CT100-3-M-DL
ASSIGNMENT 2

NAME: AHMAD FUAD BIN KHALIT
STUDENT ID:TP058497
LECTURER: PROF. DR. MANDAVA RAJESWARI
WEIGHTAGE: 60%

INSTRUCTIONS TO CANDIDATES:

1 **Submit your assignment at the administrative counter**

2 **Students are advised to underpin their answers with the use of references (cited using the Harvard Name System of Referencing)**

3 **Late submission will be awarded zero (0) unless Extenuating Circumstances (EC) are upheld**

4 **Cases of plagiarism will be penalized**

5 **The assignment should be bound in an appropriate style (comb bound or stapled).**

6 **Where the assignment should be submitted in both hardcopy and softcopy, the softcopy of the written assignment and source code (where appropriate) should be on a CD in an envelope / CD cover and attached to the hardcopy.**

Table of Contents

Abstract

The objective of this experiment is to find model that suitable for our e-KYC system in term of solving the problem in current system such as predicting human expression in most accurate way without sacrificing latency and inference of the model. The structure of the experiment will consist of experimenting with 3 well known model Convolution Network (ConvNet), VGG16 and Resnet50. Our baseline model which is a ConvNet model which will be use for our baseline result prediction to other our pre-trained model. Within our experiment, we will fully utilized Google Colab features such as Tensorboard, tf.data input pipeline, hyperparameter with Hparams dashboard. The challenges we faces in our experiment is data imbalance from our FER2013 dataset. In order to overcome the problems, we experiment with 2 FER2013 dataset that has different size and proposed a simple fully connected layer for transfer learning architecture that utilized regularization technique and able to archived 72.9% accuracy on test dataset.

Chapter 6: Model Implementation

6.1 The Development Environment

There are many way to implement our deep learning environment such as personal workstation, server environment or cloud environment. Its all depend on the hardware requirement and the size of dataset that we used to run the experiment with. For model implementation, we will use Google Colaboratory or Colab as our main environment. Since our experiment working on small & huge dataset when experiment with pre-trained model such as VGG and Resnet, the hardware require to train these model require more advance and expensive GPU to run the training efficiently and Colab offer the service to be used for free.

The model was built based on (https://github.com/ptbailey/Emotion-Recognition/blob/master/emotions.ipynb) jupyter notebook code to speed up and avoid building the model from scratch. However, we have added additional feature such as input pipeline, hyperparameter tuning, validation of the model using random image and utilization of Tensorboard feature. We also add additional pre-trained experiment on top of the original experiment using Resnet50. Apart of major change of the code, minor tweaking also has been done such as using different size of dataset, visualization of dataset using matplotlib, model summary visualization and introduce various kind of configuration such as regularization (batch normalization, dropout), faster optimizer (Momentum, Nesterov Accelerated Gradient) and Performance scheduling (learning rate scheduler, decay rate).

6.2 The Dataset

The dataset we used are from Kaggle that provided dataset for facial emotion training (https://www.kaggle.com/c/facial-keypoints-detector/data) for dataset 1 and (https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data) for dataset 2. Both dataset contain same column and width value but in different depth. The dataset 1 contain 4178 image and dataset 2 contain 28,709 that label each facial expression. The size of the image is 48 height x 48 width with greyscale value and it was save in CSV format. Each of the image are labelled based on 7 basic human expression (Ekman, 1980) as followed:

Label	Facial Emotion
0	Anger
1	Disgust
2	Fear
3	Happy
4	Sad
5	Surprise
6	Neutral

Table 6.1 FER2013 Dataset Label

The example of facial expression that contain in the dataset as followed :-

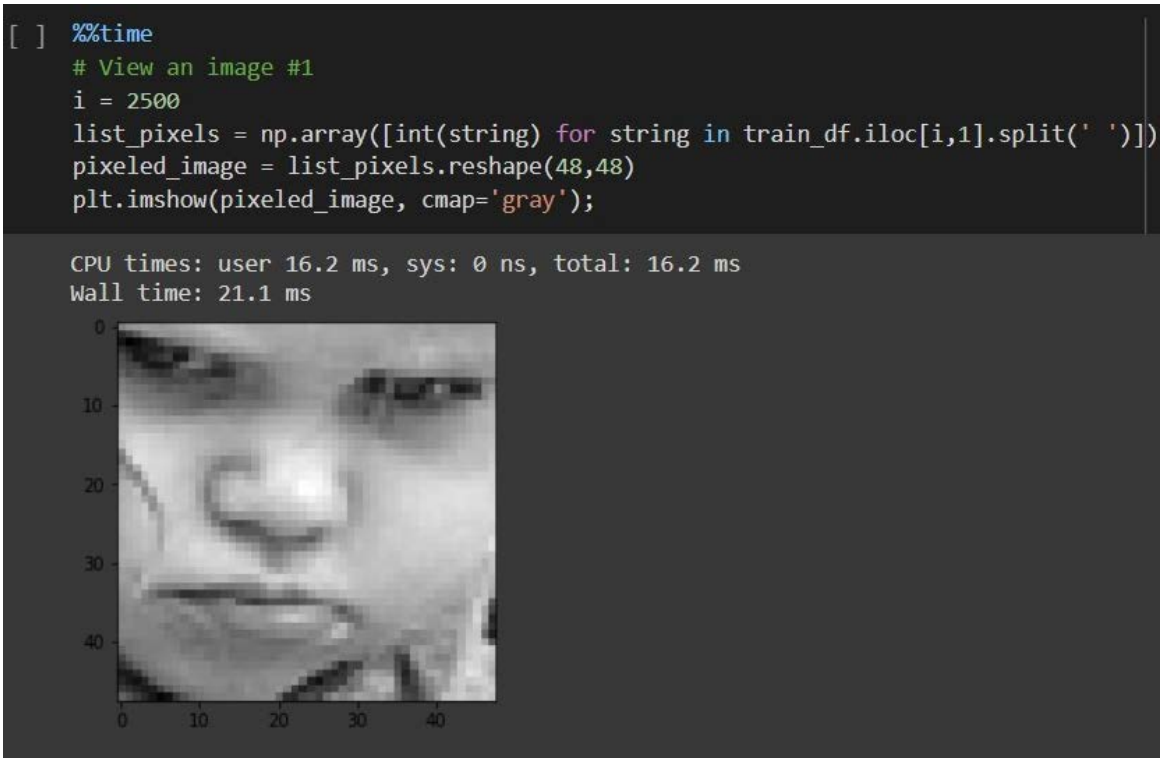


Figure 6.1 Example of Image in Dataset

```
%time
# View an image #2
i = 1500
list_pixels = np.array([int(string) for string in train_df.iloc[i,1].split(' ')])
pixeled_image = list_pixels.reshape(48,48)
plt.imshow(pixeled_image, cmap='gray');
```

```
CPU times: user 3 µs, sys: 0 ns, total: 3 µs
Wall time: 5.48 µs
```

Figure 6.2 Example of Image in Dataset

\

6.2.1 Data Imbalance

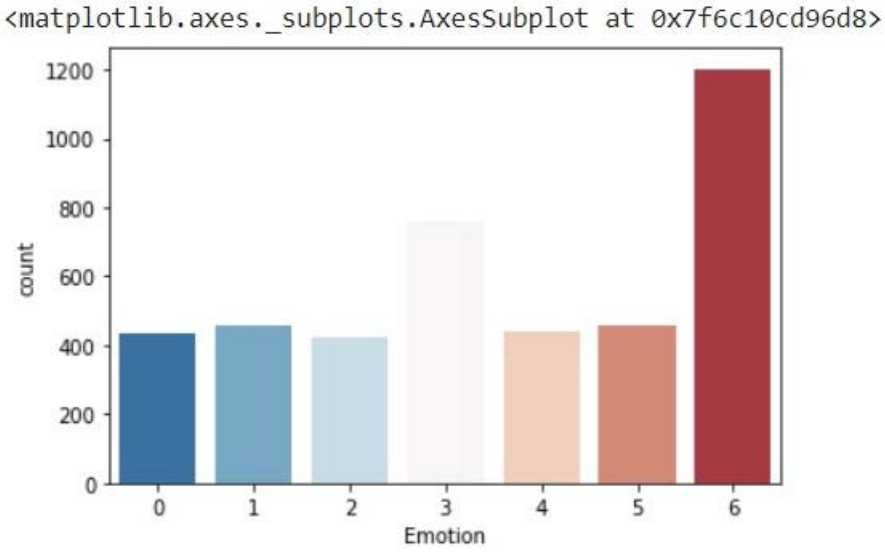


Figure 6.21 Checking Data Imbalance (Dataset 1)

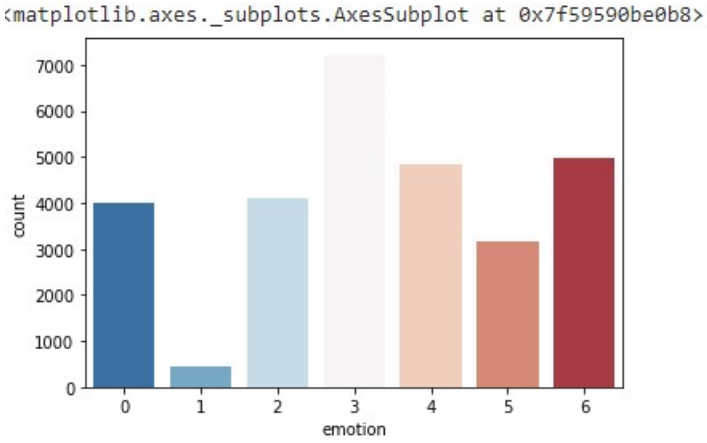


Figure 6.22 Checking Data imbalance (Dataset 2)

Dataset 1 contain very balance between its labels accept for emotion 3 which is happy and emotion 6 which is neutral. This is acceptable due to in psychology most of the time when human take portrait picture they will show common 2 expression which is smiling face(happy) or resting facial expression (neutral). While dataset 2 has more variance in the data and not very balance compare to dataset 1.

6.3 Data Preprocessing

In our experiment, we will use multidimensional data structures to train the image. Common library for data manipulation such as NumPy that efficient for array calculation and Pandas for tabular data analyzing will be utilizing during whole experiment.

Depending on the model that will be used in the experiment, we will convert our array to from 1 dimension array up to 3 dimension array that is required as input in transfer learning pre trained architecture. The process of converting image into array are as below:

1. After dataset is loaded into program and stored using pandas dataframe, convert it to 2 dimension array using Numpy as per **Figure 6.31** & **Figure 6.32**

```
%time
# Read in the files from Google Drive

train_df = pd.read_csv('/content/drive/MyDrive/Fuad_Assignment/train.csv')
#train_df = pd.read_csv('/content/drive/MyDrive/Deep Learning/Assignment2/

train_df
```

```
CPU times: user 2 µs, sys: 0 ns, total: 2 µs
Wall time: 4.53 µs
```

	Emotion	Pixels
0	3	221 240 251 254 255 255 255 255 255 255 255 25...
1	6	100 107 108 104 103 113 117 115 120 130 138 14...
2	4	35 50 56 57 63 76 74 79 85 86 105 133 145 152 ...
3	6	119 124 129 135 136 140 142 149 159 156 163 16...
4	2	160 173 186 194 188 185 175 162 153 143 135 12...

Figure 6.31 Load csv into dataframe

```
[ ] #%%timeit
    %%memit
    # Reshape 1D array to 2D matrix for Conv2D

    x_train = []
    for i in range(len(train_df)):
      # convert string of pixels to list of integers
      list_pixels = [int(string) for string in train_df.iloc[i,1].split(' ')]
      # reshape 1D to 2D matrix
      pixeled_image = np.array(list_pixels).reshape(48,48)
      x_train.append(pixeled_image)

    x = np.array(x_train)
    y = train_df.iloc[:,0]
```

Figure 6.32 Convert dataframe into 1D array to 2D array

2. Label (y) will be transformed into categorical using keras function to_categorical()

```
[ ]  %time
     # Categorize y for compatiblity with softmax activation
     from keras.utils import to_categorical
     y = to_categorical(np.array(y))
```

Figure 6.33 Data Transformation of label

3. Then, we partition the data into training and testing with ratio of 90:10. Then from training dataset we further split it to training and validation with ratio 80:20. Final composition of the data will be 72% training, 18% Validation and 10% Testing (Holdout) from 4178 image.

```
[ ]  # Train_test_split training data into training and test sets
     x_train1, x_test, y_train1, y_test =  train_test_split(x, y, random_state = 123, test_size = 0.10)

[ ]  # Train_test_split training data into training and validation sets
     x_train, x_val, y_train, y_val = train_test_split(x_train1, y_train1, random_state = 123, test_size = 0.20)

     x_train, x_val, y_train, y_val = train_test_split(x_train1, y_train1, random_state = 123, test_size = 0.20)
```

Figure 6.34 Data Partition of the dataset

4. And final step then is to expend the dimension using expend_dims () function by specify the axis =3. This will reshape the array into input 4d tensor.

```
[ ]  # Reshape the data to be  (2924, 48, 48, 1) and (1254, 48, 48, 1) for x_train and x_val
     # respectively. need to do this because input_shape in model_cnn.add needs a 2D image with
     # depth specified as 1
     x_train_reshaped = np.expand_dims(x_train, axis = 3)
     x_val_reshaped = np.expand_dims(x_val, axis = 3)
     x_test_reshaped = np.expand_dims(x_test, axis = 3)
```

Figure 6.35 Reshape Numpy Array to Input 4D Tensor

5. Additional Step: When training with pre trained model, the pre trained model accept different channel which is RGB channel so we need to change our Numpy array from 48x48x1 to 48x48x3 using Numpy repeat() function.

```
[ ]  # Need to repeat our 48x48x1 from 1 channel to 3 channel, because transfer learning models take RGB format
     #For Resnet Model
     x_train_rgb50 = np.repeat(x_train_reshaped5, 3, axis=3)
     x_val_rgb50 = np.repeat(x_val_reshaped5, 3, axis=3)
     x_test_rgb50 = np.repeat(x_test_reshaped5, 3, axis=3)
```

Figure 6.36 Changing Greyscale into RGB input tensor

6.4 Baseline Model: Convolution Neural Network (ConvNet)

Convolution Neural Network (ConvNet) model is simplest type of model that consist of convolution layer and pooling layer that stack up on top of each other. The simplicity of the model is the reason we choose it as our baseline model and compare with other pre trained model. The step of setting up our ConvNet model are as below: -

```
[ ]  # Setting Hyperparameter (For Manual Search of Hyperparameter)
     unit1 = 64
     unit2 = 7
     drop_rate = 0.2

     # Define model as Sequential class

     model_cnn = models.Sequential()
     model_cnn.add(Conv2D(32, kernel_size=(5, 5), strides=(2, 2),
                          activation='tanh',
                          input_shape=(48,48,1))) # kernel size = filter window size. 32 = number of output channels
     model_cnn.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
     model_cnn.add(Conv2D(64, (5, 5), activation='tanh'))
     model_cnn.add(MaxPooling2D(pool_size=(2, 2)))
     model_cnn.add(layers.Flatten())
     model_cnn.add(layers.Dense(unit1, activation='tanh'))
     model_cnn.add(layers.Dropout(drop_rate))
     model_cnn.add(layers.Dense(unit2, activation='softmax')) # 7 classifications
     model_cnn.summary()
```

Figure 6.41 Model Construction using Keras Sequential Model

In the model construction, we declare unit1, unit2 and drop rate to be use for our hyperparameter tuning using manual search. Then as **Figure 6.41** show, we start our convolution 2D(Conv2D) layer with this layer accepted 4D input tensor and output 4D tensor. Before going to next convolution layer, we stack a MaxPooling2D in between. MaxPooling2D will down sample the 4D tensor input from first layer of convolution by taking maximum value of the define pool size. This resulted the shape of tensor reduce from 22 to 11. Then will go to second conv2D layer & MaxPolling2D until the output shape became small enough to 3x3 then from convolution layer we feed the output to fully connected layer which contain flatten layer and hidden layer. Inside the hidden layer between 2 dense layer we applied dropout layer that help to prevent overfitting.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 22, 22, 32)        832
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 11, 11, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 7, 7, 64)          51264
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 3, 3, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 576)               0
_________________________________________________________________
dense (Dense)                (None, 64)                36928
_________________________________________________________________
dropout (Dropout)            (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 7)                 455
=================================================================
Total params: 89,479
Trainable params: 89,479
Non-trainable params: 0
_________________________________________________________________
```

Figure 6.42 Model Summary

Once we build our ConvNet model, we will compile the model with specifying the training configuration by using categorical cross entropy as our loss function, use SGD as our optimizer and monitor the accuracy from the training and validation. The justification of the categorical cross entropy instead of binary cross entropy is that our model has 7 exclusive class and considered as multi class classification. The use of optimizer option such as SGD, Adam or RMSprop will be decided based manual search hyper parameter.

```
[ ]  model_cnn.compile(loss='categorical_crossentropy',
                       optimizer="sgd",
                       metrics=['acc'])

     logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
     tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir="./logdir/CNN1/", histogram_freq=1, profile_batch='15,20')

     history_cnn = model_cnn.fit(x_train_reshaped,
                                 y_train,
                                 epochs = 100,
                                 batch_size=100,
                                 validation_data=(x_val_reshaped, y_val),
                                 callbacks=[tensorboard_callback]
                                )

     print("Average test loss: ", np.average(history_cnn.history['loss'])
```

Figure 6.43 Compile and fitting the model.

6.5 Transfer Learning Model 1: VGG 16

For our experiment with transfer learning, we will use Keras pre trained model of VGG16. As we can see in **Figure 6.51** we set our vgg configuration as followed. First the top layer of the pre train model is cut, then we use the default weight from the model that was train on imagenet dataset and the last part we set the input shape as 48x48x3.

```
vgg_base = VGG16(include_top= False, weights='imagenet', input_shape = (48,48,3))
```

Figure 6.51 Load the VGG pre trained model

After that, we set our experiment setup for fine tune the VGG architecture. In this setup we use trainable () function to freeze and unfreeze the architecture layer. Freeze mean that we don't modify the weights while unfreeze mean we let backpropagation to tweak the original weight of the model. Then we set variable 'fine_tune_at' that will be manually define which layer we want to unfreeze and train the weight and at the end of the code we do the iteration on the pre trained layer based on the variable 'fine_tune_at' that was set before the iteration.

```
#set freeze & unfreeze layer
#vgg_base.trainable = False #freeze all so gradient descend wont modify them
vgg_base.trainable = True #unfreeze specified layer to let backprop tweak them to increase performance (Rule: >train data >layer can unfreeze

print("Number of layers in the base model: ", len(vgg_base.layers))

# Fine-tune from this layer onwards (make sure to unfreeze layer first before finetune)
fine_tune_at = 3 # backprop will tweak layer 16 to 19

# Freeze all the layers before the `fine_tune_at` layer
for layer in vgg_base.layers[:fine_tune_at]:
  layer.trainable =  False
vgg_base.summary()
```

Figure 6.52 Fine tune Pre-Trained Model

Since the top of the architecture was cut during the model was load to the program, now we need to reconstruct new top layer for our VGG. Since VGG is a simple architecture compare to Resnet, we will only use sequential model and build a neural network with 3 dense layers similar to what that has been implement on our baseline ConvNet layer except that we substitute the convolution layer with VGG layer and the fully connected layer remain the same. Each parameters in each layer have been hardcoded and model summary is shown in **Figure 6.53**.

```
[ ]  # Define model as Sequential class
     model_cnn = models.Sequential()
     model_cnn.add(vgg_base)
     model_cnn.add(layers.Flatten())
     model_cnn.add(layers.Dense(128, activation='tanh'))
     model_cnn.add(layers.Dense(64, activation='tanh'))
     model_cnn.add(layers.Dense(7, activation='softmax')) # 7 classifications
     model_cnn.summary()

     Model: "sequential"
     _________________________________________________________________
     Layer (type)                 Output Shape              Param #
     =================================================================
     vgg16 (Functional)           (None, 1, 1, 512)         14714688
     _________________________________________________________________
     flatten (Flatten)            (None, 512)               0
     _________________________________________________________________
     dense (Dense)                (None, 128)               65664
     _________________________________________________________________
     dense_1 (Dense)              (None, 64)                8256
     _________________________________________________________________
     dense_2 (Dense)              (None, 7)                 455
     =================================================================
     Total params: 14,789,063
     Trainable params: 14,750,343
     Non-trainable params: 38,720
     _________________________________________________________________
```

Figure 6.53 VGG top layer configuration

For our VGG training configuration, we will use Stochastic Gradient Descent(SGD) optimizer. Other configuration such as loss function and metrics will be the same as our baseline which is categorical cross entropy loss function and monitor the training and validation accuracy as shown as **Figure 6.54**

```
[ ]  model_cnn.compile(loss='categorical_crossentropy',
                       optimizer="sgd",
                       metrics=['acc'])

[ ]  logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
     tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir="./logdir/vgg/", histogram_freq=1, profile_batch='15,20')

[ ]  history_cnn = model_cnn.fit(x_train_rgb,
                         y_train,
                         epochs=15,
                         batch_size=10,
                         validation_data=(x_val_rgb, y_val),
                         callbacks = [tensorboard_callback])
```

Figure 6.54 Training Configuration & Fitting the Model of VGG16

6.6 Transfer Learning Model 2 : Resnet50

Final pre trained model that we will used in this experiment is the Resnet50 from the Keras API. Not many previous researchers worked on this architecture on FER2013 and we intended to explore the used of Resnet architecture with FER2013 dataset. The workflow of training our Resnet50 are as followed: -

6.6.1 Load the Pre-Trained Model from Keras

```
resnet_base = ResNet50(include_top= False, weights='imagenet', input_shape = (48,48,3))
```

Figure 6.61 Load the Pre trained Resnet to our program

6.6.2 Configure the Fine tune of the Resnet

```
[ ]  resnet_base.trainable = True

     #Fine-tune from this layer onwards (make sure to unfreeze layer first before finetune)
     fine_tune_at = 20 # backprop will tweak layer 20 to 175

     # Freeze all the layers before the `fine_tune_at` layer
     for layer in resnet_base.layers[:fine_tune_at]:
        layer.trainable =  False
```

Figure 6.62 Freezing and unfreeze layer

Resnet architecture is complex in nature. This is due to the used of residual layer in the architecture. In traditional network such as VGG or ConvNet, each layer feeds the output to the next layer but in Resnet residual layer the output of each layer will feed not only the next layer but the layer 2 or 3 hops away. This called skip connection. The Resnet we use in keras is 3x3 conv2d

6.6.3 Constructing Top Layer/ Fully Connected Layer

```
data_augmentation = keras.Sequential(
    [
        layers.experimental.preprocessing.RandomFlip("horizontal"),
        layers.experimental.preprocessing.RandomRotation(0.1),
    ]
)

x = resnet_base.output
x = keras.layers.GlobalAveragePooling2D()(x)
x = keras.layers.Dense(128,activation='tanh')(x)
x = keras.layers.Dropout(0.2)(x)  # Regularize with dropout
x = keras.layers.Dense(64, activation='tanh')(x)
# A Dense classifier with a single unit (binary classification)
outputs = keras.layers.Dense(7, activation='softmax')(x)
model = keras.Model(resnet_base.input, outputs)
model.summary()
```

Figure 6.63 Using functional model to construct Resnet fully connected layer

```
conv5_block3_2_bn (BatchNormali (None, 2, 2, 512)    2048        conv5_block3_2_conv[0][0]
__________________________________________________________________________________________
conv5_block3_2_relu (Activation (None, 2, 2, 512)    0           conv5_block3_2_bn[0][0]
__________________________________________________________________________________________
conv5_block3_3_conv (Conv2D)    (None, 2, 2, 2048)   1050624     conv5_block3_2_relu[0][0]
__________________________________________________________________________________________
conv5_block3_3_bn (BatchNormali (None, 2, 2, 2048)   8192        conv5_block3_3_conv[0][0]
__________________________________________________________________________________________
conv5_block3_add (Add)          (None, 2, 2, 2048)   0           conv5_block2_out[0][0]
                                                                 conv5_block3_3_bn[0][0]
__________________________________________________________________________________________
conv5_block3_out (Activation)   (None, 2, 2, 2048)   0           conv5_block3_add[0][0]
__________________________________________________________________________________________
global_average_pooling2d (Globa (None, 2048)         0           conv5_block3_out[0][0]
__________________________________________________________________________________________
dense (Dense)                   (None, 128)          262272      global_average_pooling2d[0][0]
__________________________________________________________________________________________
dropout (Dropout)               (None, 128)          0           dense[0][0]
__________________________________________________________________________________________
dense_1 (Dense)                 (None, 64)           8256        dropout[0][0]
__________________________________________________________________________________________
dense_2 (Dense)                 (None, 7)            455         dense_1[0][0]
==========================================================================================
Total params: 23,858,695
Trainable params: 23,703,879
Non-trainable params: 154,816
```

Figure 6.64 Snapshot of final layer of Resnet & total parameter

Resnet was train on image classification of an object and our dataset is on facial emotion recognition there is a need to build different classification layer. Since we are using the Resnet as our feature extractor and we already cut off the top layer of Resnet layer, we need to construct ourselves the fully connected layer (top layer). Due to Resnet use skip connection in their architecture, the simple sequential model API will not work in Resnet.

Functional model API allows our model to connect any layers in the architecture. In functional model, the architecture make it possible for us to use complex network as part or all of the inputs directly connected to output layer as we can see in **Figure 6.63.**

Our fully connected layer contain global average pooling layer that will retain much of the information element in the block and blend it, 2 dense layer with 1 for classify the output and a dropout to avoid overfitting.

6.6.4 Training Configuration and Fit The Model

```
[ ]  model.compile(loss='categorical_crossentropy',
                   optimizer=keras.optimizers.SGD(1e-5),
                   metrics=['acc'])

[ ]  logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
     tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir="./logdir/resnet/", histogram_freq=1, profile_batch='15,20')

[ ]  history_res = model.fit(x_train_rgb50,
                     y_trainRN,
                     epochs=100,
                     batch_size=10,
                     validation_data=(x_val_rgb50, y_valRN),
                     callbacks = [tensorboard_callback])
```

Figure 6.65 Training Configuration

The configuration of our training will be similar to our previous VGG model except that we specify the learning rate to be small as this will in fact increase the performance of the model. The rest of the model configuration such as loss function categorical cross entropy and we monitor the accuracy metrics.

CHAPTER 7 : TUNING AND VALIDATION

As we foresee in our future Tensorflow project that that our model will be run on big data instead of smaller dataset we currently use, the use of input pipeline is important aspect of our big data analytics solution. In real life environment, our image data will come from distributed file system such as Hadoop or Spark that makes the Numpy or Pandas library that use in-memory computation less effective in manipulating the data. Therefore we will utilize Tensorflow tf.dataset API that able to process large dataset that do not fit in memory.

Once we finish preprocessing our data using Tensorflow dataset API, we will use Tensorflow hyperparameter tuning with Hparams Dashboard. This will help us to choose best combination of hyperparameter for our model and choose the best hyperparameter as the model performance is highly dependent on the choice of hyperparameter.

7.1 Input Pipeline

Input pipeline solve many of large and complex project in term of pre processing the data before feeding it to the models. Large complex data require different workflow from small project that can utilize in-memory computation without having any issues. The first problem storage problem where large data cannot be stored in local hard-disk or in-premise server and often required distributed file system such as Hadoop HDFS. This led to second problem, fetching the data from the source (i.e. HDFS) that require shuffling and batches. And final problem is often single accelerator (GPU or TPU) wont be enough to process large data.

Tensorflow Dataset API will help us to build more efficient data pipeline as it was build on ETL (Extract, Load, Transform) which suitable for big data pre-processing. This experiment data pipeline will follow ETL principle as followed:-

7.1.1 Extract

```
[ ]  #extract dataset into Tensorflow tf.data (API for input pipeline)
     train_ip1 = tf.data.Dataset.from_tensor_slices((x_train_rgb50, y_trainRN)).batch(32)
     valid_ip1 =tf.data.Dataset.from_tensor_slices((x_val_rgb50, y_valRN)).batch(32)
```

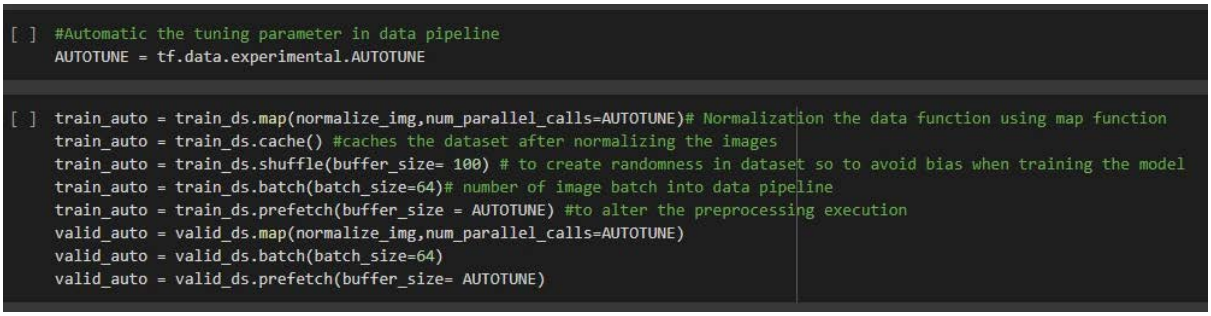
Figure 7.1 Extract data into tensorflow tf.data

In this extract process, due to our dataset is small and still fit into memory we just use from_tensor_slices () function that will convert our dataset into tf.Tensor objects. If we have large dataset we can scale up the extract process by using tf.data.TFRecordDataset("hdfs://path") to pull data from the HDFS cluster.

7.1.2 Transform

Below are the transformation technique in this experiment to transform data using tf input pipeline:-

- Shuffle - Prevent data from remembering pattern from training data that will make the model unable to generalize well.
- Batch - for very large dataset, processing small chunk of data at time will increase the performance speed due to the network didn't have to store all the error value of the images in memory.
- Prefetch - prefetching increase efficiency of accelerator (GPU/TPU) by preventing pipeline waiting for preprocessing by CPU to finish. its prefetch the data ahead of time to GPU to avoid valuable cycle doing nothing.
- Interleave - Interleave is similar to shuffle function except its shuffle the instance to avoid data learning from ordering of the data. Then there is AUTOTUNE which we let tensorflow to automatically choose the value dynamically at runtime.



```
[ ]  #Automatic the tuning parameter in data pipeline
     AUTOTUNE = tf.data.experimental.AUTOTUNE

[ ]  train_auto = train_ds.map(normalize_img,num_parallel_calls=AUTOTUNE)# Normalization the data function using map function
     train_auto = train_ds.cache() #caches the dataset after normalizing the images
     train_auto = train_ds.shuffle(buffer_size= 100) # to create randomness in dataset so to avoid bias when training the model
     train_auto = train_ds.batch(batch_size=64)# number of image batch into data pipeline
     train_auto = train_ds.prefetch(buffer_size = AUTOTUNE) #to alter the preprocessing execution
     valid_auto = valid_ds.map(normalize_img,num_parallel_calls=AUTOTUNE)
     valid_auto = valid_ds.batch(batch_size=64)
     valid_auto = valid_ds.prefetch(buffer_size= AUTOTUNE)
```

Figure 7.2 Transformation Process in Input Pipeline

7.1.3 Load

Once the data is in the pipeline, we can use it to feed our model without using in memory computation for the preprocessing which will save our valuable resources such as memory and improve the performance.

7.2 Hyper Parameter Tuning

Hyperparameter optimization or HParams Dashboard in tensorflow is another tools we introduce in experiment. Instead of manually guessing the hyperparameter or Keras default parameter, we applied hyperparameter tuning in addition to improve the model and to unlocking the maximum potential of the model.

For tuning our VGG, we experiment with the following hyperparameter

- Layer to unfreeze the pre-trained model
- Number of unit in dense layer
- The Dropout rate for dropout layer
- Finally the optimizer to be use in our model

For tuning our Resnet, we experiment with different hyperparameter:-

- Layer to unfreeze the pre-trained model
- Number of unit in dense layer
- The Dropout rate for dropout layer
- Learning rate in optimizer

To set up Hparams in our experiment, we followed the orders as below.

7.2.1 List the combination to try and log the configuration to Tensorboard

```
[ ]  #Hyperparameter 1st Run
     HP_FREEZE = hp.HParam('fine_tune_at', hp.Discrete([5, 9, 16]))
     HP_NUM_UNITS = hp.HParam('num_units2', hp.Discrete([256,512]))
     HP_DROPOUT = hp.HParam('dropout', hp.RealInterval(0.1,0.3))
     HP_OPTIMIZER = hp.HParam('optimizer', hp.Discrete(['adam', 'sgd']))

     METRIC_ACCURACY = 'accuracy'

     with tf.summary.create_file_writer('logs/hparam_vgg').as_default():
       hp.hparams_config(
         hparams=[HP_FREEZE,HP_NUM_UNITS, HP_DROPOUT, HP_OPTIMIZER],
         metrics=[hp.Metric(METRIC_ACCURACY, display_name='Accuracy')],
       )
```

Figure 7.3 Combination & Tensorboard log setting

As per **Figure 7.3**, we first set combination of the hyperparameter that we want to test and find the specify the metric to be display on the Tensorboard. Then we create a file that log the progress and result for Tensorboard to call. For additional information, there are other platform to choose other than Tensorboard such as Wandb, MLflow, Facebook Visdom and Neptune that able to visualize and has different user interface that we can write outside of Tensorflow. This platform can offer different filtering of the hyperparameter tuning that we can choose.

7.2.2 Define the choosen model in function

```
[ ]  def train_test_resnetmodel(hparams):
        resnet_base = VGG16(include_top= False, weights='imagenet', input_shape = (48,48,3))
        resnet_base.trainable = True #False = Freeze all | True = Train Selected Layer

        #Fine-tune from this layer onwards (make sure to unfreeze layer first before finetune)
        fine_tune_at = hparams[HP_FREEZE] # backprop will tweak layer 20 to 175

    # Freeze all the layers before the `fine_tune_at` layer
        for layer in resnet_base.layers[:fine_tune_at]:

            layer.trainable =  False

    #Adapt TensorFlow run & log it into tensorboard
    def train_test_model(hparams):
      vgghp_model = tf.keras.models.Sequential([tf.keras.applications.VGG16(include_top= False, weights='imagenet', input_shape = (48,48,3)),
                                        tf.keras.layers.Flatten(),
                                        tf.keras.layers.Dense(hparams[HP_NUM_UNITS], activation='tanh'),
                                        tf.keras.layers.Dropout(hparams[HP_DROPOUT]),
                                        tf.keras.layers.Dense(7, activation=tf.nn.softmax), # 7 classifications
                                       ])

      vgghp_model.compile(optimizer=hparams[HP_OPTIMIZER],
                    loss='categorical_crossentropy',
                    metrics=['accuracy'],
      )

      history = vgghp_model.fit(train_auto,epochs=15)
      validation_data=(valid_auto),

      return history.history['accuracy'][-1]
```

Figure 7.4 Define the choosen model

During this model, we plug in the variable that we specify on previous step into the layer of choosen model. For example, in dense layer we plug in the variable h.params[HP_NUM_UNITS].

7.2.3 Log the summary and run the tuning process.

```
[ ]  #For each run, log an hparams summary with the hyperparameters and final accuracy:
     def run(run_dir, hparams):
       with tf.summary.create_file_writer(run_dir).as_default():
         hp.hparams(hparams)  # record the values used in this trial
         accuracy = train_test_model(hparams)
         accuracy= tf.reshape(tf.convert_to_tensor(accuracy), []).numpy()
         tf.summary.scalar(METRIC_ACCURACY, accuracy, step=10)
```

```
[ ]  #Start run & log them
     session_num = 0
     for fine_tune_at in HP_FREEZE.domain.values:
       for num_units in HP_NUM_UNITS.domain.values:
         for dropout_rate in (HP_DROPOUT.domain.min_value, HP_DROPOUT.domain.max_value):
           for optimizer in HP_OPTIMIZER.domain.values:
             hparams = {
               HP_FREEZE: fine_tune_at,
               HP_NUM_UNITS: num_units,
               HP_DROPOUT: dropout_rate,
               HP_OPTIMIZER: optimizer,
           }
           run_name = "run-%d" % session_num
           print('--- Starting trial: %s' % run_name)
           print({h.name: hparams[h] for h in hparams})
           run('logs/hparam_vgg/' + run_name, hparams)
           session_num += 1
```

Figure 7.5 Log and Run tuning process

Once the line is executed, it will run multiple experiment with different set of hyperparameter that we set previous. This technique called grid search and more efficient that other optimization technique such as random search or manual search. More advance technique of hyperparameter tuning are using algorithm in case of Bayesian optimizer that used gaussian processes.

As precaution when running the hyperparameter, we only set up the combination as minimum as possible even though the correct way to do hyperparameter is to test every possible combination of hyperparameter in the model. This is because the iteration process required extensive computation resources and time. For example, in **Figure 7.6** our VGG tuning run into 11 combination of our vgg hyperparameter. This takes more than 1 hour to complete where each run has epoch of 15 and each epoch takes almost 28 second to finish (28x15x11 = 4620 sec or 77 minute or 1 hour 17 minute to complete).

```
--- Starting trial: run-11
{'fine_tune_at': 16, 'num_units2': 512, 'dropout': 0.3, 'optimizer': 'sgd'}
Epoch 1/15
41/41 [==============================] - 28s 674ms/step - loss: 1.9092 - accuracy: 0.2307
Epoch 2/15
41/41 [==============================] - 28s 675ms/step - loss: 1.6985 - accuracy: 0.3331
Epoch 3/15
41/41 [==============================] - 28s 672ms/step - loss: 1.5962 - accuracy: 0.3768
Epoch 4/15
41/41 [==============================] - 28s 672ms/step - loss: 1.5096 - accuracy: 0.4231
Epoch 5/15
41/41 [==============================] - 27s 670ms/step - loss: 1.4026 - accuracy: 0.4687
Epoch 6/15
41/41 [==============================] - 28s 673ms/step - loss: 1.3650 - accuracy: 0.4837
Epoch 7/15
41/41 [==============================] - 28s 674ms/step - loss: 1.3244 - accuracy: 0.4993
Epoch 8/15
41/41 [==============================] - 28s 675ms/step - loss: 1.2875 - accuracy: 0.5090
Epoch 9/15
41/41 [==============================] - 28s 675ms/step - loss: 1.2536 - accuracy: 0.5278
Epoch 10/15
41/41 [==============================] - 28s 675ms/step - loss: 1.2070 - accuracy: 0.5469
Epoch 11/15
41/41 [==============================] - 28s 676ms/step - loss: 1.1932 - accuracy: 0.5520
Epoch 12/15
41/41 [==============================] - 28s 675ms/step - loss: 1.1617 - accuracy: 0.5678
Epoch 13/15
41/41 [==============================] - 28s 676ms/step - loss: 1.1812 - accuracy: 0.5541
Epoch 14/15
41/41 [==============================] - 28s 676ms/step - loss: 1.1219 - accuracy: 0.5775
Epoch 15/15
41/41 [==============================] - 28s 675ms/step - loss: 1.1093 - accuracy: 0.5821
```

Figure 7.6 Hyperparameter experiment with 11 combination of parameter

7.3 Save and Load Keras model

Once finish with the hyperparameter tuning, next step is to recalibrate our hyperparameter into our model and run model again using the best hyperparameter. At the end of the training we will save our trained model into google drive and load it back on our final stage of the experiment, the validation on random image.

To save the model, there are 3 option that we can save the trained model. Option 1 is to save entire model using model.save which will save model architecture, weights and configurations in single folder. Its can be save on 2 different format the HDF5(.h5) or SavedModel. The option 2 is just to save weights of the model and option 3 is to save its in json format.

```
#save all model
modelx.save('/content/drive/MyDrive/Fuad_Assignment/resnet_model(All)v1.h5')# save in hdfs format
tf.keras.models.save_model(modelx,'/content/drive/MyDrive/Fuad_Assignment/resnet_tfmodel(All)v1')# save in Tensorflow format
#save weight only
modelx.save_weights('/content/drive/MyDrive/Fuad_Assignment/resnet_model(weight)v1.h5')
#save architecture only
json_string=modelx.to_json()
```

Figure 7.7 Save the trained model

7.4 Validation on Random Image

Final part of the experiment is testing outside experiment environment for evaluating real time performance against some random image. this is to validate that our training models can performed well against unseen data that not come from the same sources.

7.4.1 Load Trained Model And Define Function for Visualization

```
[ ] #Load train model
    from keras.models import load_model
    new_model = tf.keras.models.load_model('/content/drive/MyDrive/Fuad_Assignment/resnet_tfmodel(All)')

    #function for drawing bar chart for emotion preditions
    def emotion_analysis(emotions):
        objects = ('angry', 'disgust', 'fear', 'happy', 'sad', 'surprise', 'neutral')
        y_pos = np.arange(len(objects))

        plt.bar(y_pos, emotions, align='center', alpha=0.5)
        plt.xticks(y_pos, objects)
        plt.ylabel('percentage')
        plt.title('emotion')

        plt.show()
```

7.4.2 Load Random Image and Predict

```
#load some random image
from keras.preprocessing import image
img = tf.keras.preprocessing.image.load_img('/content/drive/MyDrive/Fuad_Assignment/jackman.png', grayscale=False, target_size=(48, 48) )#change image from google drive
#preprocess the random image

x = image.img_to_array(img)
x = np.expand_dims(x,axis=0)
x /= 255.
custom = new_model.predict(x)
emotion_analysis(custom[0])

#x = np.array(x, 'float32')
#x = x.reshape([48, 48]);

plt.gray()
plt.imshow(img)
plt.show()
```

CHAPTER 8 : RESULT AND ANALYSIS

8.1 Base Layer: ConvNet 1

Dataset 1 (4171 total, Training set = 2256, Validation Set = 1504 , Test Set = 418)

Architecture: - Sequential (Feature Extraction consist of 3 Conv2D, 3 Maxpooling2D & Fully Connected layer consist of 1 Flatten, 2 Dense)

Training Configuration:- Early Stopping (monitor = 'val loss',Min_delta(0.01), Patience (10)), Optimizer (SGD)

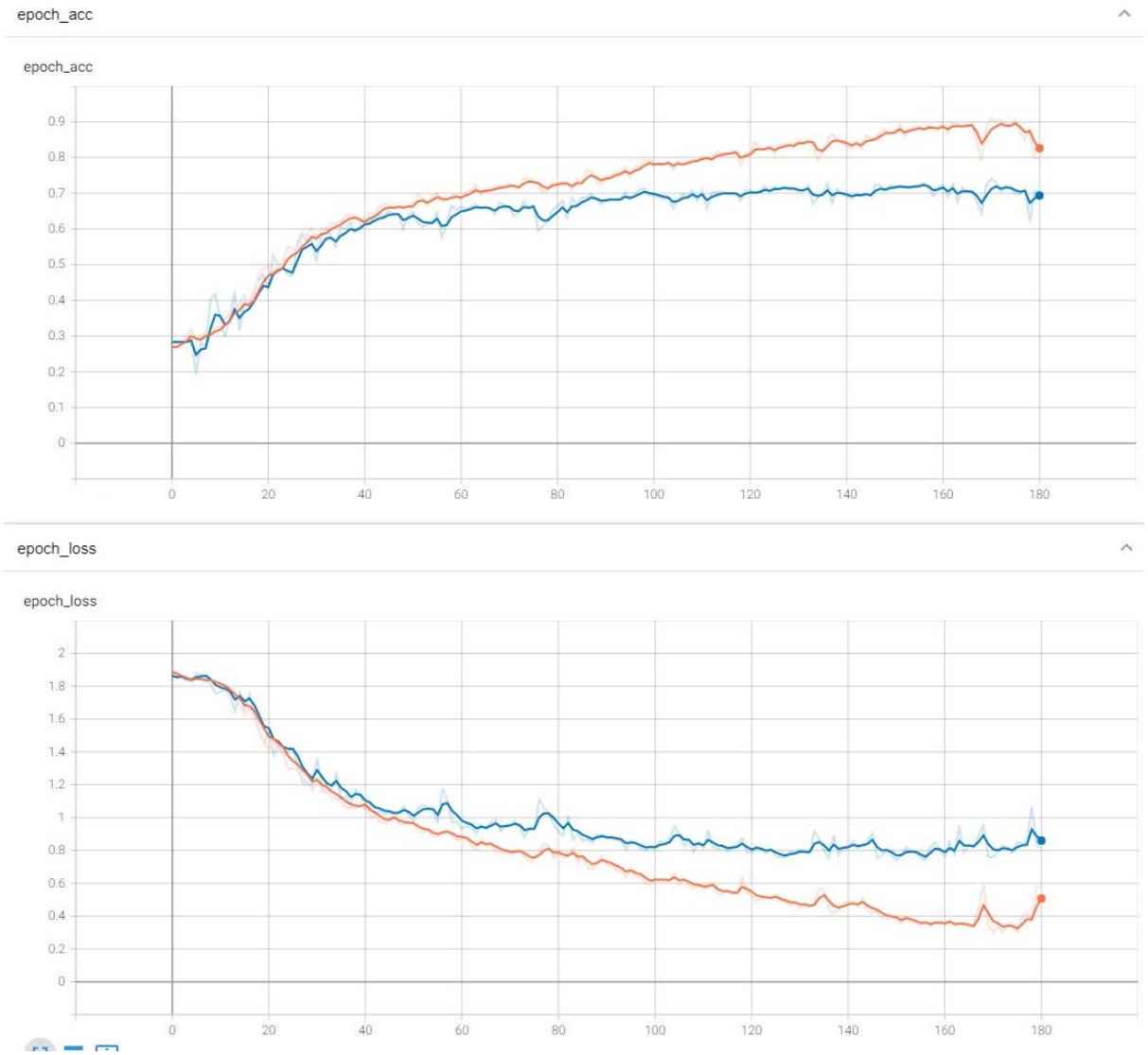


Figure 8.1.1 Epoch Accuracy & Epoch Loss Graph on DataSet 1 (ConvNet)

Dataset 2 (28709 total, Training set = 15502, Validation Set = 10336, Test Set=2871)

Architecture: - Sequential (Feature Extraction consist of 4 Conv2D, 4 Maxpooling2D & Fully Connected layer consist of 1 Flatten, 3 Dense layer, 1 Dropout)

Training Configuration: - Early Stopping (monitor = 'val loss',Min_delta(0.01), Patience (10)), Optimizer (SGD)

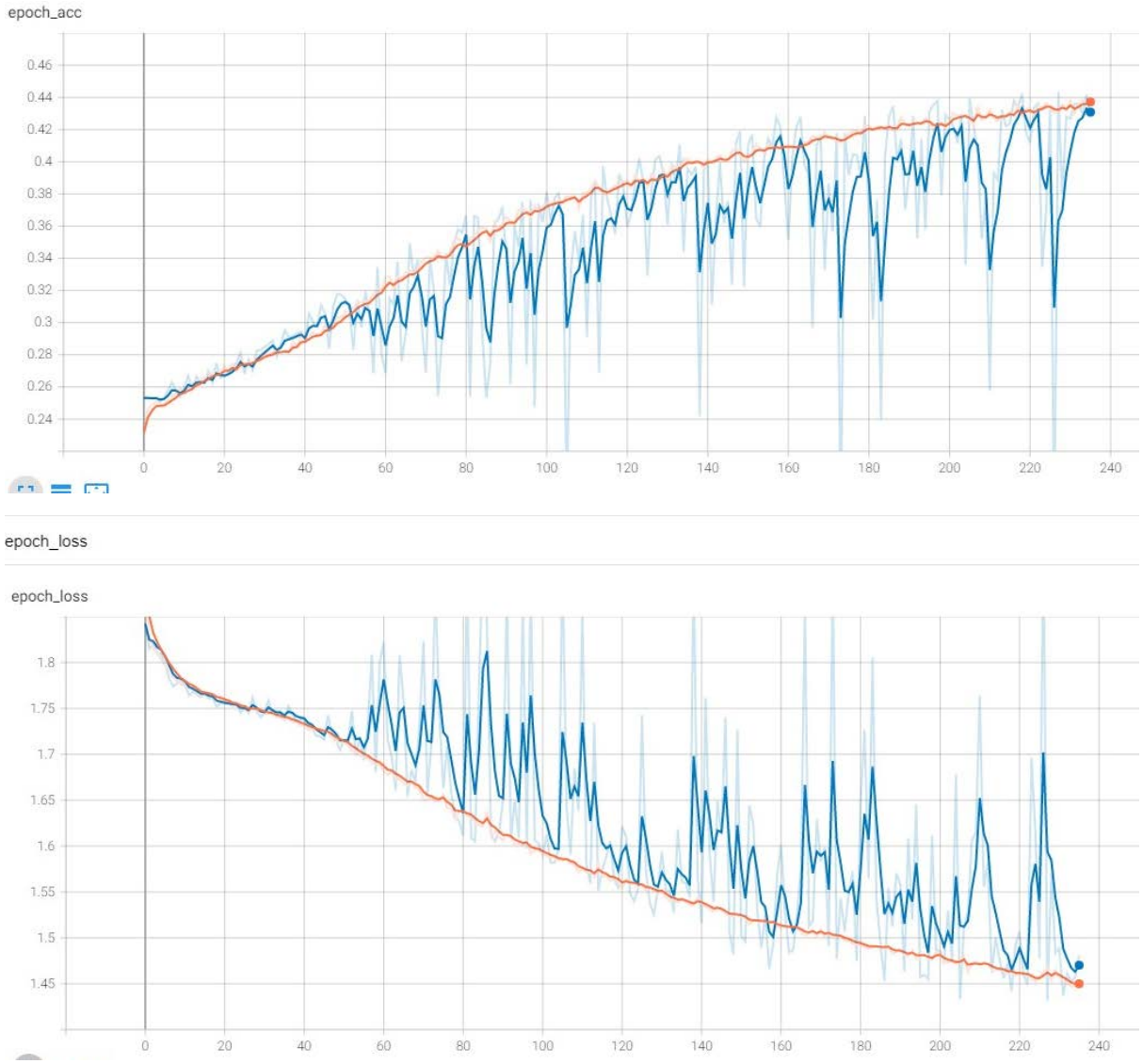


Figure 8.1.2 Epoch Accuracy & Epoch Loss Graph on DataSet 2 (ConvNet)

8.1.1 Summary Experiment

Dataset Size	Accuracy	Loss	Epoch
Small (4171)	Train:80% Validation:70% Test:72.49%	Train:1.181 Validation:1.898 Test:0.7810	181 steps
Large (28709)	Train:43.92% Validation:42.76% Test:43.75%	Train:1.45 Validation:1.481 Test:1.4656	236 steps

Table 8.1 Table of Summary Experiment (ConvNet)

8.2 VGG Model

Dataset 1 (4171 total, Training set = 2256, Validation Set =1504 , Test Set =418)

Fully Connected Architecture:- Sequential **(**Feature Extraction consist of VGG Pre Trained Layer & Fully Connected Layer consist of 1 GlobalAveragePool2D, 2 Dense, 1 Dropout) Activation Function Tanh & Softmax (Final Dense Layer)

Training Configuration:- Pretrained Network (VGG) Freeze 17/19, Batch Normalization with momentum(0.5), Dropout, Early Stopping (monitor = 'val loss',Min_delta(0.01), Patience (10) **)**

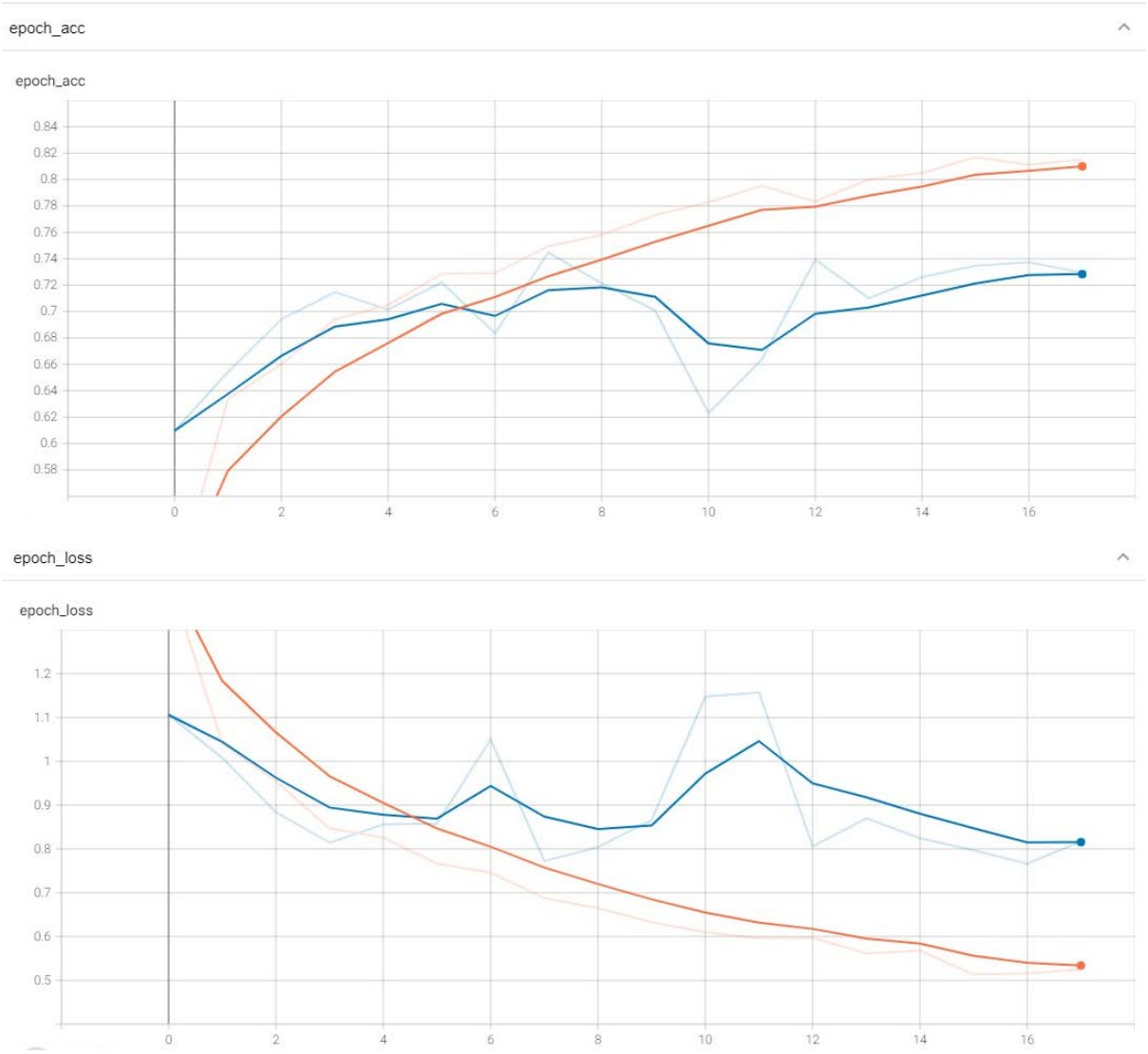


Figure 8.2.1 Epoch Accuracy & Epoch Loss Graph on DataSet 1 (VGG)

Dataset 2 (28709 total, Training set = 15502, Validation Set =10336 , Test Set=2871)

Fully Connected Architecture: - Sequential (Feature Extraction consist of VGG & Fully Connected Layer consist of 1 GlobalAveragePool2D, 4 Dense, 4 Batch Normalization, 3 Dropout) Activation Function Tanh & Softmax (Final Dense Layer)

Training Configuration:- Pretrained Network (VGG), Batch Normalization with Momentum (0.5), Optimizer SGD (Learning Rate= 0.01 & Momentum 0.5) Early Stopping (monitor = 'val loss',Min_delta(0.01), Patience (10) **)**

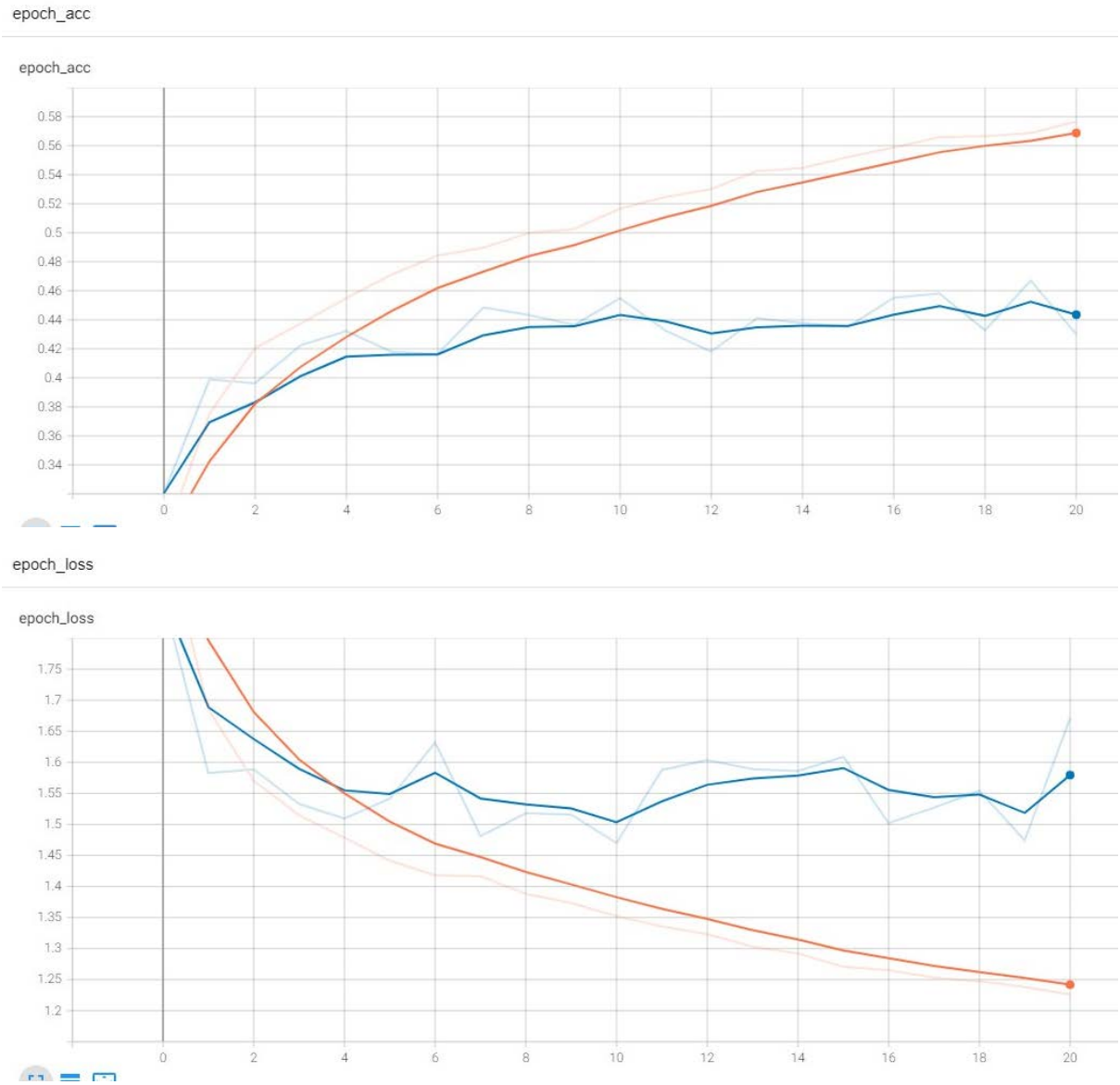


Figure 8.2.2 Epoch Accuracy & Epoch Loss Graph on DataSet 2 (VGG)

8.2.1 Summary Experiment

Dataset Size	Accuracy	Loss	Epoch
Small (4171)	Train:81.52% Validation:72.94% Test:72.97%	Train:0.5247 Validation:0.8170 Test:0.7737	18 steps
Large (28709)	Train:57.66% Validation:43.01% Test:43.96%	Train:1.226 Validation:1.671 Test:1.6582	21 steps

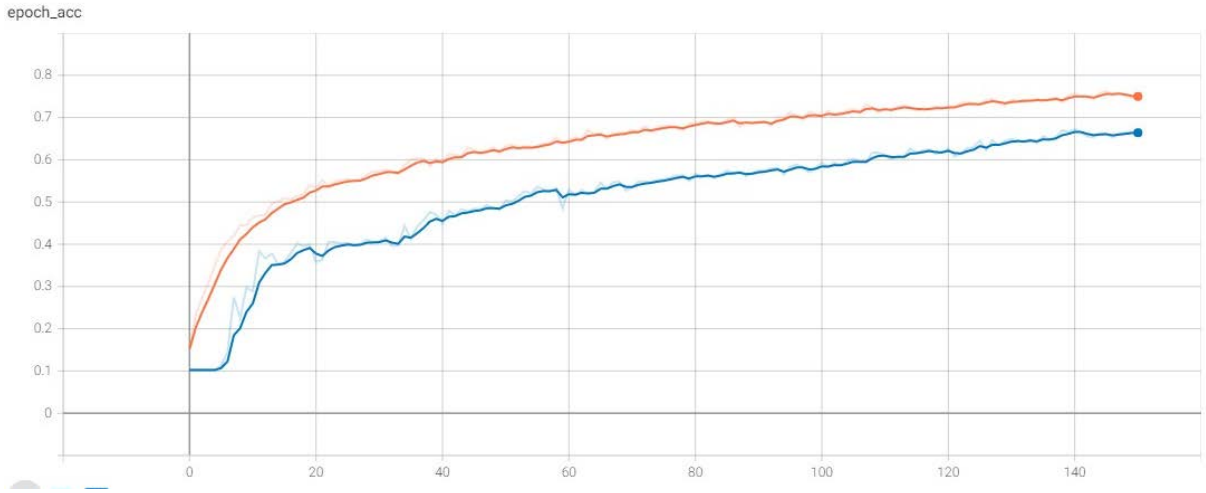
Table 8.2 Table of Summary Experiment (VGG)

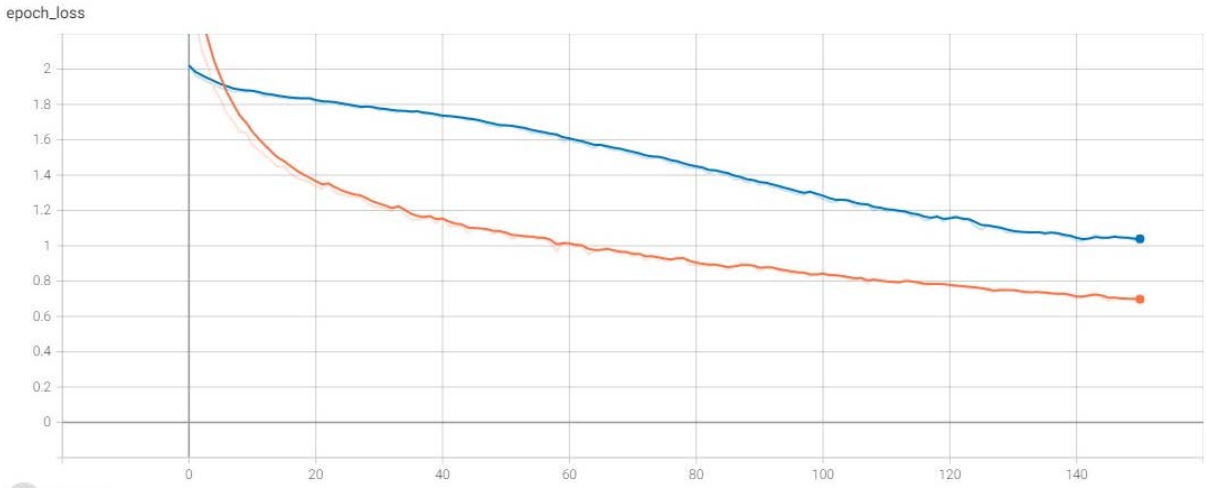
8.3 RESNET50 Model

Dataset 1 (4171 total, Training set = 2256 , Validation Set = 1504, Test Set =418)

Fully Connected Architecture:- Functional (Feature Extraction consist of Resnet & Fully Connected Layer consist of 1 GlobalAveragePool2D, 2 Dense, 1 Dropout) Activation Function Tanh & Softmax (Final Dense Layer)

Training Configuration:- Pretrained Network (Resnet50) Unfreeze(172 onward), Batch Normalization with Momentum (0.5), Optimizer SGD (Learning Rate= 0.01 & Momentum 0.5) Early Stopping (monitor = 'val loss',Min_delta(0.01), Patience (10) **)**






```
#evaluate the model using test set
model.evaluate(x_test_rgb50,y_test_r)
model
```

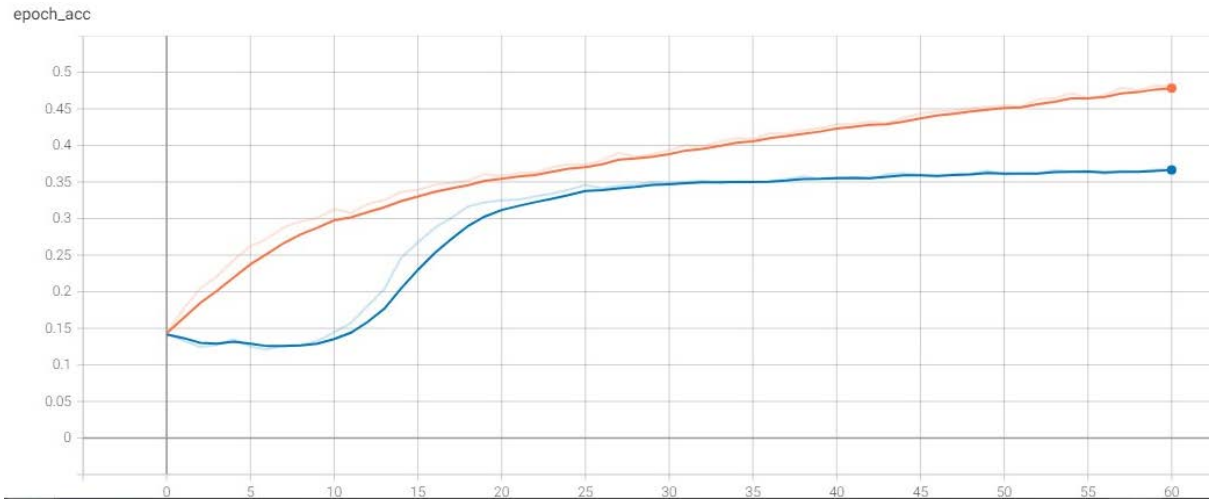
```
20/20 [==============================] - 1s 24ms/step - loss: 1.0204 - acc: 0.6093
<tensorflow.python.keras.engine.functional.Functional at 0x7ff986cf4860>
```

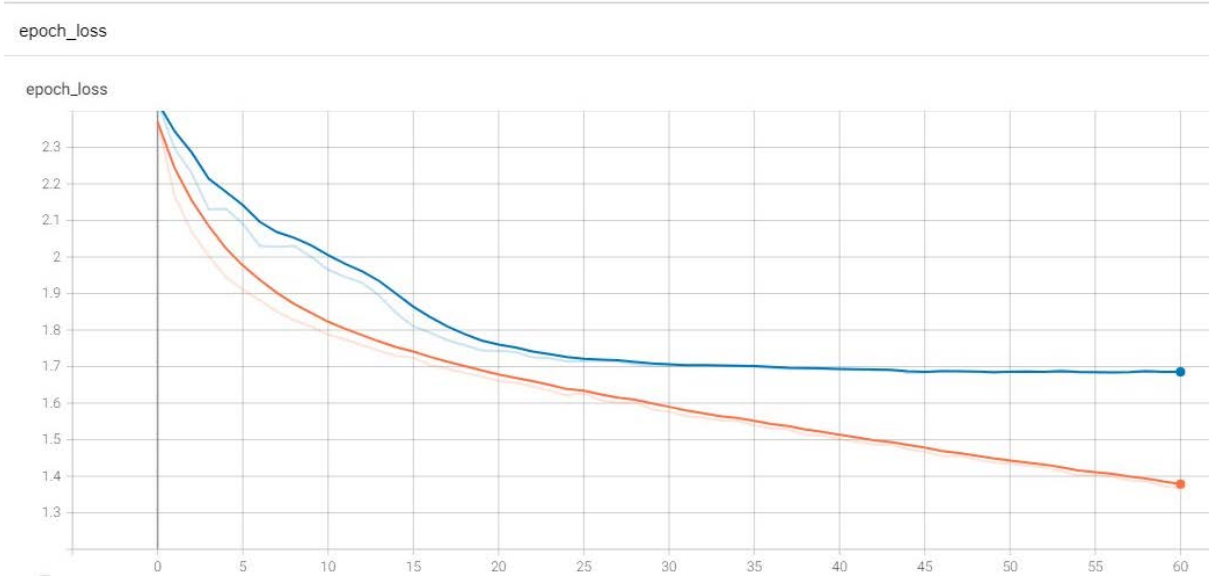
Figure 8.3.1 Epoch Accuracy & Epoch Loss Graph on DataSet 1 (Resnet50)

Dataset 2 (28709 total, Training set = , Validation Set = , Test Set=)

Fully Connected Architecture:- Functional (Feature Extraction consist of Resnet50 & Fully Connected Layer consist of 1 GlobalAveragePool2D, 4 Dense, 4 Batch Normalization, 3 Dropout) Activation Function Tanh & Softmax (Final Dense Layer)

Training Configuration :- Pretrained Network (Resnet50) Unfreeze(172 onward), Batch Normalization with Momentum (0.5), Optimizer SGD (Learning Rate= 0.01 & Momentum 0.5) Early Stopping (monitor = 'val loss',Min_delta(0.01), Patience (10) **)**






```
[26] #evaluate the model using test set
     model.evaluate(x_test_rgb50,y_test_r)
     model

     135/135 [==============================] - 2s 15ms/step - loss: 1.7009 - acc: 0.3522
     <tensorflow.python.keras.engine.functional.Functional at 0x7f78a5a34da0>
```

Figure 8.3.2 Epoch Accuracy & Epoch Loss Graph on DataSet 2 (Resnet50)

8.3.1 Summary Experiment

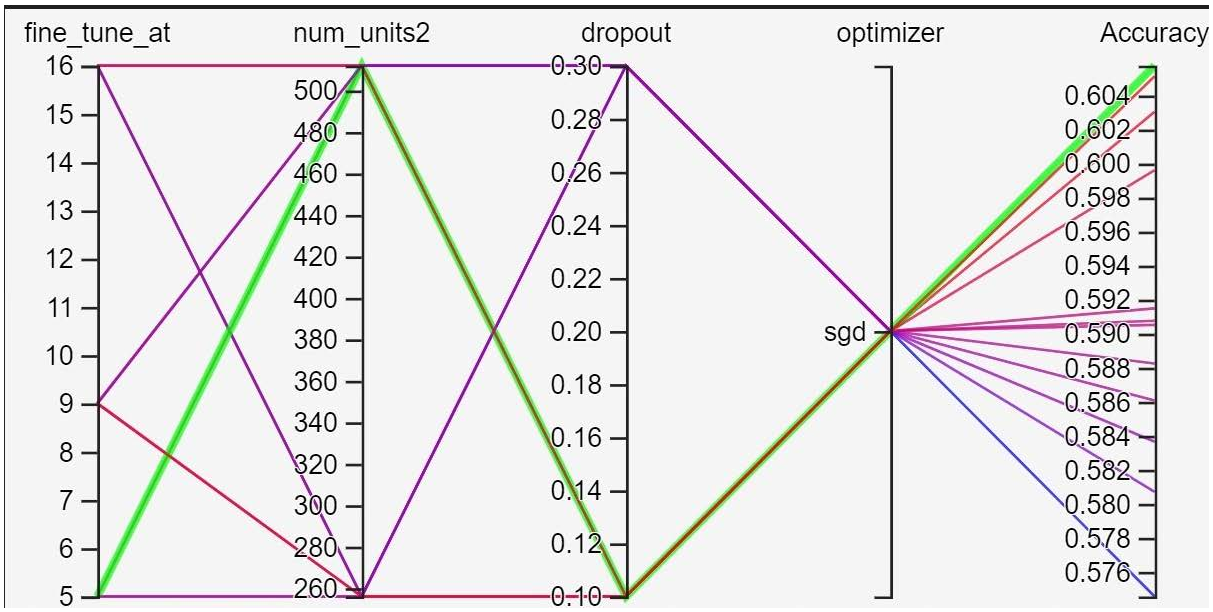
Table 8.3 Table of Summary Experiment

Dataset Size	Accuracy	Loss	Epoch
Small (4171)	Train:74.81% Validation:66.51% Test:60.93%	Train:0.6976 Validation:1.037 Test:1.0204	151 steps
Large (28709)	Train:47.82% Validation:36.84% Test:35.22%	Train:1.368 Validation:1.686 Test:1.7003	60 steps

Table 8.3 Table of Summary Experiment (Resnet50)

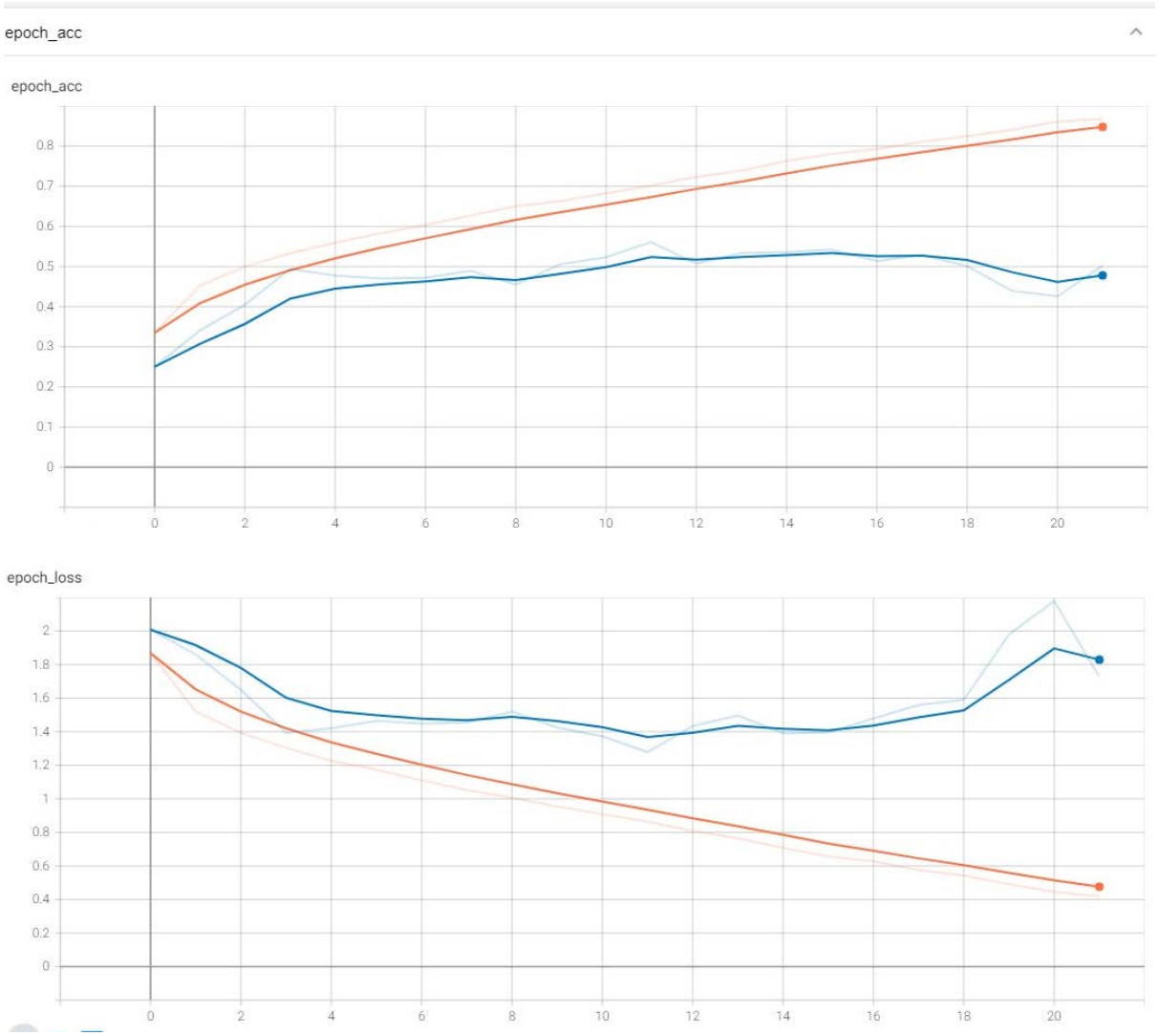
8.4 Hyperparameter Tuning Result

8.4.1 VGG (Best Combination = Fine tune at (5 layer), Dense unit (512), Dropout (0.1), Optimizer (SGD)



Trial ID	fine_tune_at	num_units2	dropout	optimizer	Accuracy
c24875b70c0f1...	5.0000	512.00	0.10000	sgd	0.60576

Figure 8.4.1 Hyperparameter Tuning Result Dataset 2(VGG)




```
#evaluate the model using test set
model_vggbest.evaluate(x_test_rgb,y_test)
model_vggbest
```

```
90/90 [==============================] - 2s 15ms/step - loss: 1.7208 - acc: 0.5064
<tensorflow.python.keras.engine.sequential.Sequential at 0x7f7bde2059b0>
```

Figure 8.4.2 Hyperparameter Epoch Accuracy & Epoch Loss Graph on DataSet 2 (VGG)

8.4.2 Summary Experiment

Dataset Size	Accuracy	Loss	Epoch
Large (28709)	**Train:86.76%** **Validation:47.79%** **Test:50.64%**	**Train:0.4188** **Validation:1.829** **Test:1.7208**	**21**

Table 8.4 Table of Hyperparameter Summary Experiment (VGG)

8.4.2 Resnet (Best Combination = Fine tune at (5 layer), Dense unit (512), Dropout (0.1), Optimizer (SGD

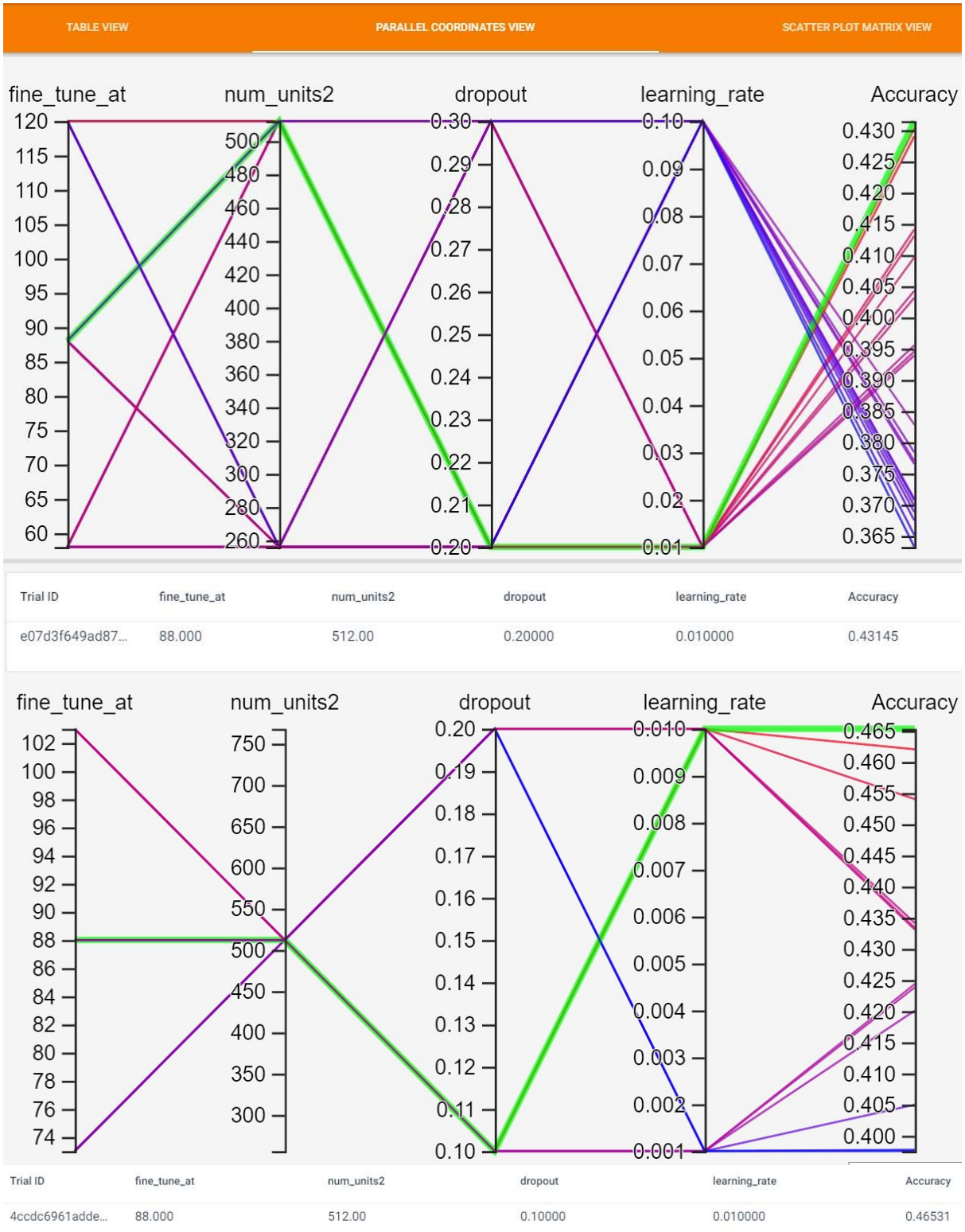
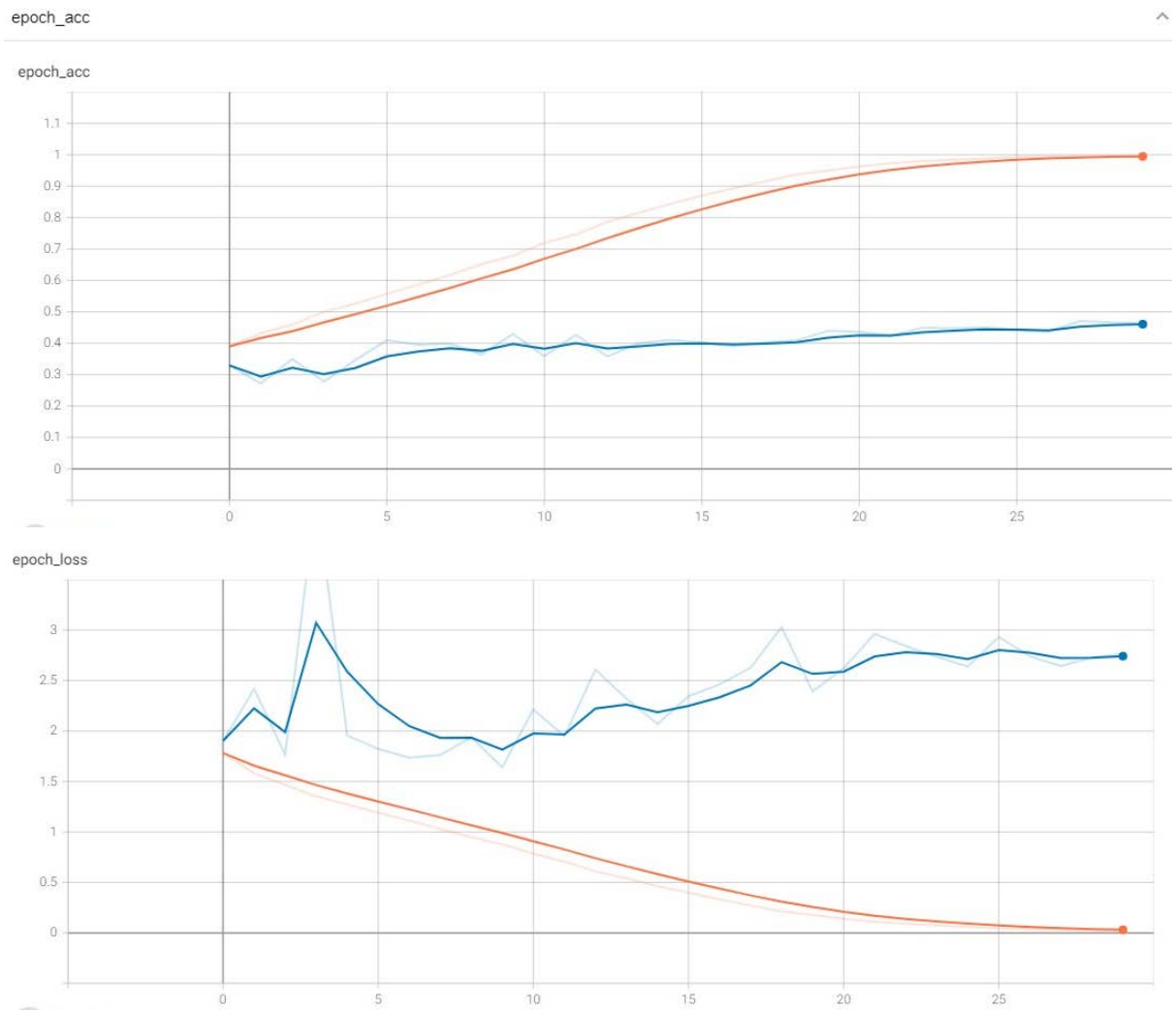


Figure 8.5.1 Hyperparameter Tuning Result Dataset 2(Resnet50). Above- First Run, Below- Second Run (Refinement)




```
#evaluate the model using test set
modelx.evaluate(x_test_rgb50,y_test_r)
modelx
```

```
135/135 [==============================] - 2s 15ms/step - loss: 2.7480 - acc: 0.4634
<tensorflow.python.keras.engine.functional.Functional at 0x7f78104b03c8>
```

Figure 8.5.2 Hyperparameter Epoch Accuracy & Epoch Loss Graph on DataSet 2 (Resnet50)

8.4.2 Summary Experiment

Dataset Size	Accuracy	Loss	Epoch
Large (28709)	**Train:99.67% Validation:46.07% Test:46.34%**	**Train:0.021 Validation:2.741 Test:2.74**	**30**

Table 8.5 Table of Hyperparameter Summary Experiment (Resnet)

8.5 Best Result Summary

	Top Accuracy on Test Set (%)	
Base Architecture	**FER2013 (4178)**	**FER2013 (28709)**
ConvNet	72.49	43.75
VGG16	72.97	43.96/50.64(*FT)
ResNet50	60.63	35.22/46.34(*FT)
***FT -After Hyperparameter Tuning**		

Table 8.6 Summary of Experiment on Test data/Hold Out Data

8.6 Model Validation

8.6.1 Visualization of the result

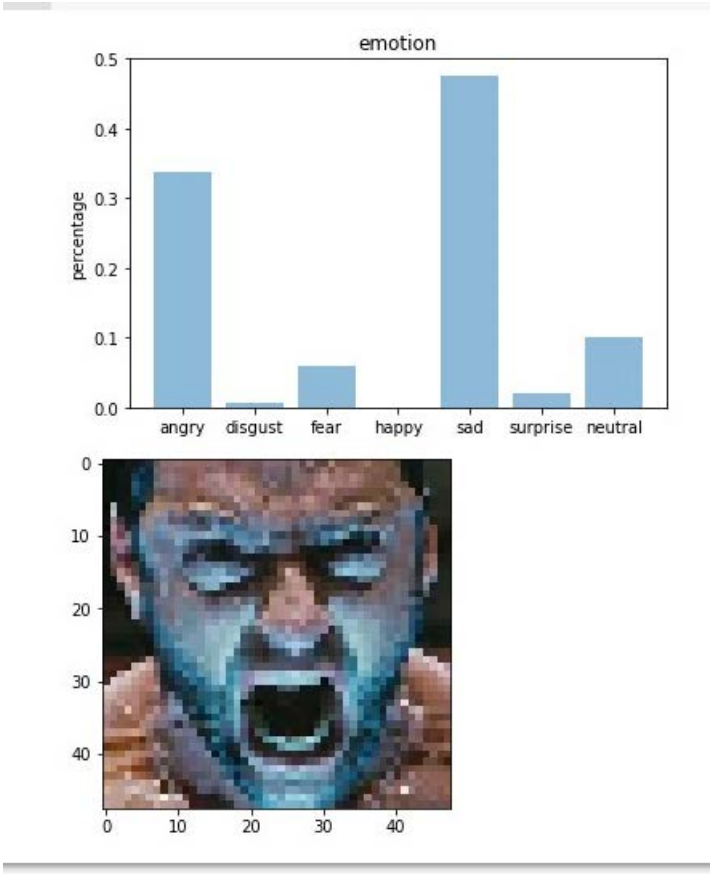


Figure 8.7.1 Model predicted 48% Sad & 35% Anger Face

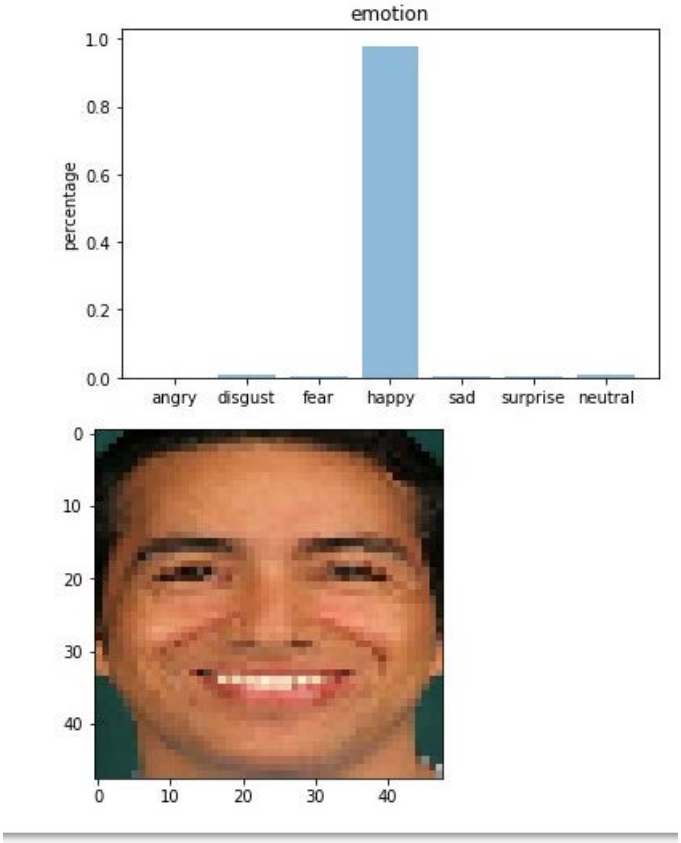


Figure 8.7.2 Model Predicted 95% Happy Face

8.7 Critical Analysis

8.7.1 Top Layer Classifier

In our experiment, we test several top layer/fully connected layer for our dataset and the best top layer is as per **Figure 8.8.1** for small dataset and **Figure 8.8.2** for large dataset. We find out that using Global Average Pooling is the best layer to down sampling for our architecture compare to Flatten layer. Its work by compute the mean of entire feature map and drop remaining spatial information compare to Flatten layer that just reshape each input image into 1 dimension array.

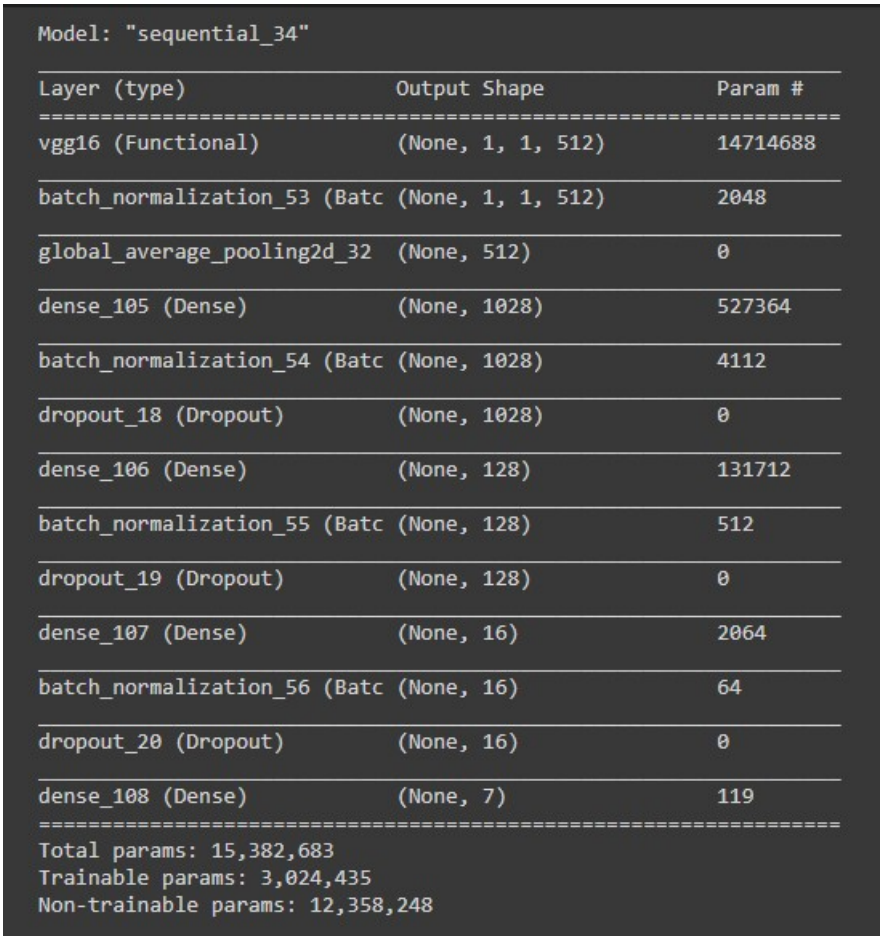
Our next layer is Dense layer. For small dataset we just use 2 dense layer with 128 node/neuron on the first layer and our 7 for our prediction layer. For large dataset, we use 4 dense layer with the first dense layer have 1028 node followed by 128, 16 and finally our prediction layer has 7 nodes. The reason we use more dense layer for our bigger dataset is because larger dataset has more dimension and complexity therefore more nodes will help our model to find signal in thousands of potential features.

The use of regularization in our architecture is to avoid the model behave erratically during training and control overfitting thus improve model generalization performance. There are 2 regularization we used in this architecture, Dropout and Batch Normalization. We find out that placing the regularization layer between each dense layer add stability to the model during training and avoid the exploding gradient that we often face when training with both pre-trained model.

During the training of our model, we use large batch (512) to reduce our steps per epoch and speed up our training time per epoch. However using large batch will result our model to unable to converge thus resulting poor generalization of the data. Acknowledging the problem, we offset the problem by using more slower learning rate(0.01) and introduce momentum(0.5) hyperparameter.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 1, 1, 512)         14714688
_________________________________________________________________
global_average_pooling2d_1 ( (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 128)               65664
_________________________________________________________________
batch_normalization_1 (Batch (None, 128)               512
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_3 (Dense)              (None, 7)                 903
=================================================================
Total params: 14,781,767
Trainable params: 2,426,631
Non-trainable params: 12,355,136
_________________________________________________________________
```

Figure 8.8.1 Best Top Layer (Small Dataset)



```
Model: "sequential_34"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 1, 1, 512)         14714688
_________________________________________________________________
batch_normalization_53 (Batc (None, 1, 1, 512)         2048
_________________________________________________________________
global_average_pooling2d_32  (None, 512)               0
_________________________________________________________________
dense_105 (Dense)            (None, 1028)              527364
_________________________________________________________________
batch_normalization_54 (Batc (None, 1028)              4112
_________________________________________________________________
dropout_18 (Dropout)         (None, 1028)              0
_________________________________________________________________
dense_106 (Dense)            (None, 128)               131712
_________________________________________________________________
batch_normalization_55 (Batc (None, 128)               512
_________________________________________________________________
dropout_19 (Dropout)         (None, 128)               0
_________________________________________________________________
dense_107 (Dense)            (None, 16)                2064
_________________________________________________________________
batch_normalization_56 (Batc (None, 16)                64
_________________________________________________________________
dropout_20 (Dropout)         (None, 16)                0
_________________________________________________________________
dense_108 (Dense)            (None, 7)                 119
=================================================================
Total params: 15,382,683
Trainable params: 3,024,435
Non-trainable params: 12,358,248
```

Figure 8.8.2 Best Top Layer (Large Dataset)

8.7.2 Input Pipeline And Hyperparameter Tuning

There are several way to feed our model with data. The common way is to use feed_from_dict() or use more efficient way by use tf.data input pipeline. We used another Tensorboard functionality, TF Profiler to analyze our hardware (CPU & GPU) resources consumption and avoid performance bottlenecks that will effect our model to execute more faster. **Figure 8.8.3** showed the analysis of our resources utility. The profiler didn't found any bottleneck on our model due to the size of our dataset is consider small but there is improvement on kernel launch time (from 12.6% to 11.7%). However in future if we extract big data from distributed system such as Hadoop cluster, most common problem that we will face is performance bottleneck and spending time to find the bottleneck can be tedious. Therefore the use of tf.data input pipeline is crucial for our big data pipeline.

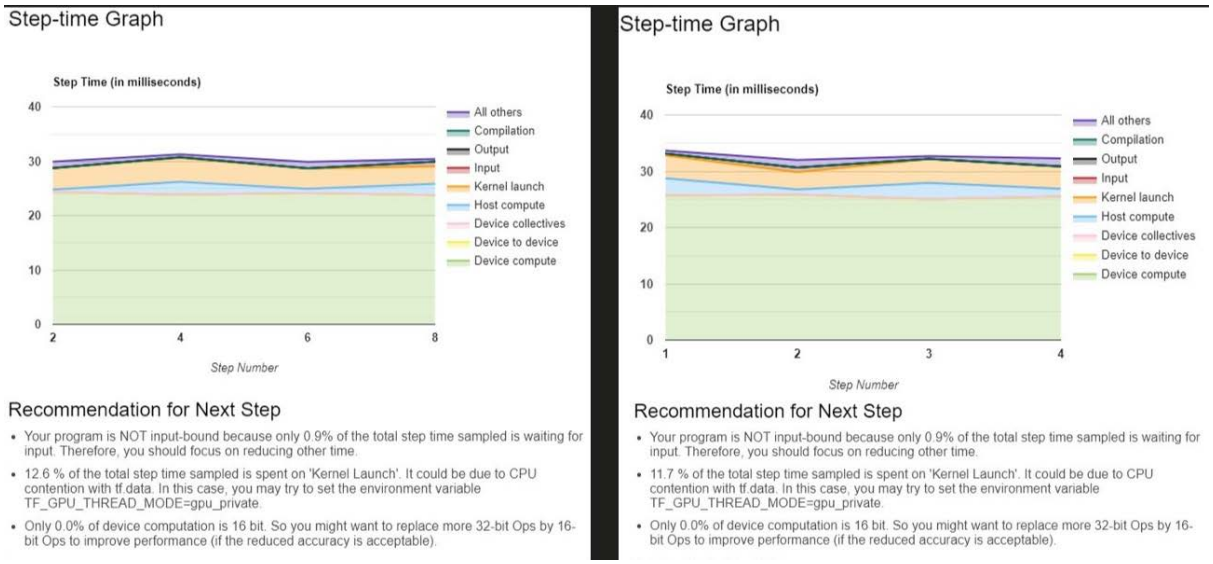


Figure 8.8.3 Tensorboard Profiler Analysis (Left- No input pipeline, Right- With Input Pipeline)

Hyperparameter tuning on the other hands, able to improve our our accuracy on both pre trained model such for VGG there is 15% increase in accuracy while Resnet there is 31.57% increase. However our Resnet suffer from overfitting quite badly due to the Resnet50 architecture is too deep (175 layer compare to 19 on vgg) and thus more parameter added that increase complexity to the model to generalize. This phenomenon can be observed based on paper published by Belkin et al. (2018) that explain that the more parameter added to the model, the training error decrease to close to zero but the test error increase. As **Figure 8.8.4** shown explain why our Resnet suffer from overfitting due to over-parameterized in the model.

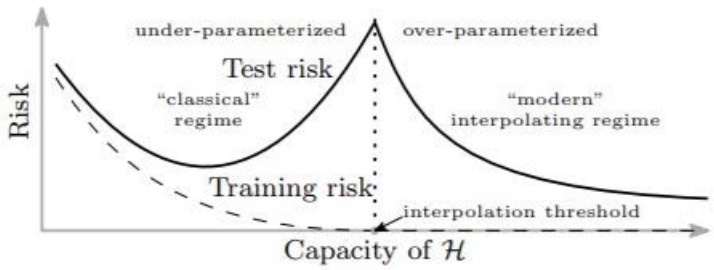


Figure 8.8.4 Model over parameterized (Belkin at el. 2018)

8.7.3 Validation and Comparation to Previous Work

At final stage of our experiment, we conduct validation test outside our experiment environtment by save and load the data and compare it with random image. Our model able to predict multiple emotion in a picture as in random image number 1 where it predict anger and sadness that appear in the image. We can conclude that its predict correctly due to there is emotion overlap that human express in their facial expression and in this case the model trying to predict an emotion that not being label (human has more than 7 emotion in real world).

Random image number 2 is more simple as smiling face is normally associate with happy expression. This shown in our model that its predict almost 95% on happy label.

Table x.xx shown the comparation of previous work with our model. 2 of the researcher using ConvNet that achieve 65% and 62% accuracy while another 2 researcher use transfer learning technique with the use of VGG architecture. As we can see in our experiment, when using ConvNet we able to achieve high accuracy but the time for gradient descent to converge it high. Compare with transfer learning using VGG, able we to shorten the epoch time to converge from 181 epoch into just 18 epoch and able to achieve same accuracy as ConvNet. The efficiency usage of resource are important criteria for us to deploy the model in our e-KYC system.

Researcher	Validation Acc (%)	Baseline Architecture	Remarks
Agrawal & Mittal (2020)	65.00	ConvNets	Did not use fully connected layer
Liu, Zhang &Pan (2016)	62.44	ConvNets	Use very high dense layer units (2x4096)
Knyazev et al. (2017)	37.9 /48.3(*FT)	VGG	Use small FER2013 dataset 4096
Bawa & Kumar (2019)	61.95	VGG	Custom build activation function (ALiSA) instead of RELU

* result after Fine Tune

Table 8.7 FER2013 Previous Work Comparation Result

8.8 Conclusion

In this experiment, we have constructed a model to predict facial expression recognition using transfer learning and we proposed a effective top layer to predict based on our dataset FER2013. Our model able to beat result of research done previously by utilizing hyperparameter tuning and the proposed method are compatible with big data pipeline framework. In future work, to improve our Resnet transfer learning result, we would like to use Resnet18 instead of Resnet50 as we believe residual layers is more efficient than VGG layer.

CHAPTER 9.0:- Reference

Agrawal, A. and Mittal, N., 2020. Using CNN for facial expression recognition: a study of the effects of kernel size and number of filters on accuracy. *The Visual Computer*, *36*(2), pp.405-412.

Liu, K., Zhang, M. and Pan, Z., 2016, September. Facial expression recognition with CNN ensemble. In *2016 international conference on cyberworlds (CW)* (pp. 163-166). IEEE.

Knyazev, B., Shvetsov, R., Efremova, N. and Kuharenko, A., 2017. Convolutional neural networks pretrained on large face recognition datasets for emotion classification from video. arXiv preprint arXiv:1711.04598.

Bawa, V.S. and Kumar, V., 2019. Linearized sigmoidal activation: A novel activation function with tractable non-linear characteristics to boost representation capability. Expert Systems with Applications, 120, pp.346-356.

Belkin, M., Hsu, D., Ma, S. and Mandal, S., 2019. Reconciling modern machine-learning practice and the classical bias–variance trade-off. Proceedings of the National Academy of Sciences, 116(32), pp.15849-15854.

Ekman, P., Freisen, W.V. and Ancoli, S., 1980. Facial signs of emotional experience. *Journal of personality and social psychology*, *39*(6), p.1125.