Predicting the Attention Rate of Patients by Monitoring their Emotions

- Name: Akshara Shukla
- Class: AI43
- Genuine Challenge_01

Introduction

This project consists of creating a report on aiding doctors in large multinational hospitals to understand different patient behaviors. Fundamentally, the project goal is to showcase the power of Artificial Intelligence and Machine Learning in healthcare and how to provide a better patient experience by classifying them on their emotions, mainly the four basic emotions (happy, sad, angry, and relaxing) and accompanying the ones that are in pain and require immediate or more attention and care. The system is Areya and it aims at providing some portrait of notification to the doctors about patients who illustrate negative emotions i.e., angry and sad.

This notebook consists the training of the dataset FER-2013. The algorithm chosen was building a convolutional neural network.

Setup

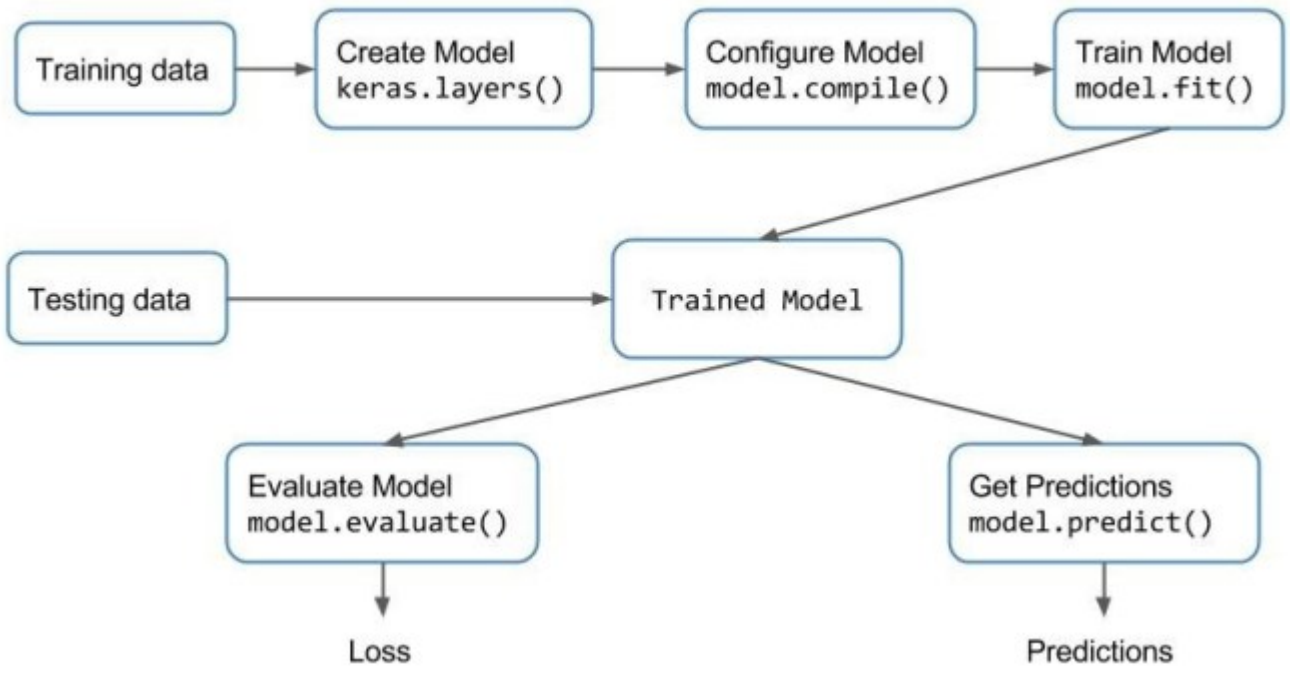
We start with importing needed/required libraries.

In [2]:

```
import keras                            #For working our neural network which provides us va
rious functions for building network.
from keras import models, layers        #for instantiating our model and forming differen
t layers.
from keras.models import Sequential     #for building a model with layers in sequence.
#from keras.layers import Conv2D, MaxPool2D, Flatten,Dense,Dropout,BatchNormalization
from keras.layers import Convolution2D, Activation, BatchNormalization, MaxPooling2D, Drop
out, Dense, Flatten, AveragePooling2D

from keras.layers.core import Dense,Flatten
from keras.preprocessing.image import ImageDataGenerator, load_img #for rescaling and impo
rting our image files from our pc's directory
from keras.optimizers import Adam,RMSprop  #the optimizer for minimising loss
from keras.metrics import categorical_crossentropy #the function of of our optimizer
from keras.layers.normalization import  BatchNormalization #for further normalization on o
ur images in different batches
from keras import regularizers
from tensorflow.keras.applications.vgg16 import VGG16 #pretrained model for image classifi
cation

#Importing the different callback API (functions) we will be using while training our mode
l.
from keras.callbacks import EarlyStopping
from keras.callbacks import CSVLogger
from keras.callbacks import ReduceLROnPlateau
from keras.callbacks import ModelCheckpoint

#Importing libraries for visualizations and numerical analysis
import matplotlib.pyplot as plt
import seaborn as sns
import sklearn.metrics
from sklearn.metrics import classification_report, confusion_matrix
import numpy as np

import random as rn
import tensorflow as tf

from ann_visualizer.visualize import ann_viz #visualizing our cnn model with it's layers
import matplotlib.ticker as mtick #for turning y axis interval to percentages
import os
os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/Graphviz/bin'   #importing grap
hviz library and installing it on our pc's path for forming our model
```

1- Defining Keras Model

Following is the lifecycle of building, training, testing and finally evaluating a Keras Model. This approach will be followed in this notebook.



2- Understanding CNN Architecture

The algorithm chosen to build our model Areya, is a custom Convolutional Neural Network (CNN). The algorithm is known for it's ability to detect vital patterns and offer high prediction rates while solving any problem which involves having an image as the input data. One of the greatest advantage of using CNNs is the reduced human efforts and is fast to implement. Initially, the definition and the reason for choosing to form a particular layer and assign values to our parameters has been defined.

Step 1: Importing & Creating a Sequential Model

We will be creating our own model as a **Sequential**. The sequential model allows us to create model in a layer by layer manner what that means is that there will be stacks of layers where each of the unique layer is connected and have their own set of input and output tensors each. Additionally, it's offers more flexibility to add or remove layers.

Our first step in building a sequential model, is to create our **convolutional** layers which are the first layer which our image's pixel values are fed and iniates the process. The important hyperparamters to set here are the *filters*, *kernel_size*, *activation* and the *input_size*. Lets understand each of these terms one by one:

> Our filters are actually what detects the patters (eg, multiple edges, texture, shapes, objects etc) and our kernel_size is the length and width of a small window which goes over each of the pixel values.We set the filters for our first layer as 64 and the kernel_size as 3x1, 1x3.
>
> An activation function is a function which determines the distribution of the weights of the input transforming to output layer from one node to other in a network. I chose *relu* which stands for *Rectified Linear Unit for non-linear operation* since it's known for performing outstandingly while combining the key performance parameters of the other known activation function.
>
> Lastly, the input shape which tells the shape of the image which is accepted by the model.

Step 3: Adding Batch Normalization and downsizing using MaxPooling

After adding our essential conv2d layers, adding a **BatchNormalization** technique which helps for scaling or standardizing the inputs to a layer in every batches formed. It helps the model to learn and regularizses the model and additionally reduces the effect of dropout.

Essentially, after scaling, adding a **MaxPooling2d** which helps in downsizing the input image by half (with the most important features) which helps the model understand complex problems.

Step 4: DropOut Layers

Tha addition of a **DropOut Layer** is done to specify the percentage of output features which will be randomly shut or ignored.

Step 5: Adding Dense Layers and Output Layers

The addition of a **Flatten** layer is followed which helps the model to flatten the input image. For example, if an input of shape (batch_size, 2,2) is applied to a layer, then the output shape of that layer would be (batch_size, 4).

Lastly, I have added the **Dense Layer** which are the hidden layers of my model. I have been increasing the filters in each layer with the power of 2 which are the default values. And assigning the final *activation function* as **softmax** which is used for the output layer of the neural network which predicts the probability distribution of each 4 outcomes (emotions).

3- Building CNN Architecture

After having learnt the principles and the vitality of each layers in the performance of the model, the building of the CNN is followed.

Firstly, I have defined a function **produce_same_results()** which performs similar functionality as that to random state and assigning stratify to our y_train in machine learning models. It helps in producing the same results after each run, since CNNs models are known for producing new results after every training session on one computer's/laptop's CPU, the positioning of this function was done before defining the structure of the model.

In [3]:

```
#Defining function to produce same results after every run.
#Setting pythonhashseed to a random string interger number since assigning to a int value,
it is used as a fixed seed for generating the hash() of the types covered by the hash rand
omization.

def produce_same_results():
    os.environ['PYTHONHASHSEED']=str(42) # Setting the PYTHONHASHSEED environment variable
to 0 before the model starts.
    tf.random.set_seed(42)    #Setting seed value to 42 which is the universal seed value
 used.
    np.random.seed(42)
    rn.seed(42)
```

In [4]:

```
#Calling the produce_same_results() function before starting to build the model.
produce_same_results()
#Setting the inputshape of the input layers
input_shape = (48,48,3)

#Instantiating my Sequential Model for building layers in a sequence.
model = Sequential()

#First part of the model having 2 Conv2d Layers with 64 batch size.
#Normalizing the pixel values of the images in this batch using BatchNormalization() and u
sing activation function as 'relu'
#MaxPooling,scaling our images with *pool_size* as 2x2 and strides as 1 since we want the
 kernel window to skip one pixel one time.
#Setting padding to same, because we want the kernel values to be efficiently fit.
#It is called SAME because, for stride 1 , to get the output image same as that of input.
model.add(Convolution2D(64, (3, 1), padding='same', input_shape=input_shape))
model.add(Convolution2D(64, (3, 3), padding='same', input_shape=input_shape))
#model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=1, padding='same'))


#Second part of the model having 2 Conv2d layers with 128 batch size.
model.add(Convolution2D(128, (3, 3), padding='same', input_shape=input_shape))
model.add(Convolution2D(128, (1, 3), padding='same', input_shape=input_shape))
#model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=1, padding='same'))
model.add(Dropout(0.1))

#Third part of the model having 2 Conv2d layers with 256 batch size.
model.add(Convolution2D(256, (3, 3), padding='same', input_shape=input_shape))
model.add(Convolution2D(256, (1, 3), padding='same', input_shape=input_shape))
#model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=1, padding='same'))

#Fourth part of the model having 2 Conv2d layers with 512 batch size.
model.add(Convolution2D(512, (3, 3), padding='same', input_shape=input_shape))
model.add(Convolution2D(512, (1, 3), padding='same', input_shape=input_shape))
#model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=1, padding='same'))
model.add(Dropout(0.1))
model.add(Flatten())

#Adding the first dense layer (hidden layer) with 512 neurons
model.add(Dense(512))
#model.add(BatchNormalization())
model.add(Activation('relu'))

#Adding the second dense layer with 256 neurons.
model.add(Dense(256))
#model.add(BatchNormalization())
model.add(Activation('relu'))
```

```
#Adding our last output layer with 4 neurons corresponding to the four emotions the model
 will be classifying with softmax activation used primarily for output layers.
model.add(Dense(4))
model.add(Activation('softmax'))
```

In [5]:

```
#Reading the summary of our model created
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 48, 48, 64)        640
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 48, 48, 64)        36928
_________________________________________________________________
activation (Activation)      (None, 48, 48, 64)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 48, 48, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 48, 48, 128)       73856
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 48, 48, 128)       49280
_________________________________________________________________
activation_1 (Activation)    (None, 48, 48, 128)       0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 48, 48, 128)       0
_________________________________________________________________
dropout (Dropout)            (None, 48, 48, 128)       0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 48, 48, 256)       295168
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 48, 48, 256)       196864
_________________________________________________________________
activation_2 (Activation)    (None, 48, 48, 256)       0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 48, 48, 256)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 48, 48, 512)       1180160
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 48, 48, 512)       786944
_________________________________________________________________
activation_3 (Activation)    (None, 48, 48, 512)       0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 48, 48, 512)       0
_________________________________________________________________
dropout_1 (Dropout)          (None, 48, 48, 512)       0
_________________________________________________________________
flatten (Flatten)            (None, 1179648)           0
_________________________________________________________________
dense (Dense)                (None, 512)               603980288
_________________________________________________________________
activation_4 (Activation)    (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               131328
_________________________________________________________________
activation_5 (Activation)    (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 1028
_________________________________________________________________
activation_6 (Activation)    (None, 4)                 0
=================================================================
Total params: 606,732,484
Trainable params: 606,732,484
```

```
Non-trainable params: 0
_________________________________________________________________
```

From the above summary, we can see we have three columns namely *Layers*,*Output Shape* and *Param #*.

> The Layer column essentially lists down the different layers we have in our model in their order of appearance. The Output Shape lists the shape of the image being trained in each layer.Here our size of of the images are (48,48,3). Now, keras library makes an extra dimension for processing or working with multiple batches i.e to train many images in one epoch. Since, batches sizes can vary from model to model, therefore it's size initially is represented by None. Hence, our output shape becomes (None,48,48,64) with 64 as our neurons for first convolutional layers.
>
> The Param # gives the number of parameters used. For a conv2d, the formula for calculating is (kernel_height *kernel_width* input_channels *output_channels) + (output_channels if bias is used). Which for our model, comes out as (3* 1 *3* 64) + 64 = 640.

Step 6: Compiling the Model

Now, after fully forming our model, we will be compiling using **Adam** as our optimizer because it combines the best features of other optimizers (AdaGrad and RMSProp). Metrics is **accuracy** because we want to evalute the model on it's accuracy. And, **categorical_crossentropy** is defined as categorical cross-entropy between an output tensor and a target tensor as our loss function. It measures the perfomance of classification models whose output lies as a probability between 0 and 1.

In [6]:

```
#Compiling our model formed above
#Adam(lr=0.001, decay=1e-6)
model.compile(loss = 'categorical_crossentropy', optimizer ='adam', metrics = ['accuracy'
])
```

Step 7: Assigning training and testing directories

In [7]:

```
#Paths to our train and test directory
train_dir = 'emotion_recognition/train/'
test_dir = 'emotion_recognition/test/'
```

Performing some scaling and image augmentation to our images by zooming on them, shearing them, and flipping etc, the **ImageDataGenerator** helps us augment and generate more images from the images already exisiting in our df. So, initially rescaling the pixel values and rotating our images in a range 40 with zooming and shearing on 20% and flipping the images horizontally for the model to learn from different angles. This helped the model to better understand the images and find key similarities and patterns.

In [8]:

```
#Performing important image augmenting techniques such as twisting and turning our images
 and generating new outputs on our train images.
train_path = ImageDataGenerator(rescale = 1./255.,
                                rotation_range = 40,
                                shear_range = 0.2,
                                zoom_range = 0.2,
                                horizontal_flip = True)

#Augmenting and generating new images in our test images.
test_path = ImageDataGenerator(rescale = 1/255.0)

#Creating training set of our model consisting pixel values and emotion indices.
#Using batch size as 32 for train and test to begin experimenting
train_set = train_path.flow_from_directory(train_dir, target_size = (48,48), batch_size =
32, class_mode = 'categorical')

##Creating test set of our model which will be used to test our model's accuracy.
test_set = test_path.flow_from_directory(test_dir, target_size = (48,48), batch_size = 32,
class_mode = 'categorical')
```

```
Found 26217 images belonging to 4 classes.
Found 5212 images belonging to 4 classes.
```

In [9]:

```
train_set.class_indices
```

Out[9]:

```
{'Angry': 0, 'Happy': 1, 'Relaxed': 2, 'Sad': 3}
```

From the above chunk, we can conclude our **training set** has a total of **26,217 images** belonging to 4 classes which are the **four unique emotions ('Angry': 0, 'Happy': 1, 'Relaxed': 2, 'Sad': 3)** . Similarly, our **test set** has a total of **5212 images** of 4 emotion indices.

Step 8: Using Callback functions

A **callback function** is a function that helps in performing various tasks while the training is happening.

a) We will be using **EarlyStop** which is great for solving overfitting problems as it stops the training process if our monitored value *val_loss* has stopped improving after certain epochs which in our case is defined as 4 *patience*.

b)**CSVLogger** which streams and saves results of accuracy and loss of each epoch in a CSV file.

c) **ReduceLRPlateau** which reduces the learning rate once a metric has stopped improving or rather when it has fully learnt.

d) And **ModelCheckPoint** which saves the model with the value being monitered and saving only the best output.

In [10]:

```
#EarlyStop Callback
earlystop = EarlyStopping(monitor = 'val_loss',
                          min_delta = 0,    #training process will be stopped if the absolu
te change of val_loss is less than 0
                          patience = 4,
                          verbose = 1,
                          restore_best_weights=True)
#CVS Logger Callback
filename = 'traindataepoch.csv' #file which be saving each epoch data on my local pc
csv_logger = CSVLogger(filename, separator = ",", append = True)

#ReduceLRPlateau Callback
reduce_lr = ReduceLROnPlateau(monitor = 'val_loss',
                              factor = 0.1,   #value by which learning rate will be reduced
(new_lr = lr*factor)
                              patience = 3,   #epochs after which the learning rate would r
educe is same learning rate is being followed
                              min_lr = 0.001) #lower bound on the learning rate

#ModelCheckPoint Callback
filepath = 'weights.{epoch:02d}-{val_loss:.2f}.hdf5'
checkpoint = ModelCheckpoint(filepath, monitor = 'val_loss',
                             mode = 'min',   #saves the minimised monitered value
                             save_best_only=True,
                             verbose = 1)
```

Determining the **batch_size** our our training and testing sets. The batch_size determines the number of samples of images which will be trained in each mini batch. Our batch size is 32 which means out of the total samples of our training 26,217, there are 32 small subset sizes which are used for training the model while learning. Each batch trains network in a successive order, taking into account the updated weights coming from the appliance of the previous batch.

Additionally, calculating the **steps_per_epoch** which is the number of batch iterations which will be trained before a training epoch is considered finished. Our steps per epoch is 819 for training set which implies there will be 819 images passed through layers and trained in our model in each epoch.

In [11]:

```
steps_per_epoch = train_set.n // train_set.batch_size #double slash division returns round
ed int
print(f'The train set steps per epoch: {steps_per_epoch}')
print(f'The train set has: {train_set.n}')
print(f'Train Set batch has: {train_set.batch_size} elements')
print("----------------------------------------")
validation_steps = test_set.n //test_set.batch_size
print(f'The test set steps per epoch: {validation_steps}')
print(f'The test set has: {test_set.n}')
print(f'Test Set batch has:  {test_set.batch_size} elements')
```

```
The train set steps per epoch: 819
The train set has: 26217
Train Set batch has: 32 elements
----------------------------------------
The test set steps per epoch: 162
The test set has: 5212
Test Set batch has:  32 elements
```

4- Implementing pretrained model VGG16 on our dataset

After conducting research on the usage of CNN's in image recognition, I came across some pretrained models one of which is Vgg16 which is known for producing great results on datasets. The VGG architecture consists of blocks, where each block is composed of 2D Convolution and Max Pooling layers. Therefore, to test if building our product on vgg16 would give us a more accurate result, we assigned our input shape with our default image scaled as follows. One parameter to understand here, is the weights is set to 'imagenet' which is the original project which was used to build this architecture, therefore the same weights are added by default.

The original idea was to investigate if pretrained models have the ability to perform better than the custom made CNN models.

In [12]:

```
#Instantiating a VGG16 model and assigning it to variable base_model.
base_model = VGG16(input_shape = (48, 48, 3), # Shape of our images
                   include_top = False, # Only having the first half of the model because
 we will be defining the last fully layer accordingly.
                   weights = 'imagenet') #Setting weights as they were assigned in the ori
ginal architecture
```

In [13]:

```
#Taking all the layers in 'base model', and assigning them as False for trainable because
 we want to train our model with the last layers according to our scenario
for layer in base_model.layers:
    layer.trainable = False
```

In [14]:

```
#Following the default values of implementing vgg16
# Flattening the output layer to 1 dimension for them to be taken into hidden layers
x = layers.Flatten()(base_model.output)

# Adding a fully connected layer with 512 hidden units and ReLU activation
x = layers.Dense(512, activation='relu')(x)

# Add a dropout rate of 0.5
x = layers.Dropout(0.5)(x)

# Assigning 4 as our number of output classes with activation softmax for the last layer
x = layers.Dense(4, activation='softmax')(x)

# Lastly, assining the fully made model to modelvgg containing the layers and parameters a
ccording to our dataset
modelvgg = keras.models.Model(base_model.input, x)

# Compiling the vgg16 model with adam optimizer and Experimenting the default loss functio
n.
modelvgg.compile(optimizer = 'adam', loss = 'binary_crossentropy',metrics = ['accuracy'])
```

Fitting the model and running it on our training set with validation set as our test_set with steps_per_epoch as calculated above with specified callback functions for 20 epochs

In [15]:

```
vgghist = modelvgg.fit(train_set,
                       validation_data = test_set,
                       steps_per_epoch = steps_per_epoch,
                       epochs = 20,
                       callbacks = [earlystop, csv_logger,reduce_lr,checkpoint])
```

```
Epoch 1/20
819/819 [==============================] - 205s 248ms/step - loss: 0.5533 - a
ccuracy: 0.3721 - val_loss: 0.5159 - val_accuracy: 0.4332

Epoch 00001: val_loss improved from inf to 0.51594, saving model to weights.0
1-0.52.hdf5
Epoch 2/20
819/819 [==============================] - 195s 238ms/step - loss: 0.5260 - a
ccuracy: 0.4254 - val_loss: 0.5039 - val_accuracy: 0.4555

Epoch 00002: val_loss improved from 0.51594 to 0.50391, saving model to weigh
ts.02-0.50.hdf5
Epoch 3/20
819/819 [==============================] - 200s 244ms/step - loss: 0.5237 - a
ccuracy: 0.4268 - val_loss: 0.5003 - val_accuracy: 0.4628

Epoch 00003: val_loss improved from 0.50391 to 0.50035, saving model to weigh
ts.03-0.50.hdf5
Epoch 4/20
819/819 [==============================] - 160s 196ms/step - loss: 0.5182 - a
ccuracy: 0.4360 - val_loss: 0.4939 - val_accuracy: 0.4758

Epoch 00004: val_loss improved from 0.50035 to 0.49393, saving model to weigh
ts.04-0.49.hdf5
Epoch 5/20
819/819 [==============================] - 176s 215ms/step - loss: 0.5162 - a
ccuracy: 0.4364 - val_loss: 0.4916 - val_accuracy: 0.4829

Epoch 00005: val_loss improved from 0.49393 to 0.49160, saving model to weigh
ts.05-0.49.hdf5
Epoch 6/20
819/819 [==============================] - 168s 205ms/step - loss: 0.5147 - a
ccuracy: 0.4428 - val_loss: 0.4892 - val_accuracy: 0.4818

Epoch 00006: val_loss improved from 0.49160 to 0.48924, saving model to weigh
ts.06-0.49.hdf5
Epoch 7/20
819/819 [==============================] - 168s 206ms/step - loss: 0.5113 - a
ccuracy: 0.4554 - val_loss: 0.4915 - val_accuracy: 0.4799

Epoch 00007: val_loss did not improve from 0.48924
Epoch 8/20
819/819 [==============================] - 164s 200ms/step - loss: 0.5096 - a
ccuracy: 0.4516 - val_loss: 0.4891 - val_accuracy: 0.4818

Epoch 00008: val_loss improved from 0.48924 to 0.48909, saving model to weigh
ts.08-0.49.hdf5
Epoch 9/20
819/819 [==============================] - 166s 203ms/step - loss: 0.5099 - a
ccuracy: 0.4500 - val_loss: 0.4885 - val_accuracy: 0.4812

Epoch 00009: val_loss improved from 0.48909 to 0.48852, saving model to weigh
ts.09-0.49.hdf5
Epoch 10/20
819/819 [==============================] - 175s 213ms/step - loss: 0.5108 - a
ccuracy: 0.4450 - val_loss: 0.4814 - val_accuracy: 0.4923
```

```
Epoch 00010: val_loss improved from 0.48852 to 0.48144, saving model to weigh
ts.10-0.48.hdf5
Epoch 11/20
819/819 [==============================] - 146s 178ms/step - loss: 0.5052 - a
ccuracy: 0.4615 - val_loss: 0.4870 - val_accuracy: 0.4789

Epoch 00011: val_loss did not improve from 0.48144
Epoch 12/20
819/819 [==============================] - 167s 204ms/step - loss: 0.5048 - a
ccuracy: 0.4602 - val_loss: 0.4814 - val_accuracy: 0.4873

Epoch 00012: val_loss improved from 0.48144 to 0.48136, saving model to weigh
ts.12-0.48.hdf5
Epoch 13/20
819/819 [==============================] - 165s 201ms/step - loss: 0.5064 - a
ccuracy: 0.4550 - val_loss: 0.4879 - val_accuracy: 0.4779

Epoch 00013: val_loss did not improve from 0.48136
Epoch 14/20
819/819 [==============================] - 169s 206ms/step - loss: 0.5035 - a
ccuracy: 0.4613 - val_loss: 0.4814 - val_accuracy: 0.4977

Epoch 00014: val_loss did not improve from 0.48136
Epoch 15/20
819/819 [==============================] - 158s 193ms/step - loss: 0.5012 - a
ccuracy: 0.4622 - val_loss: 0.4805 - val_accuracy: 0.4860

Epoch 00015: val_loss improved from 0.48136 to 0.48054, saving model to weigh
ts.15-0.48.hdf5
Epoch 16/20
819/819 [==============================] - 163s 199ms/step - loss: 0.5018 - a
ccuracy: 0.4609 - val_loss: 0.4847 - val_accuracy: 0.4841

Epoch 00016: val_loss did not improve from 0.48054
Epoch 17/20
819/819 [==============================] - 159s 194ms/step - loss: 0.4999 - a
ccuracy: 0.4697 - val_loss: 0.4815 - val_accuracy: 0.4964

Epoch 00017: val_loss did not improve from 0.48054
Epoch 18/20
819/819 [==============================] - 142s 174ms/step - loss: 0.5022 - a
ccuracy: 0.4593 - val_loss: 0.4781 - val_accuracy: 0.4988

Epoch 00018: val_loss improved from 0.48054 to 0.47808, saving model to weigh
ts.18-0.48.hdf5
Epoch 19/20
819/819 [==============================] - 132s 161ms/step - loss: 0.5002 - a
ccuracy: 0.4632 - val_loss: 0.4763 - val_accuracy: 0.5075

Epoch 00019: val_loss improved from 0.47808 to 0.47630, saving model to weigh
ts.19-0.48.hdf5
Epoch 20/20
819/819 [==============================] - 128s 157ms/step - loss: 0.4989 - a
ccuracy: 0.4690 - val_loss: 0.4748 - val_accuracy: 0.5004

Epoch 00020: val_loss improved from 0.47630 to 0.47485, saving model to weigh
ts.20-0.47.hdf5
```

From the above evaluation, the model ran for all 20 epochs which indicates there was no need for *EarlyStop* to be inabled hence the learning rate and our monitored value *val_loss*, the loss value for the test set decreased efficiently. As we can also, see the model got saved with the values in an hd5f format file on my computer which I can use for future evaluation.

5- Evaluation of model VGG16

Calculating the accuracy of training and testing set accordingly below.

In [126]:

```
train_loss, train_acc = modelvgg.evaluate(train_set)
test_loss, test_acc = modelvgg.evaluate(test_set)
print("Training set Accuracy: {:.2f}" .format(train_acc))
print("Test set Accuracy: {:.2f}" .format(test_acc))
```

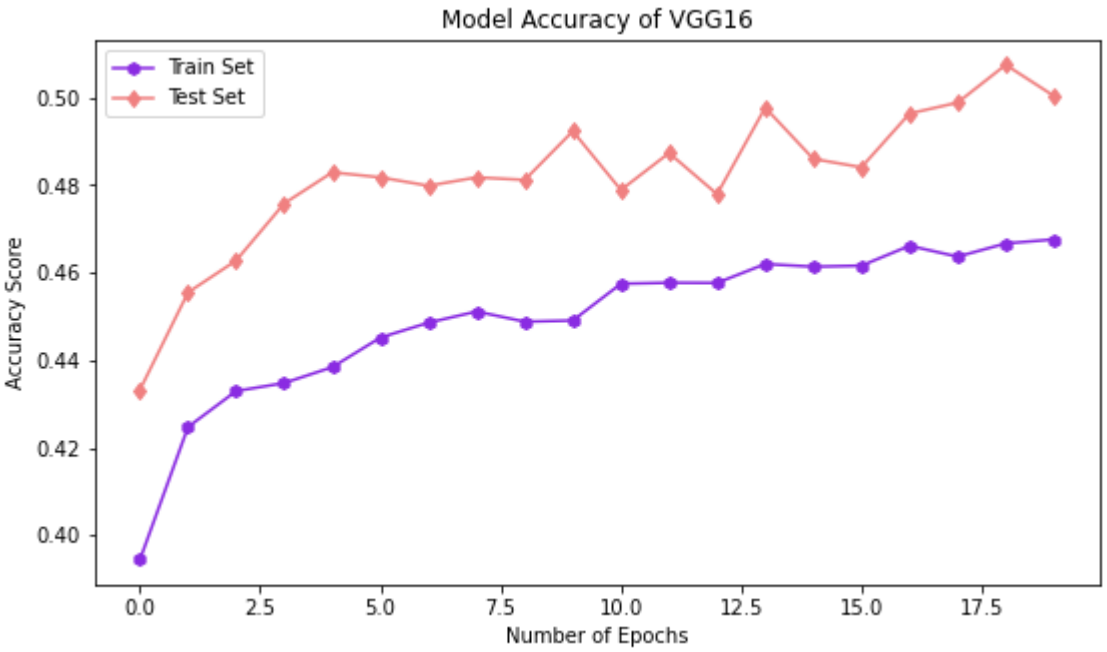
```
820/820 [==============================] - 110s 134ms/step - loss: 0.4899 - a
ccuracy: 0.4824
163/163 [==============================] - 21s 129ms/step - loss: 0.4748 - ac
curacy: 0.5004
Training set Accuracy: 0.48
Test set Accuracy: 0.50
```

As calculated above, we can see vgg16 didn't perform so well on our model. It generated an accuracy of 50% on our test set and with a loss value of 0.47. This can be due to the fact that the resolution of our images are grayscale with 48x48 pixel values with a total of 31,429 data points. While vgg16 is known to be performing quite well when the data points are majorily in millions and with high resolution values.Since, the model didn't perform well therefore we will be going foreword with evaluating the custom made cnn's performance.

Plotting Loss and Accuracy Vs Epochs Plot for VGG16 model

In [36]:

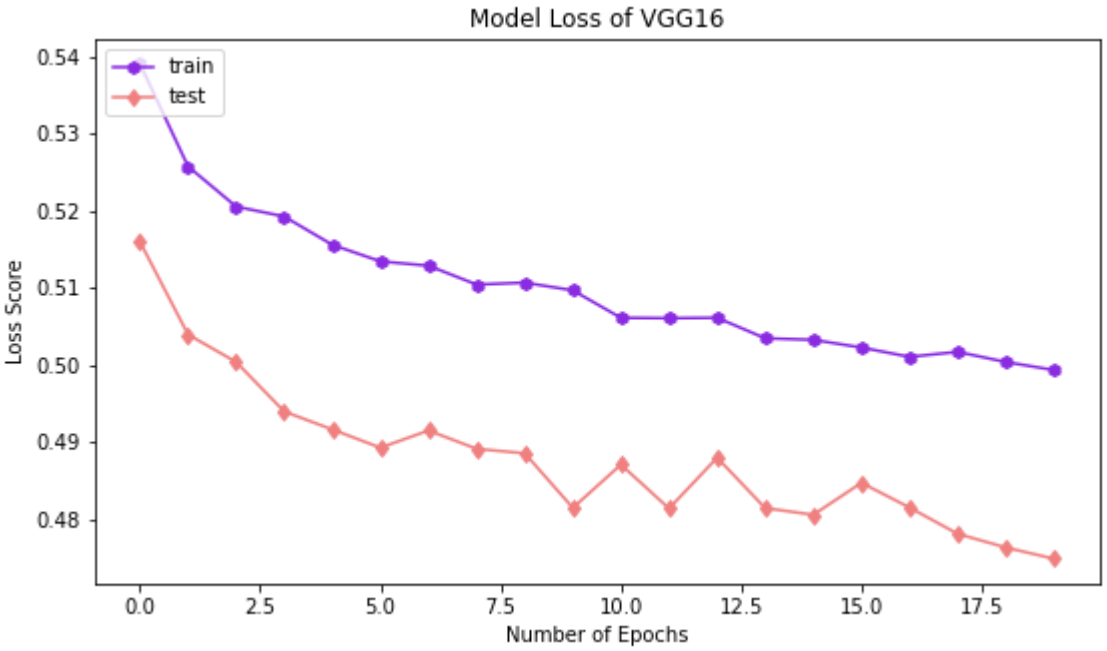
```
#Model Accuracy
plt.figure(figsize=(9,5))
plt.plot(vgghist.history['accuracy'],color = 'blueviolet',marker = "h", label = "Train Se
t")
plt.plot(vgghist.history['val_accuracy'],color = 'lightcoral', marker = "d", label = "Test
Set")

plt.title('Model Accuracy of VGG16')
plt.xlabel('Number of Epochs')
plt.ylabel('Accuracy Score')
plt.legend(loc='upper left')
plt.show()
```



The above graph shows the variance of our training and testing accuracy in each epoch. The purple line representing the training set's accuracy rate tells us that the learning rate was okay but could be improved additionally, the test set's accuracy fluctuated and only reached up to above 50% accuracy magnitude. To make the performance much more smooth, training for a longer period of time with some tweaking of parameters could help.

In [37]:

```
#Model Loss
plt.figure(figsize=(9,5))
plt.plot(vgghist.history['loss'],color = 'blueviolet', marker = "h")
plt.plot(vgghist.history['val_loss'],color = 'lightcoral', marker = "d")

plt.title('Model Loss of VGG16')
plt.xlabel('Number of Epochs')
plt.ylabel('Loss Score')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



The above plot shows that the learning rate of training set is gradually decreasing with a slight decrease which indicates a good learning rate while the difference in distance is a bit further away which can mean that the possibility that this model underfitted can be potential. To improve this, training for longer epochs with magnifying the dropout layer or due to the result of the large number of convolutional and deep layers present in the architecture of VGG16 .

Confusion Matrix of VGG16

For evaluating how the model performed on our test set, we first assign our y_predvgg variable which stores all the predicted emotions of our images in our test set. The result we get is in an array storing the indices of each emotion.

In [28]:

```
y_predvgg = modelvgg.predict(test_set)
y_predvgg = np.argmax(y_predvgg, axis=1) #Assigning, axis = 1, to get the highest common v
alues and for classification metrics to handle a mix of multiclass and continuous-multiout
put targets
y_predvgg
```

Out[28]:

```
array([2, 2, 1, ..., 1, 2, 1], dtype=int64)
```

Next, we will be storing the different emotions classes i.e. [0:'Angry', 1: 'Happy',2: 'Relaxed', 3: 'Sad'] as intergered indices in variable **true_classes** and moreover creating a list of them in with their alphabetic names in variable **class_labels**

In [29]:

```
true_classes = test_set.classes
class_labels = list(test_set.class_indices.keys())
class_labels
```

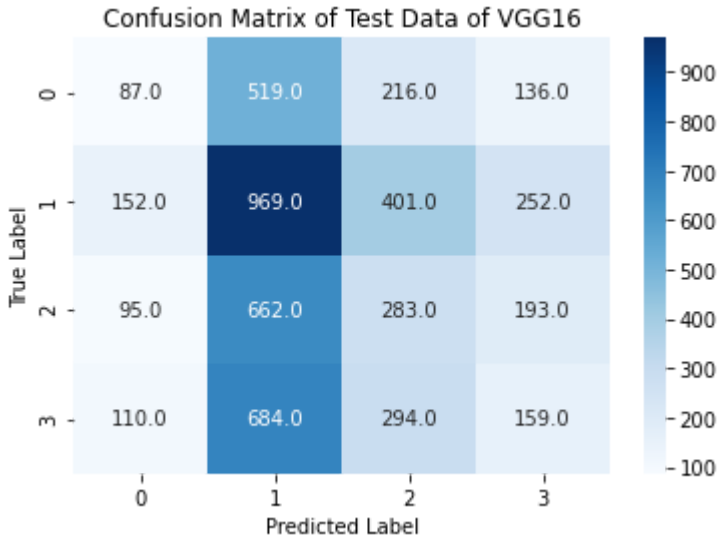
Out[29]:

```
['Angry', 'Happy', 'Relaxed', 'Sad']
```

Plotting the confusion matrix of the VGG16 model for testing

In [38]:

```
test_cm = sklearn.metrics.confusion_matrix(true_classes, y_predvgg)
sns.heatmap(test_cm, annot = True, cmap = plt.cm.Blues,fmt = ".1f" ) #Using fmt as .1f to
 turn the hexadecimal values to exact numerical outputs

plt.title("Confusion Matrix of Test Data of VGG16")
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```



From the confusion matrix plotted above, it can be agreed that the pretrained model learned the emotion Happy in with classifying 969 images correctly. But for predicting emotions Angry,Sad and Relaxed it got confused was wasn't able to predict their true labels. It's also worthwhile to notice that about 684 images which were 3 i.e. Sad were classified as Happy. Therefore, it can be concluded that the VGG16 model predicted only the Happy emotion correctly out of the four emotions and accumulated or atleast recognized the rest emotions as Happy with lower probability.

Plotting the classfication report of the VGG16 model for testing

In [31]:

```
clf_test = classification_report(test_set.classes, y_predvgg, target_names = class_labels)
print(clf_test)
```

```
              precision    recall  f1-score   support

       Angry       0.20      0.09      0.12       958
       Happy       0.34      0.55      0.42      1774
     Relaxed       0.24      0.23      0.23      1233
         Sad       0.21      0.13      0.16      1247

    accuracy                           0.29      5212
   macro avg       0.25      0.25      0.23      5212
weighted avg       0.26      0.29      0.26      5212
```

For evaluating the model more precisely, from the classificiation report made above, focusing on the f1-score which is also known as the harmonic mean as it combines both the values of precision and recall. From the four labels, Happy has comparatively higher f1-score with 0.42 which can be agreed upon as even in the confusion matrix, Happy images were predicted correctly as a person wearing a smile. One reason for this may be due to the reason of this class having higher support value than the rest. While the f1-score for the remaining three remained somewhat in the same range which can be imporved upon for increasing the performance of this model.

Hence, from the evaluation of the VGG16 pretrained model, it can be confirmed that every dataset requires different parameters and structure for making it's performance high, by using pretrained models which have been trained for a long period of time on larger and on one dataset, the result would vary significantly.

6- Implementing custom CNN model on our dataset

After having evaluted the VGG16 model, fitting CNN model with the relevant set parameters and assigning it to **cnn_model** variable. We will be running our model with 20 epochs and the callback functions that we defined above will be getting executed correspondingly.

Additionally, setting the validation data to our test set for validating the performance, here validation steps is similar to steps_per_epoch but holds the images of test set.

In [127]:

```
cnn_model = model.fit(train_set,
                      steps_per_epoch = steps_per_epoch,
                      epochs = 20,
                      validation_data=test_set,
                      validation_steps=validation_steps,
                      callbacks = [earlystop, csv_logger,checkpoint,reduce_lr])
```

```
Epoch 1/20
819/819 [==============================] - 351s 427ms/step - loss: 1.4115 - a
ccuracy: 0.3327 - val_loss: 1.4469 - val_accuracy: 0.3432

Epoch 00001: val_loss did not improve from 0.47485
Epoch 2/20
819/819 [==============================] - 351s 428ms/step - loss: 1.2381 - a
ccuracy: 0.4392 - val_loss: 1.7268 - val_accuracy: 0.3873

Epoch 00002: val_loss did not improve from 0.47485
Epoch 3/20
819/819 [==============================] - 348s 424ms/step - loss: 1.1096 - a
ccuracy: 0.5174 - val_loss: 1.1498 - val_accuracy: 0.5174

Epoch 00003: val_loss did not improve from 0.47485
Epoch 4/20
819/819 [==============================] - 345s 422ms/step - loss: 1.0270 - a
ccuracy: 0.5602 - val_loss: 1.0629 - val_accuracy: 0.5363

Epoch 00004: val_loss did not improve from 0.47485
Epoch 5/20
819/819 [==============================] - 345s 421ms/step - loss: 0.9782 - a
ccuracy: 0.5853 - val_loss: 0.8747 - val_accuracy: 0.6277

Epoch 00005: val_loss did not improve from 0.47485
Epoch 6/20
819/819 [==============================] - 346s 423ms/step - loss: 0.9257 - a
ccuracy: 0.6070 - val_loss: 0.9219 - val_accuracy: 0.6011

Epoch 00006: val_loss did not improve from 0.47485
Epoch 7/20
819/819 [==============================] - 350s 428ms/step - loss: 0.9124 - a
ccuracy: 0.6166 - val_loss: 1.0201 - val_accuracy: 0.5700

Epoch 00007: val_loss did not improve from 0.47485
Epoch 8/20
819/819 [==============================] - 351s 429ms/step - loss: 0.8869 - a
ccuracy: 0.6314 - val_loss: 0.8774 - val_accuracy: 0.6350

Epoch 00008: val_loss did not improve from 0.47485
Epoch 9/20
819/819 [==============================] - 343s 419ms/step - loss: 0.8761 - a
ccuracy: 0.6306 - val_loss: 1.0041 - val_accuracy: 0.5698
Restoring model weights from the end of the best epoch.

Epoch 00009: val_loss did not improve from 0.47485
Epoch 00009: early stopping
```

From the above chunk, due to the **Early Stopping** callback function applied, our model seems to have stopped at epoch 9 which applies that the difference in our monitored value *val_loss* was less than 0 so to avoid overfitting, the model immediately stopped. To check this, we can see the magnitude of val_loss didn't improve further from 0.47 for a continous 4 epochs. Additionally, all the data from each epoch is saved in a CSV file and the model is saved with keeping the best result out of all performed epochs. Few things to notice here, are the the *loss*, *accuracy*,*val_loss*, and *val_Acc*. The *loss* and *accuracy* are for our training set while *val_loss*, and *val_Acc* are for our test set.

7- CNN Model Results

Understanding the percentage of accuracy and loss for both training and testing sets, using the .evaluate().

In [128]:

```
train_loss, train_acc = model.evaluate(train_set)
test_loss, test_acc = model.evaluate(test_set)
print("Training set Accuracy: {:.2f}" .format(train_acc))
print("Test set Accuracy: {:.2f}" .format(test_acc))
```

```
820/820 [==============================] - 65s 80ms/step - loss: 0.9504 - acc
uracy: 0.5861
163/163 [==============================] - 13s 78ms/step - loss: 0.8745 - acc
uracy: 0.6278
Training set Accuracy: 0.59
Test set Accuracy: 0.63
```

From the above chunk, we can confirm our own custom made cnn has achieved an accuracy of 59% on training set and 63% on test set. It performed much better than fitting vgg16, so now I have some idea of pros and cons of when to apply and not apply pretrained models. The test accuracy performed quite well, as when comparing the results of different papers on the same dataset, the accuracy achieved on custom cnn model varies between 60% - 68% while running on 100 epochs. The paper used for referencing is mentioned in more depth in the results documents attachted. Thus, we can conclude when comparing with the results on custom made cnns, our model did quite okay but does hold potential to perform much better.

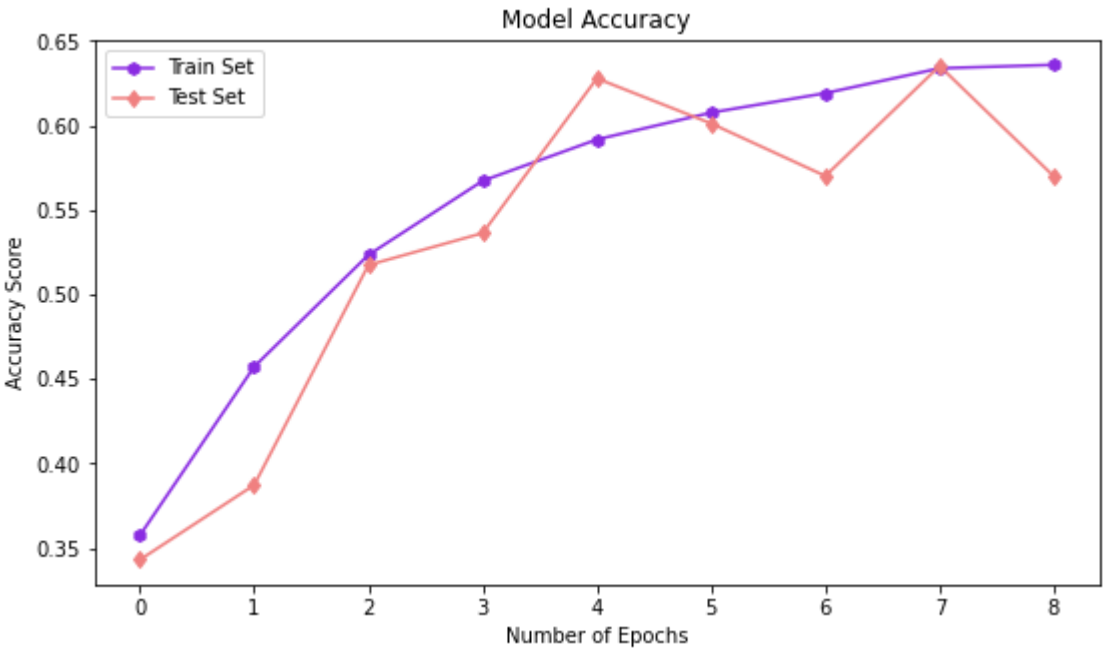
One interesting thing to notice here is that the training accuracy of both the VGG16 and CNN model came out to be slighty higher than the testing accuracy of both models. This can be due to two potention reasons, one being the models' architecture has a high dropout percentage to which I reduced the dropout to 10% from 25% initially to see if it was the main culprit. It didn't do the reverse but lowered down the difference between the two. Secondly, if the data is split in an uneven ratio but as calculated above our images are split into 80% training and 20% testing, hence this couldn't be the reason. Therefore, for future implementation, training on more number of epochs holds potential to normalize and achieve the ideal situation of test and train accuracy.

8- CNN Model Evaluation

For evaluating how well, our model did we will be plotting the graphs of accuracy and loss over the number of epochs our model is trained on.

In [129]:

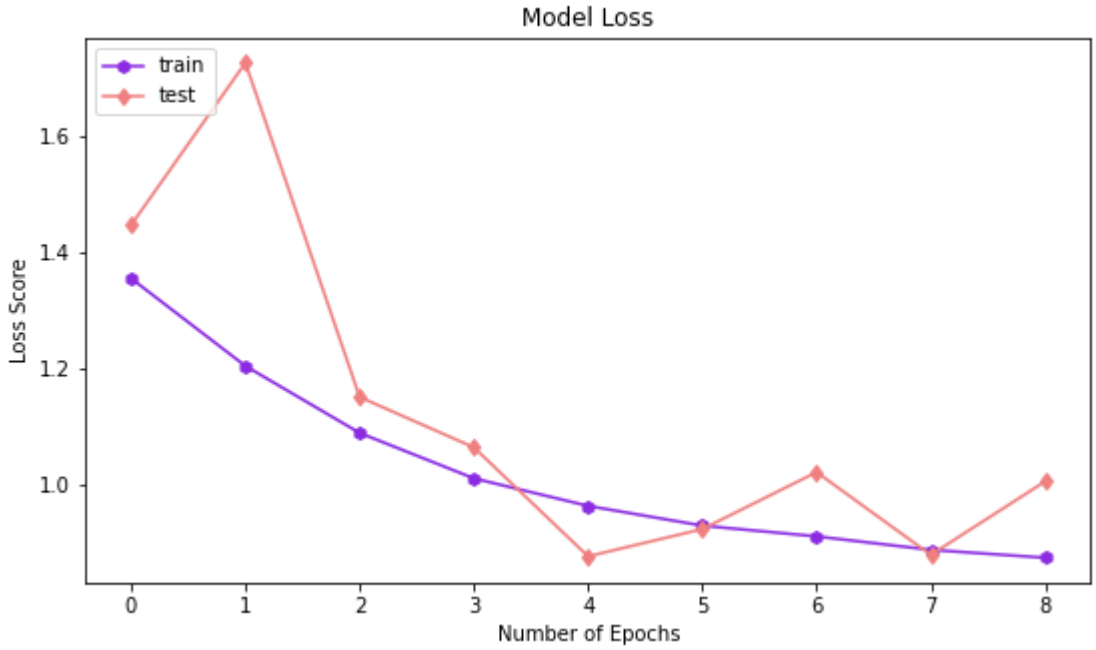
```
#Model Accuracy
plt.figure(figsize=(9,5))
plt.plot(cnn_model.history['accuracy'],color = 'blueviolet',marker = "h", label = "Train Set")
plt.plot(cnn_model.history['val_accuracy'],color = 'lightcoral', marker = "d", label = "Test Set")

plt.title('Model Accuracy')
plt.xlabel('Number of Epochs')
plt.ylabel('Accuracy Score')
plt.legend(loc='upper left')
plt.show()
```



The above graph shows the variance of our training and testing accuracy in each epoch. The purple line representing the training set seems to have plotted perfectly with the ideal training accuracy. The slight gradual increase indicates, our model has a good learning rate. The lightcoral line representing test set's accuracy, experienced great increase since epoch 0. Although, it seems to have it's own trend. Its worth noticing here that at epoch 7, the accuracy of both test and train are equal, this tells us that the model is trying to correspond with the training accuracy and learn from the images of the training set to provide similar results.

In [130]:

```
#Model Loss
plt.figure(figsize=(9,5))
plt.plot(cnn_model.history['loss'],color = 'blueviolet', marker = "h")
plt.plot(cnn_model.history['val_loss'],color = 'lightcoral', marker = "d")

plt.title('Model Loss')
plt.xlabel('Number of Epochs')
plt.ylabel('Loss Score')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```



The above graph represents the loss values of both test and train set. The purple line which is the loss value for our training set has been plotted with a slight curve which implements good learning rate as it is decreasing with a linear value. The lightcoral line representing our loss value for test set, seems to have been flunctuating but from epoch 1, we can see the model is tried to reduce the loss value with attaining values quite or somewhat closer to that of training set. Since, the difference in lines isn't much, therefore we could infer saying our model isn't overfitted nor underfitted but by tweaking and training with big data for a longer period of time with more epochs, the loss value can become gradually low.

9- Test Set

For evaluating how the model performed on our test set, we first assign our y_pred variable which stores all the predicted emotions of our images in our test set. The result we get is in an array storing the indices of each emotion.

In [131]:

```
y_pred = model.predict(test_set)
y_pred = np.argmax(y_pred, axis=1) #Assigning, axis = 1, to get the highest common values
 and for classification metrics to handle a mix of multiclass and continuous-multioutput t
argets
y_pred
```

Out[131]:

```
array([3, 0, 0, ..., 1, 3, 3], dtype=int64)
```

Next, we will be storing the different emotions classes i.e. [0:'Angry', 1: 'Happy',2: 'Relaxed', 3: 'Sad'] as intergered indices in variable **true_classes** and moreover creating a list of them in with their alphabetic names in variable **class_labels**

In [18]:

```
true_classes = test_set.classes
class_labels = list(test_set.class_indices.keys())
class_labels
```

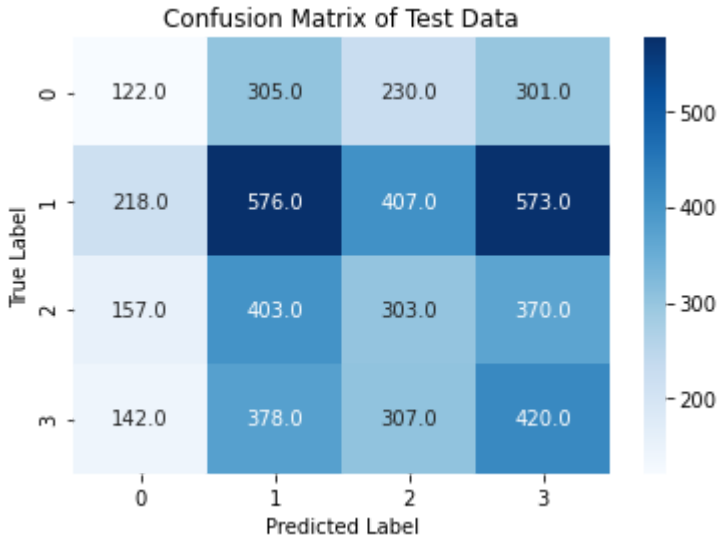
Out[18]:

```
['Angry', 'Happy', 'Relaxed', 'Sad']
```

Plotting the confusion matrix of our CNN model for testing

In [133]:

```
test_cm = sklearn.metrics.confusion_matrix(true_classes, y_pred)
sns.heatmap(test_cm, annot = True, cmap = plt.cm.Blues,fmt = ".1f" ) #Using fmt as .1f to
 turn the hexadecimal values to exact numerical outputs

plt.title("Confusion Matrix of Test Data")
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```



The confusion matrix plotted above with the predicted vs true labels of our emotions (0=Angry, 1 = Happy, 2=Relaxed, 3=Sad), we can determine the model classified the emotion Happy with the highest accuracy. Overall, we can see all of the emotions have been predicted with reference as Happy. Some of the emotions: Sad and Relaxed were quite similar having their true label around 300 instances which in reality can also be difficult to detect. In addition, emotion Angry was poorly detected which can be due the smaller number of instances present in the dataset initially.

Plotting the classfication report of our CNN model for testing

In [134]:

```
clf_test = classification_report(test_set.classes, y_pred, target_names = class_labels)
print(clf_test)
```

```
              precision    recall  f1-score   support

       Angry       0.19      0.13      0.15       958
       Happy       0.35      0.32      0.34      1774
     Relaxed       0.24      0.25      0.24      1233
         Sad       0.25      0.34      0.29      1247

    accuracy                           0.27      5212
   macro avg       0.26      0.26      0.26      5212
weighted avg       0.27      0.27      0.27      5212
```

From the above classification report, we can conclude with putting more stress to the f1-score since it's the harmonic mean of precision and recall, emotion Happy received fairly higher score than the rest which can be due to the fact that the data consists of more images for this emotion and from the confusion matrix, all of the other emotions have been classified with reference as to Happy. The values are pretty low overall considering, the number of samples provided for each class.

10- Predicting Image with Emotion

Now, for testing how well our model classifies and predicts the images of each emotion, we form an iterator of our test_set since, our test_set are images from a directory so by using .next we create *X* and *y* variables where *X* contains the pixel values of each image and *y* consists of the four emotions (0,1,2,3). We further make y as as np array with selecting the unique maximum values of each emotion index.

In [27]:

```
#X has our pixel values and y has our emotion index
X,y = test_set.next()
y = np.argmax(y, axis = 0)
print(len(X), len(y))
```

```
32 4
```

In [28]:

```
#Forming a prediction variable which stores all the predicted emotions of our cnn model
prediction = model.predict(X)
```

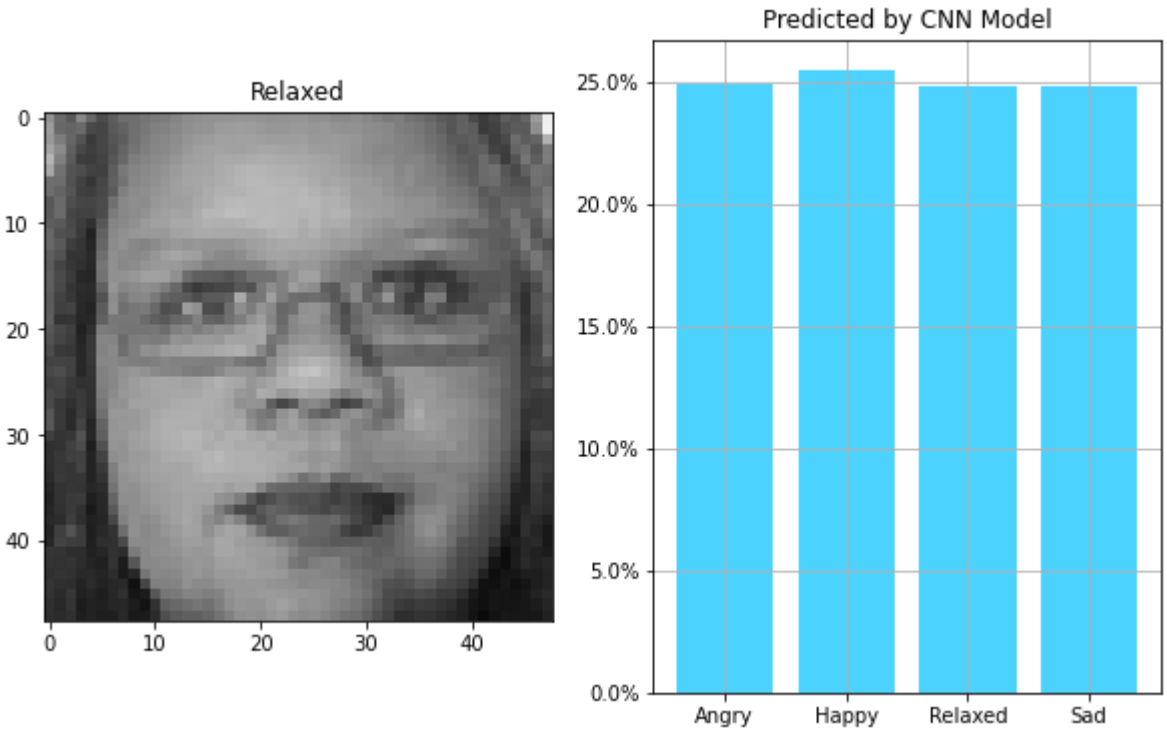
Plotting the Images predicted for each emotion by our model below

In [29]:

```
#Defining a function to display the images classified for each emotion.
def plot_predicted_img(testimg, emotion):
    fig, axis = plt.subplots(1, 2, figsize=(10, 6), sharey=False)
    bar_label = class_labels
    axis[0].imshow(testimg[emotion], 'gray')
    axis[0].set_title(class_labels[emotion])
    axis[1].set_title("Predicted by CNN Model")
    axis[1].bar(bar_label, prediction[emotion], color='deepskyblue', alpha=0.7)
    axis[1].yaxis.set_major_formatter(mtick.PercentFormatter(1.0))
    axis[1].grid()

    plt.show()
```

In [31]:

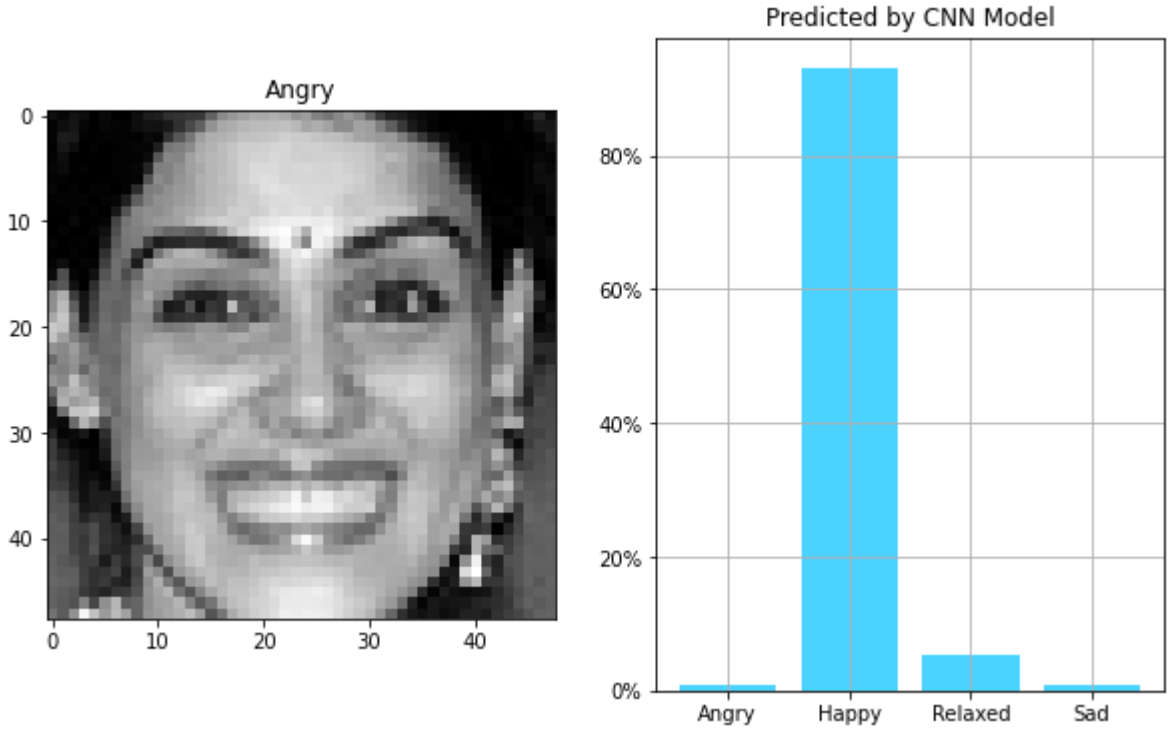
```
#Plotting images classified as emotion 3 i.e. Happy
plot_predicted_img(X,2)
```



From the plot above, we can see there's the image of emotion 'relaxed' from the test set and the model has classified it as Happy with 25% accuracy while, it's interesting to note here, that all of the other emotions are also on the same percentage. This is quite acceptible since for a human eyes, it's sometimes difficult to understand or classify what specific emotion is being depicted.

In [139]:

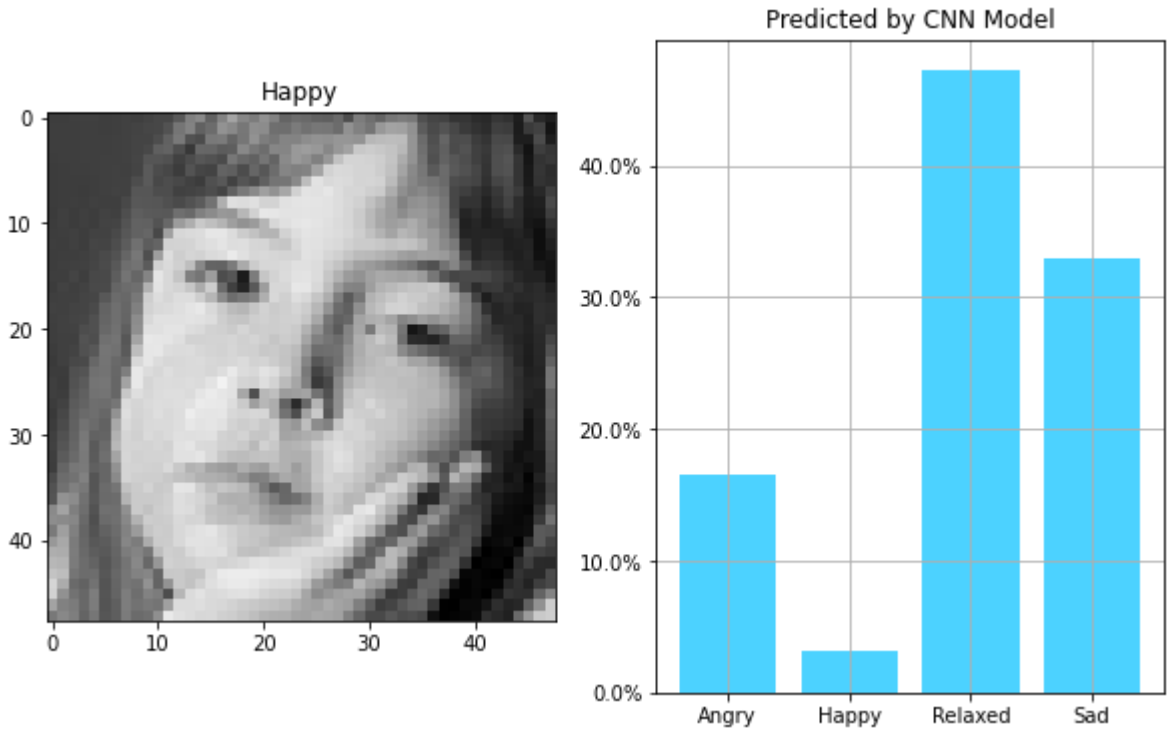
```
#Plotting images classified as emotion 1 i.e. Surprise
plot_predicted_img(X,0)
```



As we can see above, the image was essentially from 'Angry' category but the model classified it as more than 80% as Happy. This trend was seen in our confusion matrix.

In [140]:

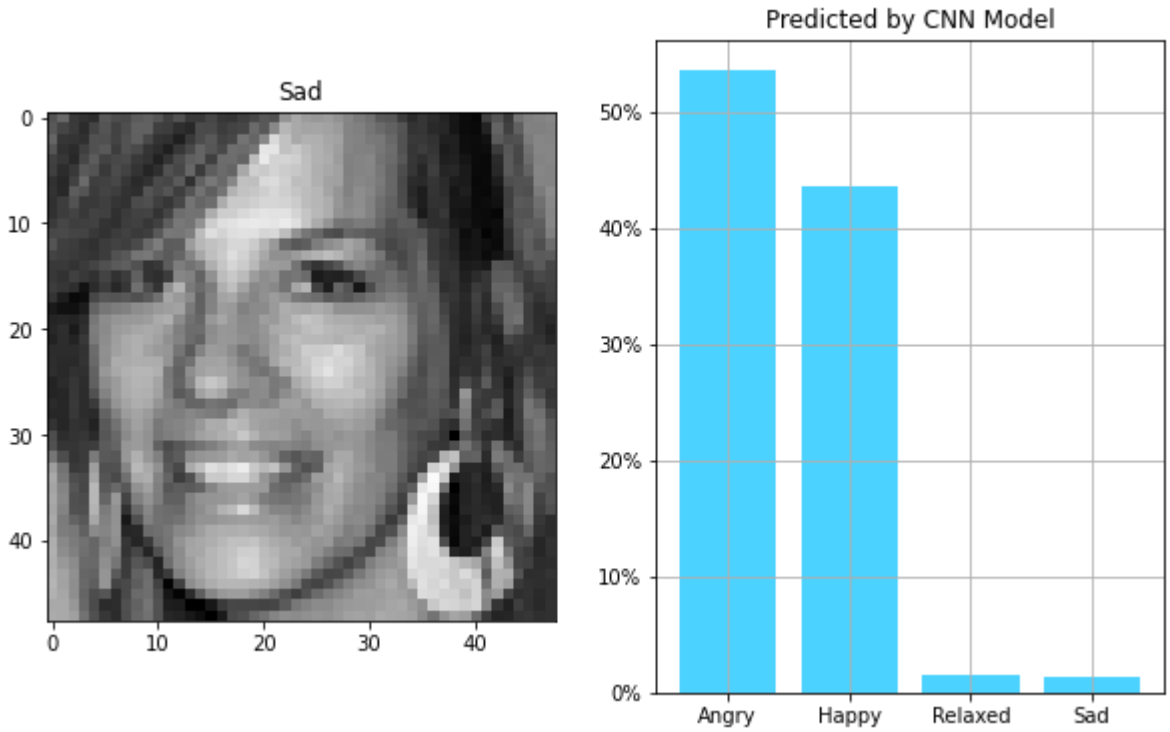
```
#Plotting images classified as emotion 3 i.e. Happy
plot_predicted_img(X,1)
```



The above emotion is 'Happy' from the test set, but the model successfully classified it as 'Relaxed'. This scenario is quite acceptible since, the emotion depicted isn't really illustrating a Happy emotion. So, we can say our model has learnt to classify "Happy" and "Relaxed emotion correctly.

In [141]:

```
#Plotting images classified as emotion 5 i.e. Sad
plot_predicted_img(X,3)
```



From the above plot, we can infer the model predicted this image which was essentially 'Sad' as 'Angry'. The reason for this could be that some of the emotions maybe really difficult to distinguish as the image displays the women smiling but is labeled as feeling sad. We can see our model predicted this image as Happy with a percentage of more than 40%.

Visualizing our CNN Model's Architecture

For visualizing our cnn model's architecture, we used the library **ann_viz** for plotting our netural network which saves our model as a pdf file on our working directory. One interesting factor to notice, here is the ann_viz function is able to display the structure of our model only without BatchNormalization() layer. The file will be uploaded with this notebook separately.

In [26]:

```
#Visualising our model with settint view to True, to open the image after running this pie
ce of chunck every time.
ann_viz(model, view = True, title = 'MyNeuralNetwork')
```

Visualising Feature Maps from our CNN layers

After forming the layers above for our model, I wanted to visualize and understand exactly how the image was being transfered from one neural layer to another our feature maps. To accomplish this, I layed out the number of layers that were in my model and assigned it to **layers_names** variable.

In [21]:

```
#Listing only the names of the intermediate layers
layers_names = [layer.name for layer in model.layers]
layers_names
```

Out[21]:

```
['conv2d',
 'conv2d_1',
 'batch_normalization',
 'activation',
 'max_pooling2d',
 'conv2d_2',
 'conv2d_3',
 'batch_normalization_1',
 'activation_1',
 'max_pooling2d_1',
 'dropout',
 'conv2d_4',
 'conv2d_5',
 'batch_normalization_2',
 'activation_2',
 'max_pooling2d_2',
 'conv2d_6',
 'conv2d_7',
 'batch_normalization_3',
 'activation_3',
 'max_pooling2d_3',
 'dropout_1',
 'flatten',
 'dense',
 'batch_normalization_4',
 'activation_4',
 'dense_1',
 'batch_normalization_5',
 'activation_5',
 'dense_2',
 'activation_6']
```

Now returning the list of output layers or more general, listing the number of layers in our input and output layers.

In [14]:

```
layer_outputs = [layer.output for layer in model.layers]
layer_outputs
```

Out[14]:

```
[<KerasTensor: shape=(None, 48, 48, 64) dtype=float32 (created by layer 'conv
2d')>,
 <KerasTensor: shape=(None, 48, 48, 64) dtype=float32 (created by layer 'conv
2d_1')>,
 <KerasTensor: shape=(None, 48, 48, 64) dtype=float32 (created by layer 'batc
h_normalization')>,
 <KerasTensor: shape=(None, 48, 48, 64) dtype=float32 (created by layer 'acti
vation')>,
 <KerasTensor: shape=(None, 48, 48, 64) dtype=float32 (created by layer 'max_
pooling2d')>,
 <KerasTensor: shape=(None, 48, 48, 128) dtype=float32 (created by layer 'con
v2d_2')>,
 <KerasTensor: shape=(None, 48, 48, 128) dtype=float32 (created by layer 'con
v2d_3')>,
 <KerasTensor: shape=(None, 48, 48, 128) dtype=float32 (created by layer 'bat
ch_normalization_1')>,
 <KerasTensor: shape=(None, 48, 48, 128) dtype=float32 (created by layer 'act
ivation_1')>,
 <KerasTensor: shape=(None, 48, 48, 128) dtype=float32 (created by layer 'max
_pooling2d_1')>,
 <KerasTensor: shape=(None, 48, 48, 128) dtype=float32 (created by layer 'dro
pout')>,
 <KerasTensor: shape=(None, 48, 48, 256) dtype=float32 (created by layer 'con
v2d_4')>,
 <KerasTensor: shape=(None, 48, 48, 256) dtype=float32 (created by layer 'con
v2d_5')>,
 <KerasTensor: shape=(None, 48, 48, 256) dtype=float32 (created by layer 'bat
ch_normalization_2')>,
 <KerasTensor: shape=(None, 48, 48, 256) dtype=float32 (created by layer 'act
ivation_2')>,
 <KerasTensor: shape=(None, 48, 48, 256) dtype=float32 (created by layer 'max
_pooling2d_2')>,
 <KerasTensor: shape=(None, 48, 48, 512) dtype=float32 (created by layer 'con
v2d_6')>,
 <KerasTensor: shape=(None, 48, 48, 512) dtype=float32 (created by layer 'con
v2d_7')>,
 <KerasTensor: shape=(None, 48, 48, 512) dtype=float32 (created by layer 'bat
ch_normalization_3')>,
 <KerasTensor: shape=(None, 48, 48, 512) dtype=float32 (created by layer 'act
ivation_3')>,
 <KerasTensor: shape=(None, 48, 48, 512) dtype=float32 (created by layer 'max
_pooling2d_3')>,
 <KerasTensor: shape=(None, 48, 48, 512) dtype=float32 (created by layer 'dro
pout_1')>,
 <KerasTensor: shape=(None, 1179648) dtype=float32 (created by layer 'flatte
n')>,
 <KerasTensor: shape=(None, 512) dtype=float32 (created by layer 'dense')>,
 <KerasTensor: shape=(None, 512) dtype=float32 (created by layer 'batch_norma
lization_4')>,
 <KerasTensor: shape=(None, 512) dtype=float32 (created by layer 'activation_
4')>,
 <KerasTensor: shape=(None, 256) dtype=float32 (created by layer 'dense_1')>,
 <KerasTensor: shape=(None, 256) dtype=float32 (created by layer 'batch_norma
lization_5')>,
 <KerasTensor: shape=(None, 256) dtype=float32 (created by layer 'activation_
```

```
5')>,
 <KerasTensor: shape=(None, 4) dtype=float32 (created by layer 'dense_2')>,
 <KerasTensor: shape=(None, 4) dtype=float32 (created by layer 'activation_
6')>]
```

From the above list, we can infer that we have a total of 31 outputs, with 23 intermediate outputs and 1 final classification output. Hence, we have 22 feature maps.

In [15]:

```
# Finding the input shape the convolutional layer takes which is 48,48,3
model.input
```

Out[15]:

```
<KerasTensor: shape=(None, 48, 48, 3) dtype=float32 (created by layer 'conv2d
_input')>
```

After we have understood the different layers both input and output, we will be assigning each of them to our *nr_feature_maps* vector which uses the models.Model() function from tensorflow library, and it takes in the two necessary arguments, inputs i.e. the input image that we will be feeding into our model and the outputs.

Therefore, since our *layer_outputs* contains 31 layers, so the image that we will be feeding will go pass through each of the 22 feature maps taking layers.

In [16]:

```
#Any image given to nr_feature_map will be returning us 22 feature maps as calculated abov
e
nr_feature_map = tf.keras.models.Model(inputs=model.input, outputs=layer_outputs)
```

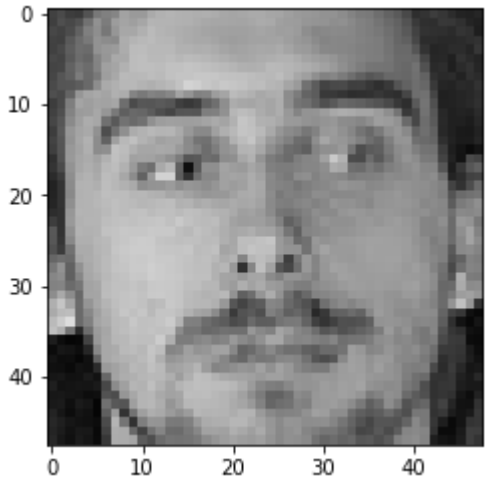
To visualize, we will be importing an image from our own dataset used of Relaxed emotion to understand how the model learnt and feeded it in it's different layers. But essentially, after loading it to our notebook, we first handedly have to convert it to an array and furtherly reshape and normalize it so as to match it with the dimensions of our images accepted in our model.

In [17]:

```
from tensorflow.keras.preprocessing.image import img_to_array, load_img
#Loading and converting the image to an array with the target size of 48x48
img = 'emotion_recognition/test/Relaxed/PrivateTest_69829588.jpg'
img = load_img(img, target_size = (48,48))
new_img = img_to_array(img)
#array to be reshaped 1, and with the shape of our img.shape
new_img = new_img.reshape((1,) + new_img.shape)
#normalizing its pixel values
new_img /= 255.
```

In [18]:

```
#Plotting the original image
img = plt.imshow(img)
```



After loading the image in our directory, we will be feeding it to our model by using predict function on our feature map vector generated above.

In [19]:

```
#Feeding the img to our model and generating feature maps for us
feature_maps = nr_feature_map.predict(new_img)
```

Now that we have our img feeded in our model, we will be **investigating the dimensions of the feature maps** to understand how the dimensions vary and change from one convolutional layer to a dense layer.

In [22]:

```
for layer_name, feature_map in zip(layers_names, feature_maps):
    print(f"The shape of the {layer_name} is =======>> {feature_map.shape}")
```

```
The shape of the conv2d is =======>> (1, 48, 48, 64)
The shape of the conv2d_1 is =======>> (1, 48, 48, 64)
The shape of the batch_normalization is =======>> (1, 48, 48, 64)
The shape of the activation is =======>> (1, 48, 48, 64)
The shape of the max_pooling2d is =======>> (1, 48, 48, 64)
The shape of the conv2d_2 is =======>> (1, 48, 48, 128)
The shape of the conv2d_3 is =======>> (1, 48, 48, 128)
The shape of the batch_normalization_1 is =======>> (1, 48, 48, 128)
The shape of the activation_1 is =======>> (1, 48, 48, 128)
The shape of the max_pooling2d_1 is =======>> (1, 48, 48, 128)
The shape of the dropout is =======>> (1, 48, 48, 128)
The shape of the conv2d_4 is =======>> (1, 48, 48, 256)
The shape of the conv2d_5 is =======>> (1, 48, 48, 256)
The shape of the batch_normalization_2 is =======>> (1, 48, 48, 256)
The shape of the activation_2 is =======>> (1, 48, 48, 256)
The shape of the max_pooling2d_2 is =======>> (1, 48, 48, 256)
The shape of the conv2d_6 is =======>> (1, 48, 48, 512)
The shape of the conv2d_7 is =======>> (1, 48, 48, 512)
The shape of the batch_normalization_3 is =======>> (1, 48, 48, 512)
The shape of the activation_3 is =======>> (1, 48, 48, 512)
The shape of the max_pooling2d_3 is =======>> (1, 48, 48, 512)
The shape of the dropout_1 is =======>> (1, 48, 48, 512)
The shape of the flatten is =======>> (1, 1179648)
The shape of the dense is =======>> (1, 512)
The shape of the batch_normalization_4 is =======>> (1, 512)
The shape of the activation_4 is =======>> (1, 512)
The shape of the dense_1 is =======>> (1, 256)
The shape of the batch_normalization_5 is =======>> (1, 256)
The shape of the activation_5 is =======>> (1, 256)
The shape of the dense_2 is =======>> (1, 4)
The shape of the activation_6 is =======>> (1, 4)
```

As we can see above, the shape for **convolutional layer** is (1,48,48,64) which can be read as the layer having 1 image of shape 48x48 and they have been distributed into 64 images or neurons. So our layer_1 has 64 features which have been clubbed into 1.

For our **batch normalization layer**, we have (1,46,46,64) which is showing us, we have 1 img as our input with size 46x46 and 64 channels/featurs and they have been classified with each of the different parts of the CNN layer.

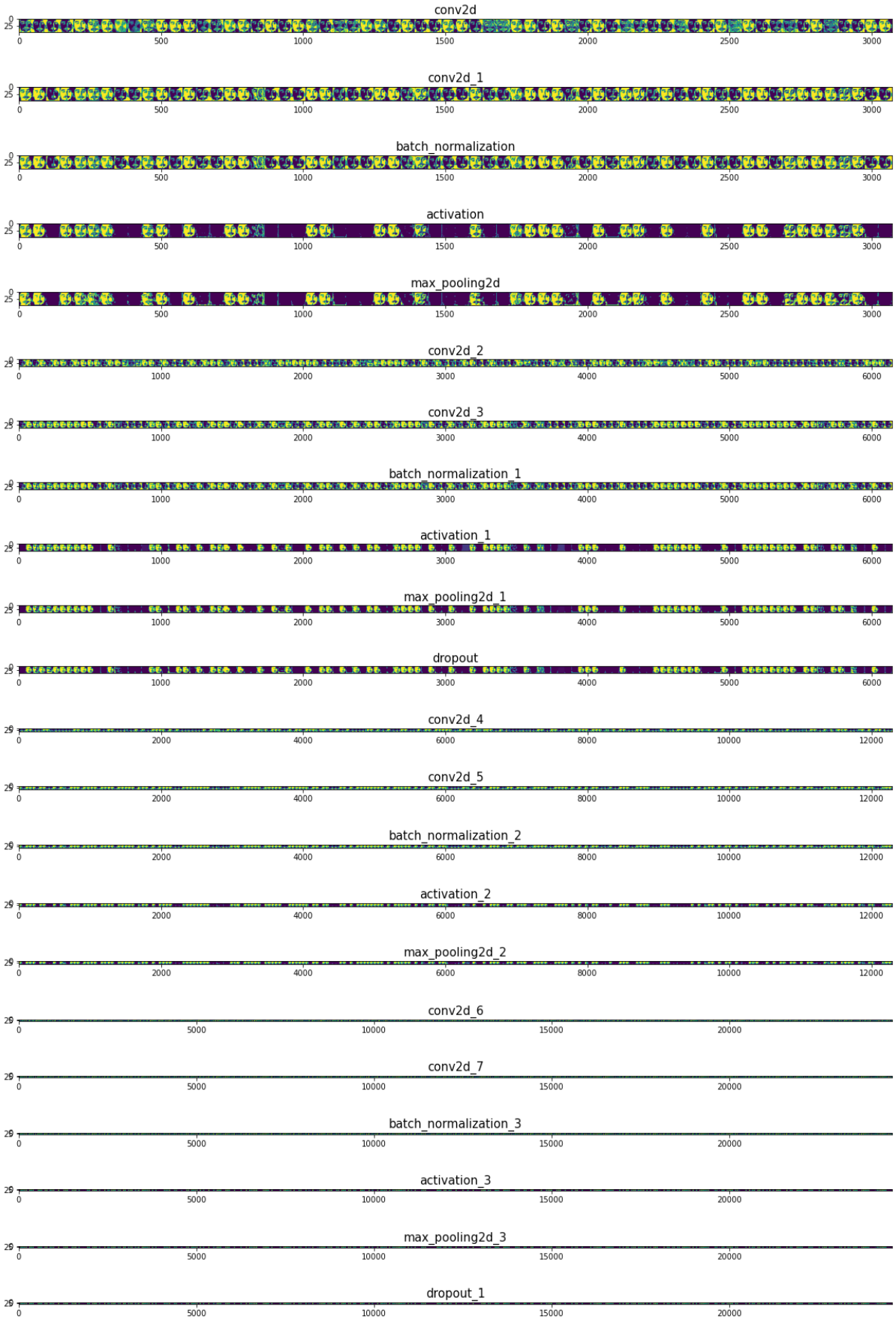
For our **flatten layer**, the layer helps in flattening our image to (1,1179648).

And the shape of our **dense_2 layer** can be read as the model having 1 input and 4 possible outcomes i.e. the emotions in our case.

Finally, now as we have understood the layers and the image has been feeded into our model as input, we can visually see how the model finds edges, textures and other important patterns to classify our image.

In [24]:

```
for layer_name, feature_map in zip(layers_names, feature_maps):
    if len(feature_map.shape) == 4: #since we will be looking only at convolutional layers
and not dense, hence we have our dimension as 4.
        # number of features in an individual feature map(starting from -1 because we want
all individual features maps in each layer)
        n_features = feature_map.shape[-1]
        # The feature map is in shape of (1, size, size, n_features)
        #basically storing the number of filters available in each feature map (48,44,44,4
3,43)
        size = feature_map.shape[1]
        # Fitting the size of our images in each row in our plot we will be plotting i.e.
 display_grid
        display_grid = np.zeros((size, size * n_features))

        # Going over the feature maps of each layer to separate.
        for i in range(n_features):
            #Further standardization and normalization of feature image to make it human e
ye friendly
            imgw = feature_map[0, :, :, i]
            imgw -= imgw.mean()
            #imgw /= imgw.std()
            imgw *= 64
            imgw += 128
            imgw = np.clip(imgw, 0, 255).astype('uint8')
            # We'll tile each filter into this big horizontal grid
            display_grid[:, i * size : (i + 1) * size] = imgw
        # Plotting the layers
        scale = 20. / n_features #20/4 = 5
        #forming a 5 width 20 length with 5 rows.
        plt.figure(figsize=(scale * n_features, scale))
        #plt.figure(figsize=(20 * 2, 2))
        plt.title(layer_name, fontsize = 15)
        plt.imshow(display_grid, aspect='auto', cmap='viridis')
        plt.show()
```


conv2d
conv2d_1
batch_normalization
activation
max_pooling2d
conv2d_2
conv2d_3
batch_normalization_1
activation_1
max_pooling2d_1
dropout
conv2d_4
conv2d_5
batch_normalization_2
activation_2
max_pooling2d_2
conv2d_6
conv2d_7
batch_normalization_3
activation_3
max_pooling2d_3
dropout_1

The above plot generated gives a pleasant visualization of how the image is transferred, twisted and manipulated by different techniques in all the different layers. **Starting from the convolutional layers, the edges, textures and some facial patterns are being selected.** In the batch normalization, the image pixels are getting normalized for each batch. While in drop out layers, some of the neurons being randomly shut to only select the most important features to predicting the emotion of the image provided. As the layers get more populated, their visulization plot becomes really small for the naked eye to see.

Saving the Model

Now that we have formed, evaluated our model, we will be saving our model. The model will be saved with a **HDF5** file named **"my_h5_model.h5"** on my logs folder in my pc and will contain the model's architecture, weights values, and compile() information. It is a light-weight method alternative to SavedModel which is the default keras saving function.

In [ ]:

```
model.save("my_h5_model.h5")
```

Conclusion

To conclude, working with CNN models is interestingly a great challenge as it's carries its own pros and cons. Its definitely a very powerful algorithm developed which helped me in recognizing images and classifying them into four basic emotions which hold the potential to be felt by a patient after undergoing surgery. After rigorous experimenting with the number of layers, optimization function, finding the good learning rate and tweaking necessary parameters, I found the model architecture which resulted in a better performance. I am confident in saying this challenge has been a great experience in understanding the back story of Deep Learning models and how each image is passed on through different layers as seen by visualizing the feature maps. The model constructed definietly holds more potential to perform better by addition of more images with their respective emotions felt as seen while plotting the emotions in the test set vs the emotion predicted by the model. Additionally, training the model for a longer duration of time with more epochs would result in a higher performance rate in real life monitoring as well. The model successfuly classified emotions Happy and Relaxed although all the emotions were classified highly as Happy, this area can be improved for future implementations. Therefore, patients who illustrate negative emotions such as Angry and Sad require higher attention rate than those who are Happy and Relaxed.