4.3 Code to train CNN_Adam

We will be training CNN using Adam first to find out if the hyperparameters other than optimizers listed in Figure 3.4.1 will improve the performance of the models. This is because we have used Adam as well in the first modeling phase, which means if the performance improves in using these hyperparameters, It will mean they are better than previous hyperparameters. Only then will we proceed with experimenting using different optimization algorithms.

- **Training :**

```
Epoch 45: val_accuracy did not improve from 0.64170
359/359 [==============================] - 49s 135ms/step - loss: 0.8199 - accuracy: 0.6620 - precision_2: 0.7662 - recall_2:
0.5391 - auc_2: 0.9318 - val_loss: 0.9753 - val_accuracy: 0.6394 - val_precision_2: 0.7293 - val_recall_2: 0.5476 - val_auc_2:
0.9220 - lr: 4.0000e-05
Epoch 45: early stopping
Time taken to train: 0:36:05.674961

Training scores:
============================
Loss:  0.8433
Accuracy:  0.6847
Precision:  0.7864
Recall:  0.5618
AUC:  0.9405
============================

Validation scores:
============================
Loss:  0.9901
Accuracy:  0.6417
Precision:  0.7319
Recall:  0.5354
AUC:  0.9194
============================

Test scores:
============================
Loss:  1.002
Accuracy:  0.6319
Precision:  0.7209
Recall:  0.5336
AUC:  0.9173
============================
```

Figure 3.4.4 Training results of CNN_Adam

Figure 3.4.4 proves that the hyperparameters are better as the accuracy for both validation and test sets have improved by 1%, and training accuracy improved by 3% compared to the first custom CNN we have built. The precision did not improve much, but the recall has improved by 3 percent. More importantly, the AUC score improved from 0.7 to 0.9173. This means our model can separate classes better now (refer to Figure 3.3.2). This model took 36 minutes to finish training because it is quite a complex model with more than 4 millions parameters. Now we have enough evidence to proceed with the experimentation of different optimization algorithms.

Optimizer 2: CNN with SGD

```
Epoch 23: val_accuracy did not improve from 0.21895
359/359 [==============================] - 45s 125ms/step - loss: 1.8744 - accuracy: 0.2123 - precision_3: 0.3790 - recall_3:
0.0173 - auc_3: 0.6135 - val_loss: 1.8791 - val_accuracy: 0.2139 - val_precision_3: 0.3516 - val_recall_3: 0.0279 - val_auc_3:
0.6360 - lr: 4.0000e-05
Epoch 23: early stopping
Time taken to train: 0:16:46.659269

Training scores:
============================
Loss:  1.8925
Accuracy:  0.2002
Precision:  0.3809
Recall:  0.0099
AUC:  0.6165
============================

Validation scores:
============================
Loss:  1.8642
Accuracy:  0.219
Precision:  0.4184
Recall:  0.0246
AUC:  0.6391
============================

Test scores:
============================
Loss:  1.8563
Accuracy:  0.2278
Precision:  0.4772
Recall:  0.0306
AUC:  0.6423
============================
```

Figure 3.4.5 CNN with SGD training output

Figure 3.4.5 shows us that SGD is clearly NOT the optimizer for our problem. The model did not improve after 23 epochs and achieved an accuracy of >20% on both validation and test sets. Precision and recall are very bad as well with 0.48 and 0.03 respectively on the test set. AUC score is also very low at 0.6423. The training time is only 16 minutes because of the inability to improve.

Optimizer 3: CNN with SGD + Nesterov

```
Epoch 57: val_accuracy did not improve from 0.36144
359/359 [==============================] - 49s 135ms/step - loss: 1.6647 - accuracy: 0.3157 - precision_4: 0.5354 - recall_4:
0.0639 - auc_4: 0.7172 - val_loss: 1.6520 - val_accuracy: 0.3614 - val_precision_4: 0.5379 - val_recall_4: 0.1346 - val_auc_4:
0.7487 - lr: 3.2000e-07
Epoch 57: early stopping
Time taken to train: 0:43:40.883280

Training scores:
============================
Loss:  1.7263
Accuracy:  0.3188
Precision:  0.4616
Recall:  0.1031
AUC:  0.7213
============================

Validation scores:
============================
Loss:  1.6516
Accuracy:  0.3614
Precision:  0.5424
Recall:  0.1348
AUC:  0.7485
============================

Test scores:
============================
Loss:  1.6464
Accuracy:  0.3583
Precision:  0.5414
Recall:  0.1431
AUC:  0.7496
============================
```

Figure 3.4.6 CNN with SGD + Nesterov acceleration training output

Figure 3.4.6 shows us that SGD is clearly NOT the optimizer for our problem. The model was able to train until 57 epochs but only achieved an accuracy of 36% on both validation and test sets. Precision and recall are very bad as well with 0.54 and 0.14 respectively on the test set. Although SGD is not a great algorithm for our problem. However the Nesterov acceleration did help increase the accuracy from 22% to 36% accuracy on the test set. AUC score is decent at 0.7496. The time taken to train the model is 43 minutes 40 seconds, which is quite long as it was able to train until 57 epochs.

This means Adam + Nesterov should give us a greater accuracy compared to basic Adam. We will see if our hypothesis is true in the next figure.

Optimizer 4: CNN with Adam + Nesterov (Nadam)

```
Epoch 50: val_accuracy did not improve from 0.65964
359/359 [==============================] - 70s 194ms/step - loss: 0.7568 - accuracy: 0.6897 - precision_6: 0.7815 - recall_6:
0.5830 - auc_6: 0.9424 - val_loss: 0.9661 - val_accuracy: 0.6570 - val_precision_6: 0.7343 - val_recall_6: 0.5645 - val_auc_6:
0.9242 - lr: 1.6000e-06
Epoch 50: early stopping
Time taken to train: 0:51:05.115799

Training scores:
============================
Loss:  0.7424
Accuracy:  0.7253
Precision:  0.8133
Recall:  0.6238
AUC:  0.9539
============================

Validation scores:
============================
Loss:  0.9646
Accuracy:  0.6596
Precision:  0.7362
Recall:  0.5644
AUC:  0.9245
============================

Test scores:
============================
Loss:  0.9702
Accuracy:  0.6485
Precision:  0.7233
Recall:  0.5655
AUC:  0.9237
============================
```

Figure 3.4.7 CNN with Adam + Nesterov acceleration training output

Awesome! Now our validation accuracy is 66% compared to 64% without the Nesterov acceleration. Our test accuracy also increased from 63.19% to 64.85% which is an increase of 1.66%. The Precision on the test set remains as 0.72. However, recall increased to 0.57 from 0.53 compared to the original Adam optimizer. The AUC score is also higher at 0.9237. The time taken to train the model is quite long as it was able to train until 50 epochs.

We have proven our hypothesis true, that Nesterov acceleration will improve the performance of the model.So far, Nadam is the best optimizer for our dataset, however we have one more optimizer to experiment with, which is RMSprop.

Optimizer 5: CNN with RMSprop

```
Epoch 46: val_accuracy did not improve from 0.61679

Epoch 46: ReduceLROnPlateau reducing learning rate to 3.200000264769187e-07.
359/359 [==============================] - 54s 150ms/step - loss: 1.0401 - accuracy: 0.5909 - precision_8: 0.7465 - recall_8:
0.4255 - auc_8: 0.9011 - val_loss: 1.0302 - val_accuracy: 0.6144 - val_precision_8: 0.7496 - val_recall_8: 0.4651 - val_auc_8:
0.9111 - lr: 1.6000e-06
Epoch 46: early stopping
Time taken to train: 0:41:42.595365

Training scores:
============================
Loss:  1.0019
Accuracy:  0.6214
Precision:  0.77
Recall:  0.4584
AUC:  0.9159
============================

Validation scores:
============================
Loss:  1.0314
Accuracy:  0.6168
Precision:  0.7474
Recall:  0.4654
AUC:  0.9108
============================

Test scores:
============================
Loss:  1.0352
Accuracy:  0.6158
Precision:  0.745
Recall:  0.473
AUC:  0.91
============================
```

Figure 3.4.8 CNN with RMSprop training output

The RMSprop algorithm also did quite well with 62% accuracy on the training set and 61.68% and 61.58% on the validation set and test set respectively. RMSprop overfits less compared to Adam and Nadam however it underfits more as the accuracy is 3% lower than Nadam. Precision Recall is decent but also no better than Nadam, the same goes for AUC score. Using RMSprop, the model trained for 46 epochs and 41 minutes, not as long as Nadam.

Step 4 : Choose the best model to be our final model

- **Evaluate the validation accuracy and loss of all the tuned CNN models**

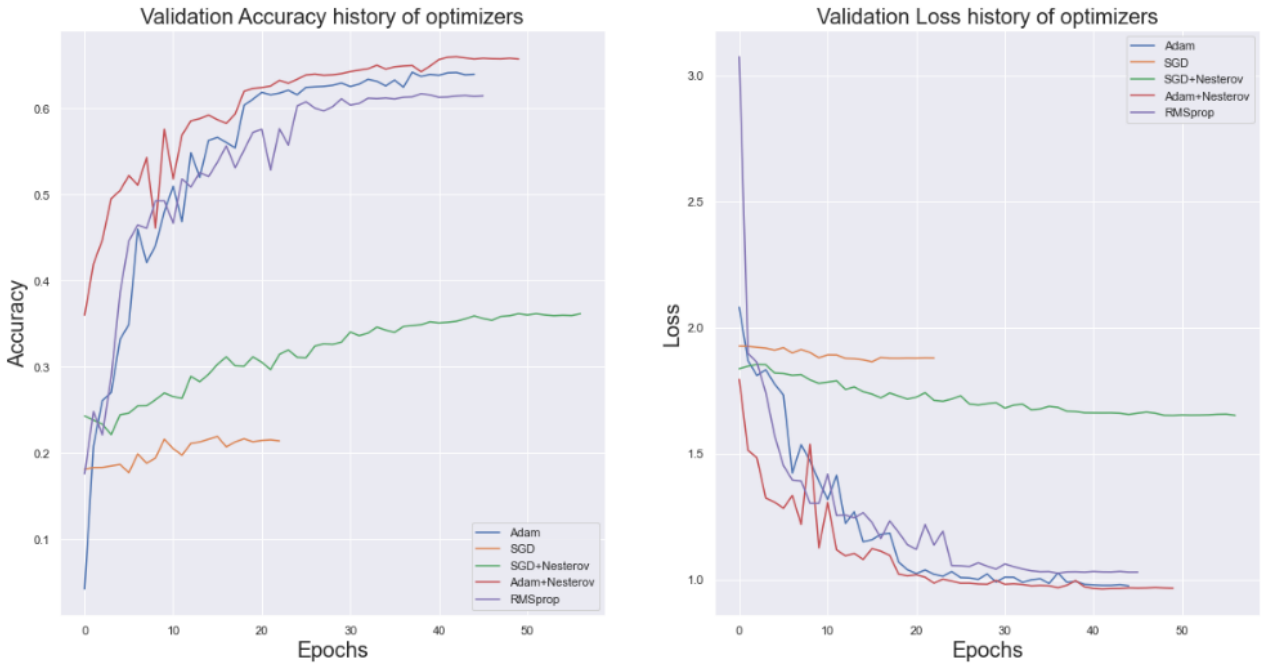


Figure 3.4.9 Validation accuracy and loss history of all the tuned models

In terms of validation accuracy, SGD and SGD+Nesterov performed the worst even though SGD+Nesterov trained for the longest epochs. This is because SGD is a weaker algorithm compared to the other algorithms for our problem. RMSprop ranked third, just a little bit weaker than Adam, it hit a plateau later than Adam at around 25 epochs. Adam ranked second, but 4th in terms of training epochs, because it stopped improving faster than the other 3. Adam+Nesterov or Nadam is the best algorithm among them all, it trained for the second longest epochs only losing to SGD+Nesterov and it achieved the highest accuracy at rounded off, 65% accuracy. Note that the training looks quite volatile compared to before because we have used a smaller batch_size, resulting in more weight updates hence more fluctuations.

- **Overall Performance Summary of all tuned models:**

	Metrics	Adam	SGD	NSGD	Nadam	RMSprop
0	Loss	1.002022	1.646398	1.646398	0.970159	1.035179
1	Accuracy	0.631931	0.358317	0.358317	0.648509	0.615770
2	Precision	0.720873	0.541381	0.541381	0.723272	0.745008
3	Recall	0.533575	0.143076	0.143076	0.565478	0.472973
4	AUC	0.917317	0.749620	0.749620	0.923660	0.910009

Figure 3.4.10 Performance of all tuned models on test sets

Figure 3.4.10 shows the performance of all tuned models on the test set. This table confirms that the best performing model is Nadam. It has the lowest loss among all models at 0.970159, highest accuracy at 65% accuracy. Highest recall at 0.565478 and highest AUC at 0.923660. However, it is only the second highest in terms of precision, with RMSprop standing at first. Overall, it is quite obvious that Nadam prevails and will be selected as our final model.

4.0 Analysis and Result

After multiple experimenting with different CNN models and pretrained models, shortlisting them, and tuning their hyperparameters as well as experimenting with different optimization algorithms, we have reached a conclusion that Nadam is the best and will be used as our final model. Nadam is basically an Adam optimizer but adds a Nesterov Accelerated Gradient. Nadam achieved a good result because Adam is a very robust optimizer that can adapt its learning rate and includes momentum. In addition to that, Nadam incorporates Nesterov Acceleration as well which helps the model "look ahead" to where the parameters will be to calculate the gradient. This wonderful technique can accelerate the initial convergence of gradient descent algorithms like Adam, allowing it to converge faster.

Momentum and Nesterov Accelerated Gradient Formula :

The NAG Update Rule

With momentum:

$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t$$

With NAG:

$$w_{look\_ahead} = w_t - \gamma \cdot update_{t-1}$$
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_{look\_ahead}$$
$$w_{t+1} = w_t - update_t$$

Performances of our final model : CNN with Nesterov Accelerated Adaptive Moment Estimation, Nadam

- **Training vs Validation Accuracy and Loss History**

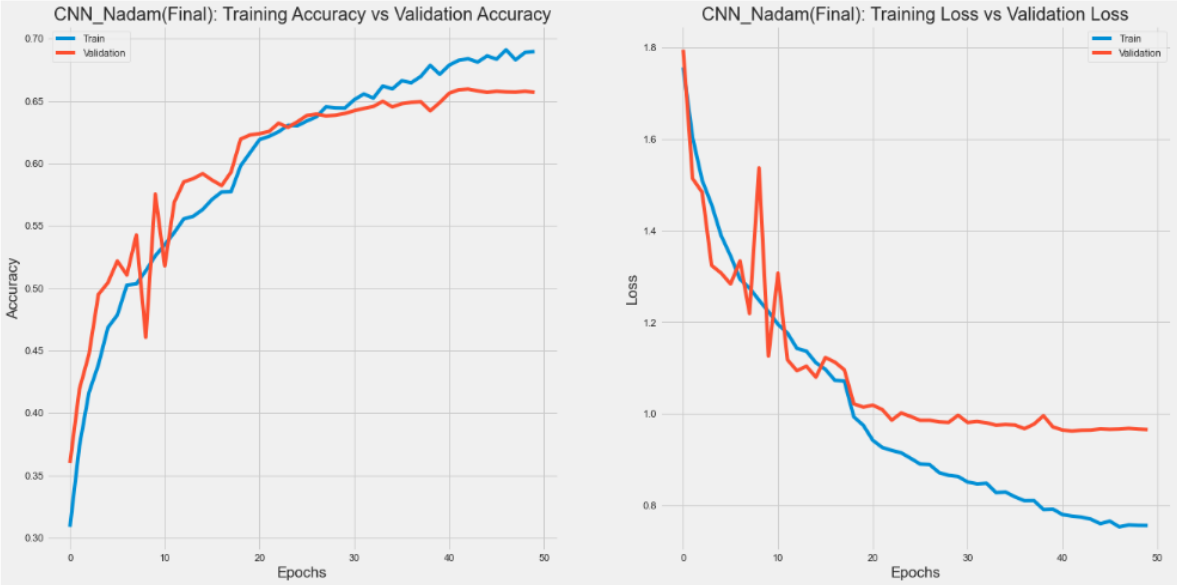


Figure 4.1 Training and validation accuracy, loss of CNN_Nadam

Figure 4.1 shows that our model has achieved quite acceptable accuracy on the train set and validation set as well. The training accuracy did not hit a plateau but the validation accuracy hit a plateau around 43 epochs. This shows our model has slight overfit, but not very serious as the validation accuracy is still above 65%. The loss graph shows the difference between train and validation increasing after around 20 epochs, and the training loss is still quite steep. This means our model is learning the train set very well but the learning on the validation set is slowly decreasing. The reason why our model is slightly overfitting can be due to the added complexity in the hyperparameter tuning phase. We added more hidden layers, neurons, and filters therefore it has high complexity and leads to a very small overfitting. However, it also improved the test accuracy, adding those layers and neurons are still helpful.

- **ROC AUC**

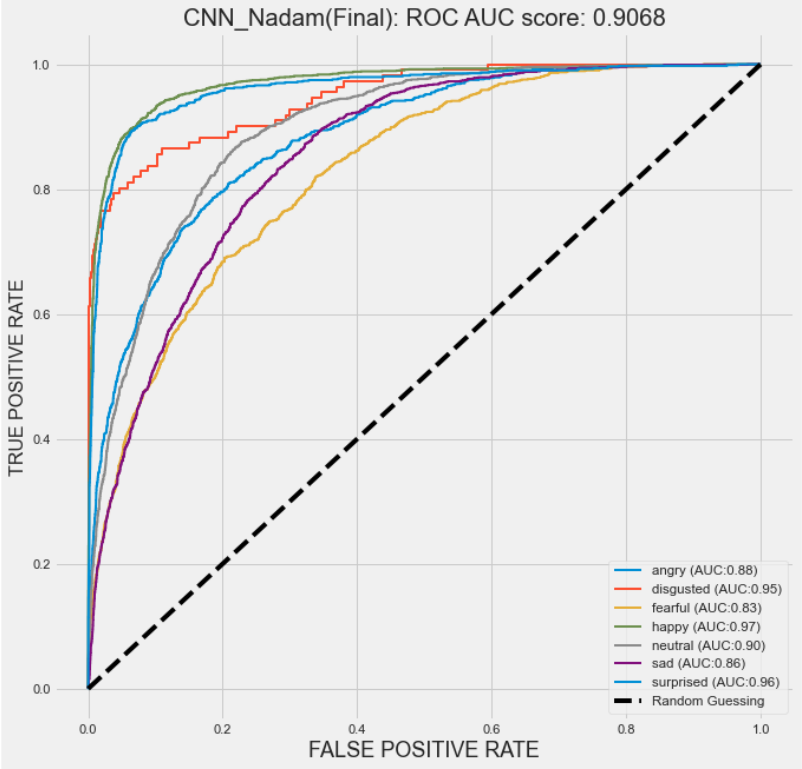


Figure 4.2 ROC AUC of final model

Our final model achieved an AUC score of 0.9068, highest of all models. This means it is able to separate the classes based on their features very well. Looking at the curves, happy and surprised are the easiest, disgusted is also easy for the model, as they have an AUC of 0.97, 0.96, 0.95 respectively. But now, the model is able to separate the other classes much better as well, with the lowest class 'fearful' having a 0.83 score, as compared to the original CNN which only scored 0.81 on 'fearful'. The overall separability of the model has improved slightly compared to the original CNN (Refer to Figure 3.3.1.4).

- **Error by class on Test set**

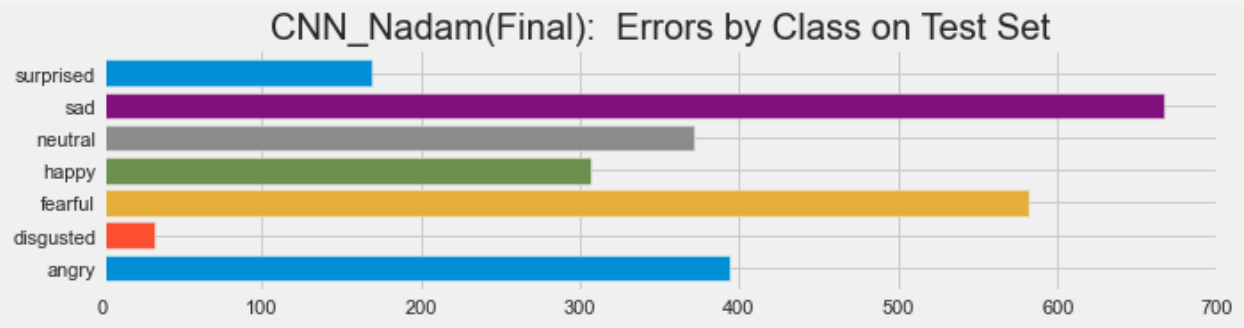


Figure 4.3 Error by class on test set

Figure 3.3.5 shows us how many pictures of each class have the model classified wrongly. As expected, 'sad' and 'fearful' have the most errors as they are the most similar emotions. Disgusted class has low error because of the less number of disgusted images in our dataset. Angry and Neutral is also hard to classify, causing the errors to be high as well. Happy and surprised have some errors because there are many examples of these classes.

- **Confusion Matrix**

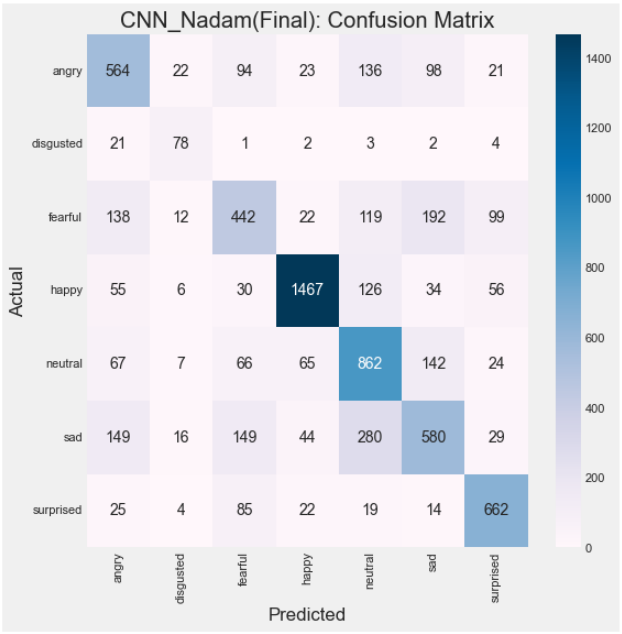


Figure 4.4 Confusion Matrix of the final model

According to the confusion matrix, the true positives for happy and neutral are higher because there are more samples of those classes and disgusted is low because there are less examples, therefore the numbers from the matrix can be misleading as this is a multiclass classification. Precision and recall analysis will be done by visualizing the classification report. However, the confusion matrix can give us some insights as well. For example, we can analyze the False Negatives of each class to find out which classes are the most similar and confused.

We analyze the False Negatives of each class:

Angry: We can see that a lot of False Negatives are neutral, sad, and fearful, this means the model confuses them a lot. This also means they look very similar, which is understandable.

Disgusted: Most FN are angry, therefore some disgusted images look angry to the model.

Fearful: Most FN of the fearful class is sad, which means they are the most similar to each other.

Happy: Most FN of the happy class is neutral, meaning they are also correlated with each other.

Neutral: Neutral and sad are also similar based on the FN of Neutral class.

Sad: Have similar features to neutral, fearful and angry

Surprised: Surprised class is most similar to fearful

- **Classification report**

CNN_Nadam(Final): Classification Report on Test Set:

	precision	recall	f1-score	support
angry	0.55	0.59	0.57	958
disgusted	0.54	0.70	0.61	111
fearful	0.51	0.43	0.47	1024
happy	0.89	0.83	0.86	1774
neutral	0.56	0.70	0.62	1233
sad	0.55	0.47	0.50	1247
surprised	0.74	0.80	0.77	831
accuracy			0.65	7178
macro avg	0.62	0.64	0.63	7178
weighted avg	0.65	0.65	0.65	7178

Figure 4.5 Classification Report

Figure 4.5 shows that the model has less False Negatives on disgusted, happy, neutral, and surprised as their recall is high (Above 0.75 is good) but very low recall on angry, fearful and sad (Less than 0.6). The model has less False positives on happy and surprised but moderate FP for the rest of the classes. This means if the model predicts a face as happy or surprised, they have a high chance to be True happy or surprised.

The best F1 scores to the worst are happy, surprised, neutral, disgusted, angry, sad, and fearful. Accuracy is 65% as mentioned before. This means the model is very good at predicting happy images, but the worst at predicting fearful images.

Testing the final model on all sample images of all 7 classes

1. Angry

```
# Test ResNet on a sample image
classify_test(final, 'angry')
```

```
D:\Anaconda\envs\GPUenabled\lib\site-packages\keras_preprocessing\im
ase use color_mode = "grayscale"
  warnings.warn('grayscale is deprecated. Please use '
```

```
Predicted Class:  angry
Probability:  tf.Tensor(
[[0.6197522  0.12829117 0.15308052 0.00591208 0.02084496 0.05337257
  0.01874644]], shape=(1, 7), dtype=float32)
```

Correct

2. Disgusted

```
# Test ResNet on a sample image
classify_test(final, 'disgusted')
```

```
Predicted Class:  disgusted
Probability:  tf.Tensor(
[[2.0562665e-04 9.9936992e-01 1.6098822e-04 3.0717285e-05 5.7193542e-05
  1.0749449e-04 6.8048030e-05]], shape=(1, 7), dtype=float32)
```

Correct

3. Fearful

```
# Test ResNet on a sample image
classify_test(final, 'fearful')
```

```
Predicted Class:  fearful
Probability:  tf.Tensor(
[[0.35274273 0.02390874 0.41360486 0.00126318 0.01587708 0.07702783
  0.11557551]], shape=(1, 7), dtype=float32)
```

Correct

4. Happy

```
# Test ResNet on a sample image
classify_test(final, 'happy')
```

```
Predicted Class:  happy
Probability:  tf.Tensor(
[[0.00325838 0.00361949 0.00413447 0.95887154 0.02053017 0.00159531
  0.00799072]], shape=(1, 7), dtype=float32)
```

Correct

5. Neutral

```
# Test ResNet on a sample image
classify_test(final, 'neutral')
```

```
Predicted Class:  neutral
Probability:  tf.Tensor(
[[5.9817497e-02 4.9741467e-04 1.6783875e-01 2.1130867e-02 6.2667876e-01
  1.1998515e-01 4.0516211e-03]], shape=(1, 7), dtype=float32)
```

Correct

6. Sad

```
# Test ResNet on a sample image
classify_test(final, 'sad')
```

```
Predicted Class:  angry
Probability:  tf.Tensor(
[[7.2560024e-01 2.2171521e-04 3.1615570e-02 1.5815845e-04 9.3419701e-02
  1.4742489e-01 1.5596667e-03]], shape=(1, 7), dtype=float32)
```

Wrong

7. Surprised

```
# Test ResNet on a sample image
classify_test(final, 'surprised')
```

```
Predicted Class:  surprised
Probability:  tf.Tensor(
[[5.2391324e-04 5.0225110e-05 3.1685498e-01 1.9299901e-03 5.6009134e-04
  1.1691700e-03 6.7891163e-01]], shape=(1, 7), dtype=float32)
```

Correct

Summary & Insights gained :

We have concluded that Nesterov Accelerated Gradient on Adam is the best optimization algorithm and our custom CNN is the best model. The best hyperparameters are:

Hyperparameters	Values
Learning Rate	0.001
Number of hidden layers in the Classification Part of the model	3 layers
Neurons in Hidden layer	256
Number of filters in the last convolution block	512
Batch Size	32
Optimizer	Nadam

We have also discovered something, which is that our dataset has some images that could belong to different classes. interesting which is the ranked similarity of each class. These are the similarities:

Emotions	Most Similar Emotions
Angry	Neutral, Sad, Fearful
Disgusted	Angry
Fearful	Sad
Happy	Neutral
Neutral	Sad
Sad	Neutral, Fearful and Angry
Surprised	Fearful

5.0 Discussion

Humans' as well as Artificial Intelligence's accuracy in recognizing emotions on our dataset is around 65% which hints the possibility that using facial expressions as an avenue to track human's emotions may not be very robust YET (. However, we disagree with that because we have browsed through some of the images during EDA and realized some classes like angry, sad, fearful are very similar. This could be due to mistakes during the data gathering process which we have no control since we took the dataset from Kaggle.

Hypothesis 1 : " Our models will have trouble separating angry images from sad, and disgusted classes as they have similar features." Have been proven to be true. Referring to Figure 4.2 and Figure 4.4, we confidently proved that most False Negatives of the 'angry' class are classified as 'sad' and the most False Negatives of the 'disgusted' class are classified as 'angry'. Therefore there are some correlations between these classes.

Hypothesis 2: "Our models will have trouble classifying disgusted classes as there are too little examples belonging to that class." Have been proven to be wrong. Once we added the parameters class_weights as defined in Figure 3.2.5, we are able to mitigate the effects of imbalance in our dataset. This can be seen when in the classification report of our final model (Figure 4.5). The F1 score for the disgusted class is quite high at 0.61, higher than some of the classes.

There are also some limitations we faced in completing this assignment.
The limitations are:

1. Not enough computing power to run models, as one model training takes nearly an hour. This is because our model is very complex.
2. Dataset is poorly picked (grayscale, imbalance), our dataset has too many overlapping images that can belong to different classes.

Recommendations for Future Works:

1. We can integrate Natural Language Processing into our human emotion recognition system to combine the power of Language and Faces in recognizing emotions.
2. We can find a dataset with more images and with colors / 3 channels, as colors could play a role in separating the emotions.
3. We can use ensemble methods to combine high accuracy, precision, recall models to further improve the performance of our system.
4. We can deploy it in a realtime environment to test the full capability of our model, for example in an interview session to track interviewees emotion changes.

6.0 Conclusion

This study has shown that certain emotions can be harder to classify by Artifical Intelligence systems, which resulted in the model reaching an accuracy plateau at around 65% on the test set. However, we have to take these findings with a grain of salt as this could be a problem which may have occurred during the Data Gathering process. Other than that, transfer learning models fail to perform when we pass in the class_weights parameter into the training. This means they are not robust for our dataset which consists of imbalance classes. Furthermore, Using Adam optimizers paired with Nesterov Accelerated Gradients can improve the accuracy of the model. Adding more hidden layers and neurons is also effective. With the **Nadam optimizer on our custom CNN**, we are able to achieve **72% accuracy on our training dataset and 65% accuracy on our test set**. Which is equivalent to humans' performance on the test set. Therefore we have achieved our objective of this project, which is to achieve a human level of accuracy of 65% or better than humans in emotion recognition on this dataset (Yousif Khaireddin).

As a conclusion, we are quite satisfied with the results of our research and we are very thankful to our magnificent Lecturer and Tutor, DR Sandy Lim Siew Mooi for providing us with all the necessary information and resources to complete this assignment. We would also like to thank all the authors which provided us with very insightful references to enhance our assignment. Without the authors and our tutor's help, we would not have been able to complete this assignment successfully.

Reference

Human Emotion Recognition System

Elon Musk 2020, Joe Rogan experience #1470 - Elon Musk, [video], Available at: <https://www.youtube.com/watch?v=RcYjXbSJBN8&ab_channel=PowerfulJRE> [Accessed 1 February 2022]

SearchEnterpriseAI. 2022. What is convolutional neural network? - Definition from WhatIs.com. [online] Available at: <https://www.techtarget.com/searchenterpriseai/definition/convolutional-neural-network> [Accessed 7 March 2022].

Saha, S., 2018. A Comprehensive Guide to Convolutional Neural Networks—the ELI5 way. [online] Medium. Available at: <https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53> [Accessed 7 March 2022].

Paperswithcode.com. n.d. Papers with Code - Inception-v3 Explained. [online] Available at: <https://paperswithcode.com/method/inception-v3#:~:text=Inception%2Dv3%20is%20a%20convolutional,use%20of%20batch%20normalization%20for> [Accessed 8 March 2022].

Thakur, R., 2019. Step by step VGG16 implementation in Keras for beginners. [online] Medium. Available at: <https://towardsdatascience.com/step-by-step-vgg16-implementation-in-keras-for-beginners-a833c686ae6c> [Accessed 8 March 2022].

Saxena, S., 2021. Lenet-5 | Lenet-5 Architecture | Introduction to Lenet-5. [online] Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2021/03/the-architecture-of-lenet-5/> [Accessed 8 March 2022].

Brownlee, J., 2017. A Gentle Introduction to Transfer Learning for Deep Learning. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/transfer-learning-for-deep-learning/> [Accessed 8 March 2022].

Committed towards better future. 2020. DenseNet Architecture Explained with PyTorch Implementation from TorchVision. [online] Available at: <https://amaarora.github.io/2020/08/02/densenets.html#:~:text=DenseNet%20Architecture%20as%20a%20collection%20of%20DenseBlocks,-We%20already%20know&text=Each%20architec

ture%20consists%20of%20four,%2C%2032%2C%2032%5D%20layers.> [Accessed 8 March 2022].

Resources.wolframcloud.com. 2017. Inception V3 - Wolfram Neural Net Repository. [online] Available at: <https://resources.wolframcloud.com/NeuralNetRepository/resources/Inception-V3-Trained-on-ImageNet-Competition-Data/> [Accessed 21 March 2022].

John Perumanoor, T., 2021. What is VGG16?—Introduction to VGG16. [online] Medium. Available at: <https://medium.com/@mygreatlearning/what-is-vgg16-introduction-to-vgg16-f2d63849f615> [Accessed 21 March 2022].

ImageNet. 2021. ImageNet. [online] Available at: <http://www.image-net.org> [Accessed 22 March 2022].

O'Reilly Online Learning. n.d. Deep Learning for Computer Vision. [online] Available at: <https://www.oreilly.com/library/view/deep-learning-for/9781788295628/fe9712f6-d5b9-4d65-b25c-eb61835ccc6a.xhtml> [Accessed 22 March 2022].

Arxiv.org. 2015. Deep Residual Learning for Image Recognition. [online] Available at: <https://arxiv.org/pdf/1512.03385.pdf> [Accessed 22 March 2022].

Yousif Khaireddin and Zhuofa Chen n.d., Facial Emotion Recognition: State of the Art Performance on FER 2013. [Research Paper] Available from: Google Scholar [Accessed 29 March 2022]

Assessment Rubric

Name(s): Tay Xue Hao, Khoo Chyi Ze, Lee Ming Yi Programme: RDS Y2S3 Group: 2 Date: 29/3/2022

Project Part A – Shortlist promising models (40%) – CLO3

No	Item	Criteria			Final Marks
		Poor	Accomplished	Good	
1	Problem statement (10)	No or very little discussion on existing problem and the project The proposed project already exists, or with very minor change. No discussion or very little of introduction given to the related system or technology. 0-4	Little discussion on existing problem and introduction of proposed project. Minor ideas are modified from existing system(s). Introduction to the related system is given, but no evaluation provided. 5-7	Good discussion and evaluation of existing problem and the proposed project. Ideas modified from existing system, with some creative ideas are added. Good discussion and evaluation of the related system. 8-10	
2	Programming (20)	The end product fails with many logic errors, many actions lacked exception handling. Solutions are over-simplified. Programming skill needs improvement. Evaluation steps of different models are not automated. 0-7	Major parts are logical, but some steps to complete a specific job may be tedious or unnecessarily complicated. Program algorithm demonstrates acceptable level of complexity. The student is qualified to be a programmer Some evaluation steps are automated. 8-15	Correct and logical flow, exceptions are handled well. Demonstrates appropriate or high level of complex algorithms and programming skills. Almost all evaluation steps are automated. 16-20	
3	Degree of completion (10)	Too much still remain to be done. Basic requirements are not fulfilled. The end product produces enormous errors, faults or incorrect results. Limited performance metrics are used. 0-4	All required features present in the interface within the required scope, but some are simplified. Or one or two features are missing. The system is able to run with minor errors. More than 5 performance metrics are shown. 5-7	All required features present in the interface within or beyond the required scope. No bugs apparent during demonstration. More than 8 performance metrics are shown. 8-10	
				Sum of Score	

Project Part B – Fine-tune the system (40%) – CLO3

No	Item	Criteria			Final Marks
		Poor	**Accomplished**	**Good**	
1	Model Optimization (12)	The model is not optimized. Default setting is used without any adjustments. 0-4	The model is optimized based on performance metrics. Different parameters are regularized to optimize the model. Ensemble classifier is not attempted. 5-8	The model is optimized based on performance metrics. Different parameters are regularized to optimize the model Ensemble classifier is evaluated. 9-12	
2	system implementation (13)	The end product is produced with different system design or approach, which is not related to the initial proposal. 0-4	The end product conforms to most of the system design, but some are different from the specification. 5-9	The end product fully conforms to the proposed system design. 10-13	
3	Results (Performance measurement) (10)	Analytical methods were missing or inappropriately aligned with data and research design. Results were confusing. 0-4	The analytical methods were identified. Results were presented. All were related to the research question and design. Sufficient metric or measurement is applied. 5-7	Analytical methods and results presentation were sufficient, specific, clear, structured and appropriate based on the research questions and research design. Extra metric or measurement is applied. 8-10	
4	Organization (5)	The structure of the paper was weak. Transition was weak and difficult to understand. 0-2	A workable structure was presented for presenting ideas. Transition was smooth and clear. 3-4	Structure was intuitive and sufficiently inclusive of important information of the research. Transition from one to another was smooth and organized. 5	
				Sum of Score	

Presentation (20%) – CLO2

No	Item	Criteria			Final Marks
		Poor	**Accomplished**	**Good**	
1	Output (10)	Inadequate information/outputs needed are generated. Most of the information/outputs generated are less accurate. Results visualization is overly cluttered or the design seems inappropriate for the problem area. Lack of information that is useful for the user 0-4	Adequate information/outputs needed are generated. The information/output generated are accurate, but some with errors. Pleasant looking, clean, well-organized results visualization The information displayed is helpful for the user, but some details are omitted. 5-7	All the necessary information/outputs are generated. All or most of the information/outputs generated are accurate. Minor errors can be ignored. The results are visually pleasing and appealing. Great use of colors, fonts, graphics and layout. The information displayed is helpful to the users and complete with necessary details. 8-10	
2	Presentation (10)	The presentation was unclear. Results were presented without justifications and reasons. Results were not supported with ML concepts and theories. 0-4	The presentation is well organized for the most part, but more clarity with transitions is needed. Answers to the research question and system performances supported ML concepts and theories. 5-7	The presentation was concise and straight to the point. The results were presented and illustrated in easily interpretable graphs or charts. The research question and system performance were answered and identified. 8-10	
				Sum of Score	