FACIAL EMOTION Recognition using TensorFlow

Under the guidance of

Prof. Sudeshna Kundu

&

Prof. Shibaprasad Sen

(Department of Computer Science)

BY PROGECT GROUP – 61

1) Meraj Munshi
2) Aryan Modi
3) Sohini Kanjilal
4) Md Mahmood Alam
5) Abhirup Sen
6) Sayantan Pan

Why Emotion Detection?

- The motivation behind choosing this topic specially lies in the huge investments large corporation do in feedbacks and surveys but fail to get equitable response on their investments.
- Emotion Detection through facial gestures is a technology that aims to improve products and services performance by monitoring customer behavior to certain products or services staff by their evaluation

Some Companies That Make Use Of Emotion Detection……

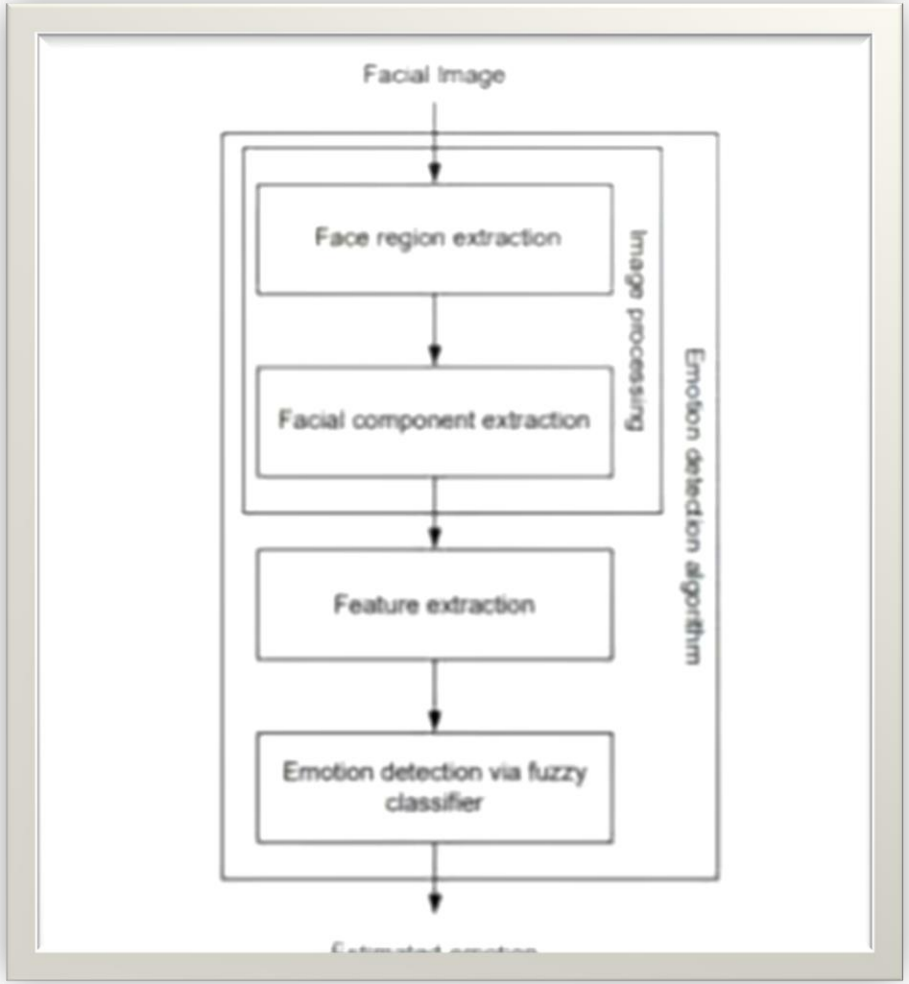
- *While Disney uses emotion-detection tech to find out opinion on a completed products, other brands have used it to directly inform advertising and digital marketing.*
- *Kellogg's is just one high-profile example , having used Affectiva's software to test audience reaction to ads for its cereal.*
- *Unilever does this , using HireVue's AI-powered technology to screen prospective candidates based on factors like body languages and mood . In doing so, the company is able to find the person whose personality and characteristics are best suited to the job.*


FACIAL EXPRESSION RECOGNITION: A DEEP LEARNING APPROACH
Surprise
Surprise

Facial Emotion Recognition by CNN

- Steps:

1. Data Pre-processing
2. Training
3. Validation



Dataset Description

The data consists of 48*48 pixel grayscale images of faces. The faces have been categorized into facial expression in to one of seven categories (0=Angry,1=Disgust,2=Fear,3=Happy,4=Sad,5=Surprise,6=Neural).

The dataset which we'll use is fer2013 which was published in International Conference on Machine Learning in 2013.

Data Preprocessing

- The fer2013.csv consist of three columns namely emotion, pixels and purpose.
- The column in pixel first of all is stored in a list format.
- Since computational complexity is high for computing pixel values in the range of (0-255), the data in pixel field is normalized to values between[0-1].
- The face objects stored are reshaped and resized to the mentioned size of 48*48.
- The respective emotion labels and their respective pixel values are stored in objects.

Let's dive into the project, first opening a new project using Jupyter Notebook . First and foremost, importing the libraries

```
import tensorflow as tf
import cv2
import os
import matplotlib.pyplot as plt
import numpy as np
```

OpenCV-Python is a library of Python bindings designed to solve computer vision problems.cv2.imread() method loads an image from the specified file. If the image cannot be read (because of the missing file, improper permissions, unsupported or invalid format) then this method returns an empty matrix. We can read the image through the following line of code:

```
img_array=cv2.imread("Training/0/Training_3908.jpg")
```

To check the size of the image, we use:

```
img_array.shape
```

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

The matplotlib function imshow() creates an image from a 2-dimensional numpy array. The image will have one square for each element of the array. The color of each square is determined by the value of the corresponding array element and the color map used by imshow().

```
1 plt.imshow(img_array)
```

```
<matplotlib.image.AxesImage at 0x22d4fde5700>
```

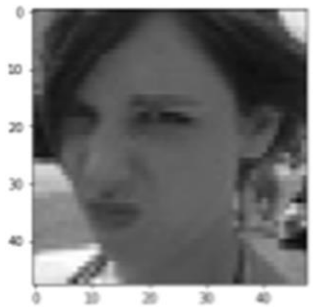
Now we'll create a variable containing the directory name and a list which will contain the names of the folders inside that directory. In our case, we have renamed the folder according to the emotion labels.

```
1 Datadirectory="dataset/Training/"
```

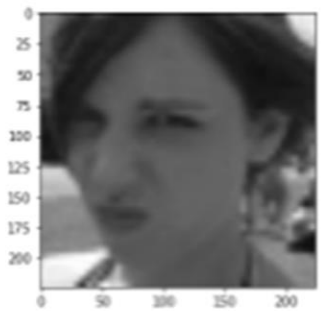
```
1 Classes = ["0","1","2","3","4","5","6"]
```

The ImageNet dataset contains images of fixed size of 224*224 and have RGB channels but as fer2013 has images of size 48*48 so we'll have to resize the images. To resize an image, OpenCV provides cv2.resize() function. cv2.cvtColor() method is used to convert an image from one color space to another.

```
for category in Classes:
    path=os.path.join(Datadirectory,category)
    for img in os.listdir(path):
        img_array=cv2.imread(os.path.join(path,img))
        plt.imshow(cv2.cvtColor(img_array,cv2.COLOR_BGR2RGB))
        plt.show()
        break
    break
```



```
img_size=224
new_array=cv2.resize(img_array, (img_size,img_size))
plt.imshow(cv2.cvtColor(new_array,cv2.COLOR_BGR2RGB))
plt.show()
```



```
new_array.shape
```

```
(224, 224, 3)
```

The reason behind executing the above code is that we're using transfer learning so for transfer learning if we want to use any deep learning classifier then these dimensions must be same. Now we'll read all the images and will convert them to array.

```
training_Data = []
def create_training_Data():
    for category in Classes:
        path=os.path.join(Datadirectory,category)
        class_num=Classes.index(category)
        for img in os.listdir(path):
            try:
                img_array=cv2.imread(os.path.join(path,img))
                new_array=cv2.resize(img_array,(img_size,img_size))
                training_Data.append([new_array,class_num])
            except Exception as e:
                pass
```

Let's call the function:

```
create_training_Data()
```

```
training_Data[0][0].shape
```

```
(224, 224, 3)
```

To make our deep learning architecture dynamic and robust, let's shuffle the sequence:

```
import random
random.shuffle(training_Data)
```

Training

Deep Learning Model For Training - Transfer Learning

Now we'll train our deep learning model using transfer learning

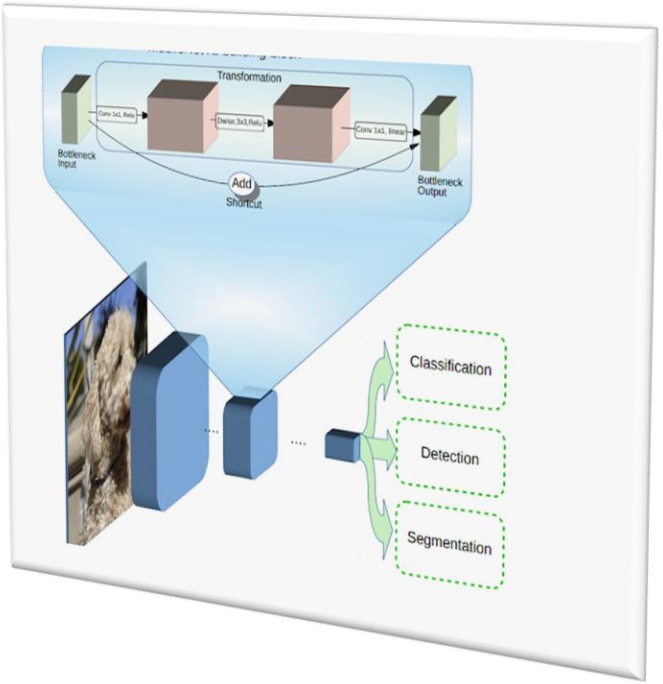
```
import math
from tensorflow import keras
from tensorflow.keras import layers
```

Keras Applications are deep learning models that are made available alongside pre-trained weights. These models can be used for prediction, feature extraction, and fine-tuning. Here is a chart of some available models.

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
ResNet101	171 MB	0.764	0.928	44,707,176	-
ResNet152	232 MB	0.766	0.931	60,419,944	-
ResNet50V2	98 MB	0.760	0.930	25,613,800	-
ResNet101V2	171 MB	0.772	0.938	44,675,560	-
ResNet152V2	232 MB	0.780	0.942	60,380,648	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-
EfficientNetB0	29 MB	-	-	5,330,571	-
EfficientNetB1	31 MB	-	-	7,856,239	-
EfficientNetB2	36 MB	-	-	9,177,569	-

MobileNetV2: The Next Generation of On-Device Computer Vision Networks

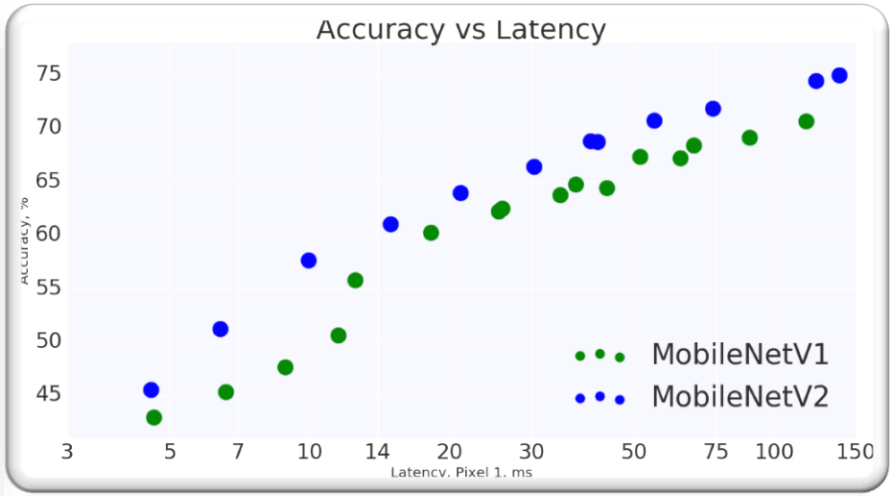
MobileNetV2 is a significant improvement over MobileNetV1 and pushes the state of the art for mobile visual recognition including classification, object detection and semantic segmentation. MobileNetV2 is released as part of TensorFlow-Slim Image Classification Library, or you can start exploring MobileNetV2 right away in Collaboratory. Alternately, you can download the notebook and explore it locally using Jupyter. MobileNetV2 is also available as modules on TF-Hub, and pretrained checkpoints can be found on github.



How does it compare to the first generation of MobileNets?
Overall, the MobileNetV2 models are faster for the same accuracy across the entire latency spectrum. In particular, the new models use 2x fewer operations, need 30% fewer parameters and are about 30-40% faster on a Google Pixel phone than MobileNetV1 models, all while achieving higher accuracy.

```
In [34]: 1 new_model.summary()
```

```
Model: "model_1"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_2 (InputLayer)            [(None, 224, 224, 3) 0
__________________________________________________________________________________________________
Conv1 (Conv2D)                  (None, 112, 112, 32) 864         input_2[0][0]
__________________________________________________________________________________________________
bn_Conv1 (BatchNormalization)   (None, 112, 112, 32) 128         Conv1[0][0]
__________________________________________________________________________________________________
Conv1_relu (ReLU)               (None, 112, 112, 32) 0           bn_Conv1[0][0]
__________________________________________________________________________________________________
expanded_conv_depthwise (Depthw (None, 112, 112, 32) 288         Conv1_relu[0][0]
__________________________________________________________________________________________________
expanded_conv_depthwise_BN (Bat (None, 112, 112, 32) 128         expanded_conv_depthwise[0][0]
__________________________________________________________________________________________________
expanded_conv_depthwise_relu (R (None, 112, 112, 32) 0           expanded_conv_depthwise_BN[0][0]
__________________________________________________________________________________________________
expanded_conv_project (Conv2D)  (None, 112, 112, 16) 512         expanded_conv_depthwise_relu[0][0
```



Now we'll use MobileNetV2

```
model = tf.keras.applications.MobileNetV2()
```

Let's change the base input

```
base_input = model.layers[0].input
```

As we want seven classes, so let's cut down output

```
base_output = model.layers[-2].output
```

The dense layer is a neural network layer that is connected deeply, which means each neuron in the dense layer receives input from all neurons of its previous layer. The activation function is a mathematical "gate" in between the input feeding the current neuron and its output going to the next layer. It can be as simple as a step function that turns the neuron output on and off, depending on a rule or threshold. Here we're using relu as activation function.

```
final_output = layers.Dense(128)(base_output)
final_output = layers.Activation('relu')(final_output)
final_output = layers.Dense(64)(final_output)
final_output = layers.Activation('relu')(final_output)
final_output = layers.Dense(7, activation='softmax')(final_output)
```

Let's create our new model.

```
1 new_model = keras.Model(inputs = base_input, outputs = final_output)
```

Compile defines the loss function, the optimizer, and the metrics. That's all. It has nothing to do with the weights and we can compile a model as many times as we want without causing any problem to pre-trained weights. we need a compiled model to train (because training uses the loss function and the optimizer).

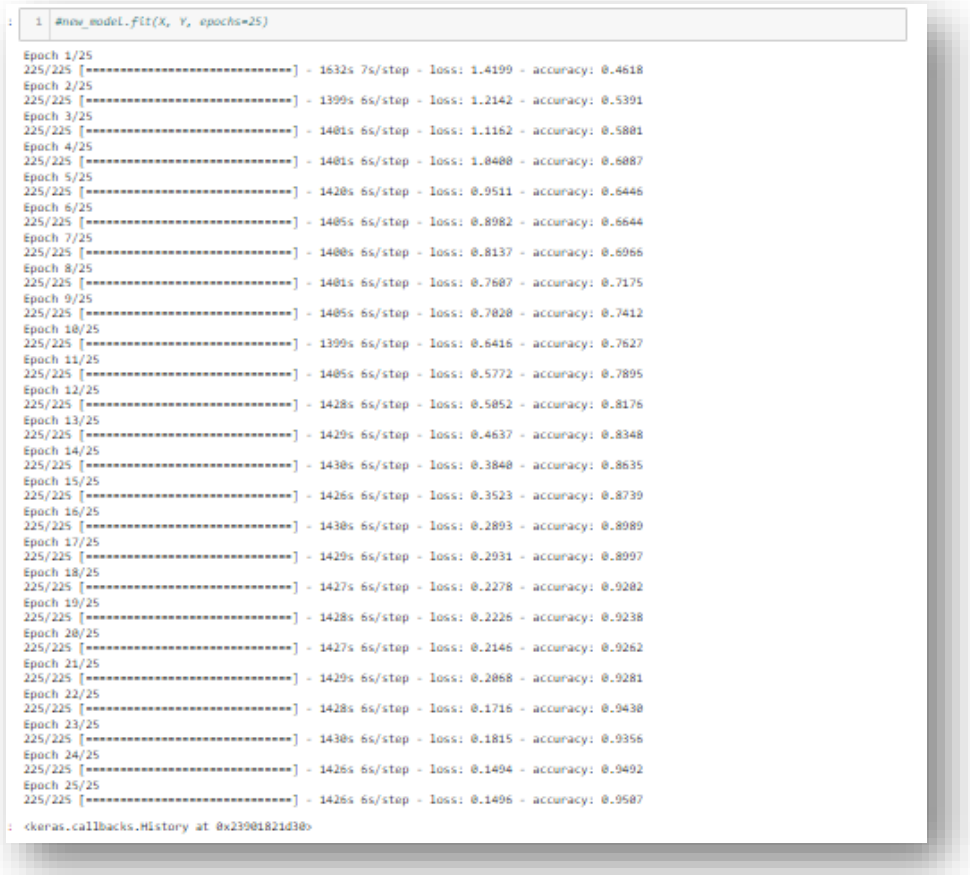
```
1 new_model.compile(loss="sparse_categorical_crossentropy", optimizer = "adam", metrics = ["accuracy"])
```

Optimizer, Loss Function and Metrics.

- **Loss Function used is categorical cross entropy.**
- **Loss function simply measures the absolute difference between our prediction and the actual value.**
- **Cross –entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. Cross- entropy loss increases as the predicted probability diverges from the actual label. So prediction a probability of 0.12 when the actual observation label is 1 would be bad and result in a high loss value.**

- **Optimizer used is the Adam() optimizer.**
- **Adam stands for Adaptive Moment Estimation. Adaptive Moment Estimation(Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta ,Adam also keeps and exponentially decaying average of past gradients M(t, similar to momentum:**

Trains The Model For 25 Number Of Epochs



```
In [ ]: 1 #new_model.fit(X, Y, epochs=25)

Epoch 1/25
225/225 [==============================] - 1632s 7s/step - loss: 1.4199 - accuracy: 0.4618
Epoch 2/25
225/225 [==============================] - 1399s 6s/step - loss: 1.2142 - accuracy: 0.5391
Epoch 3/25
225/225 [==============================] - 1401s 6s/step - loss: 1.1162 - accuracy: 0.5801
Epoch 4/25
225/225 [==============================] - 1401s 6s/step - loss: 1.0480 - accuracy: 0.6087
Epoch 5/25
225/225 [==============================] - 1420s 6s/step - loss: 0.9511 - accuracy: 0.6446
Epoch 6/25
225/225 [==============================] - 1405s 6s/step - loss: 0.8982 - accuracy: 0.6644
Epoch 7/25
225/225 [==============================] - 1400s 6s/step - loss: 0.8137 - accuracy: 0.6966
Epoch 8/25
225/225 [==============================] - 1401s 6s/step - loss: 0.7607 - accuracy: 0.7175
Epoch 9/25
225/225 [==============================] - 1405s 6s/step - loss: 0.7020 - accuracy: 0.7412
Epoch 10/25
225/225 [==============================] - 1399s 6s/step - loss: 0.6416 - accuracy: 0.7627
Epoch 11/25
225/225 [==============================] - 1405s 6s/step - loss: 0.5772 - accuracy: 0.7895
Epoch 12/25
225/225 [==============================] - 1428s 6s/step - loss: 0.5052 - accuracy: 0.8176
Epoch 13/25
225/225 [==============================] - 1429s 6s/step - loss: 0.4637 - accuracy: 0.8348
Epoch 14/25
225/225 [==============================] - 1430s 6s/step - loss: 0.3840 - accuracy: 0.8635
Epoch 15/25
225/225 [==============================] - 1426s 6s/step - loss: 0.3523 - accuracy: 0.8739
Epoch 16/25
225/225 [==============================] - 1430s 6s/step - loss: 0.2893 - accuracy: 0.8989
Epoch 17/25
225/225 [==============================] - 1429s 6s/step - loss: 0.2931 - accuracy: 0.8997
Epoch 18/25
225/225 [==============================] - 1427s 6s/step - loss: 0.2278 - accuracy: 0.9202
Epoch 19/25
225/225 [==============================] - 1428s 6s/step - loss: 0.2226 - accuracy: 0.9238
Epoch 20/25
225/225 [==============================] - 1427s 6s/step - loss: 0.2146 - accuracy: 0.9262
Epoch 21/25
225/225 [==============================] - 1429s 6s/step - loss: 0.2058 - accuracy: 0.9281
Epoch 22/25
225/225 [==============================] - 1428s 6s/step - loss: 0.1716 - accuracy: 0.9430
Epoch 23/25
225/225 [==============================] - 1430s 6s/step - loss: 0.1815 - accuracy: 0.9356
Epoch 24/25
225/225 [==============================] - 1426s 6s/step - loss: 0.1494 - accuracy: 0.9492
Epoch 25/25
225/225 [==============================] - 1426s 6s/step - loss: 0.1496 - accuracy: 0.9507

Out[ ]: <keras.callbacks.History at 0x23901821d30>
```

The model is saved

```
1  new_model.save('Second_Train.h5')
```

The model is loaded

```
1  new_model=tf.keras.models.load_model('Second_Train.h5')
```

The Model will be tested using a live webcam……

Validation

- In the validation phase , various OpenCV functions and keras functions have been used.
- Initially the video frame is stored in a video object.
- An LBP cascade classifier is used to detect facial region of interest .
- The image frame is converted into grayscale and resize and reshaped with the help of NumPy.
- Th resize image is fed to the predictor which is loaded by keras.model.load_model()function.
- The max argument is output.
- A rectangle is drawn around the facial regions and the output is formatted above the rectangular Box.

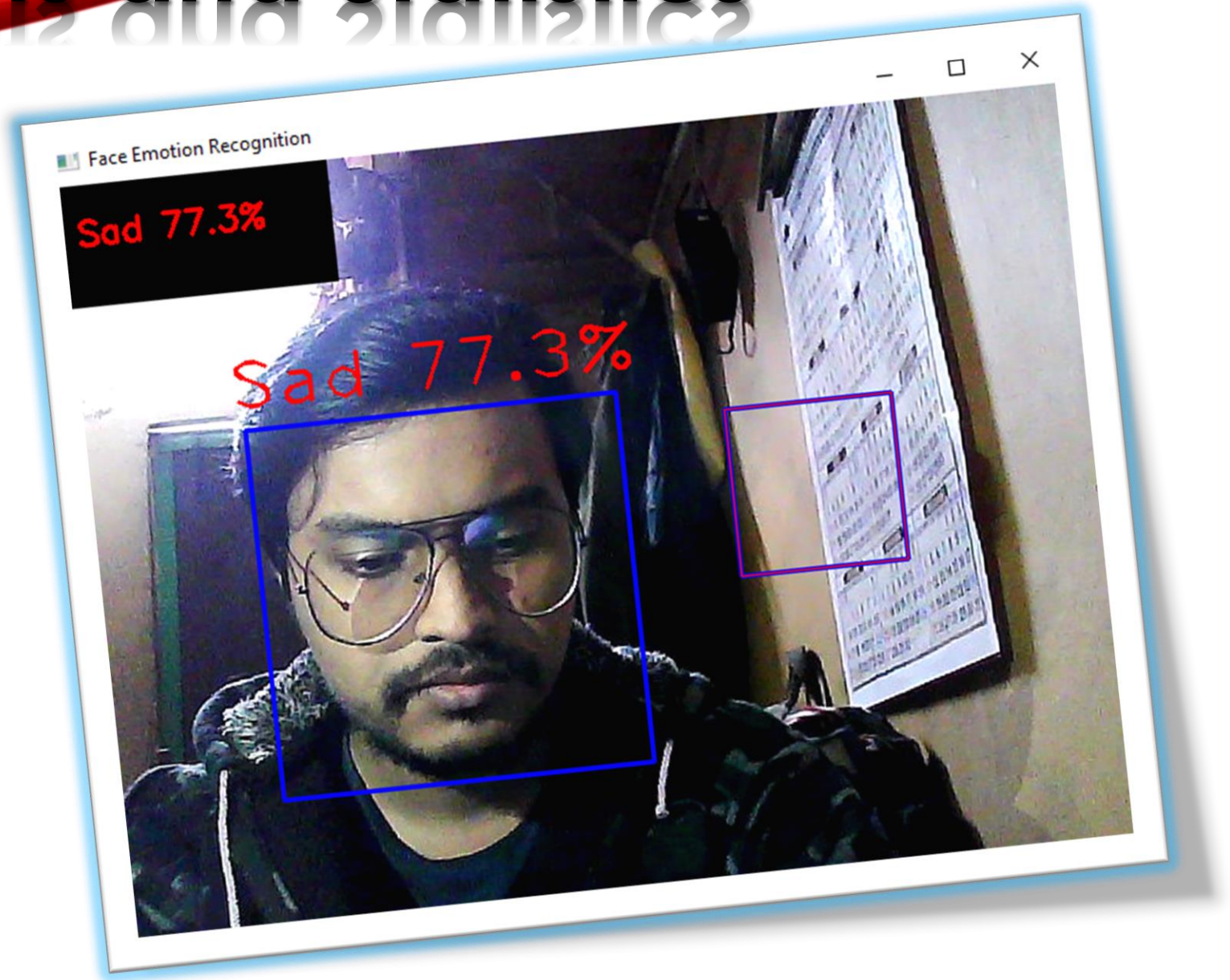
Performance Evaluation

The model gives 90-95% accuracy on validation set while training the model. The CNN model learns the representation features of emotions from the training images. Below are few epochs of training process with batch size of 24.

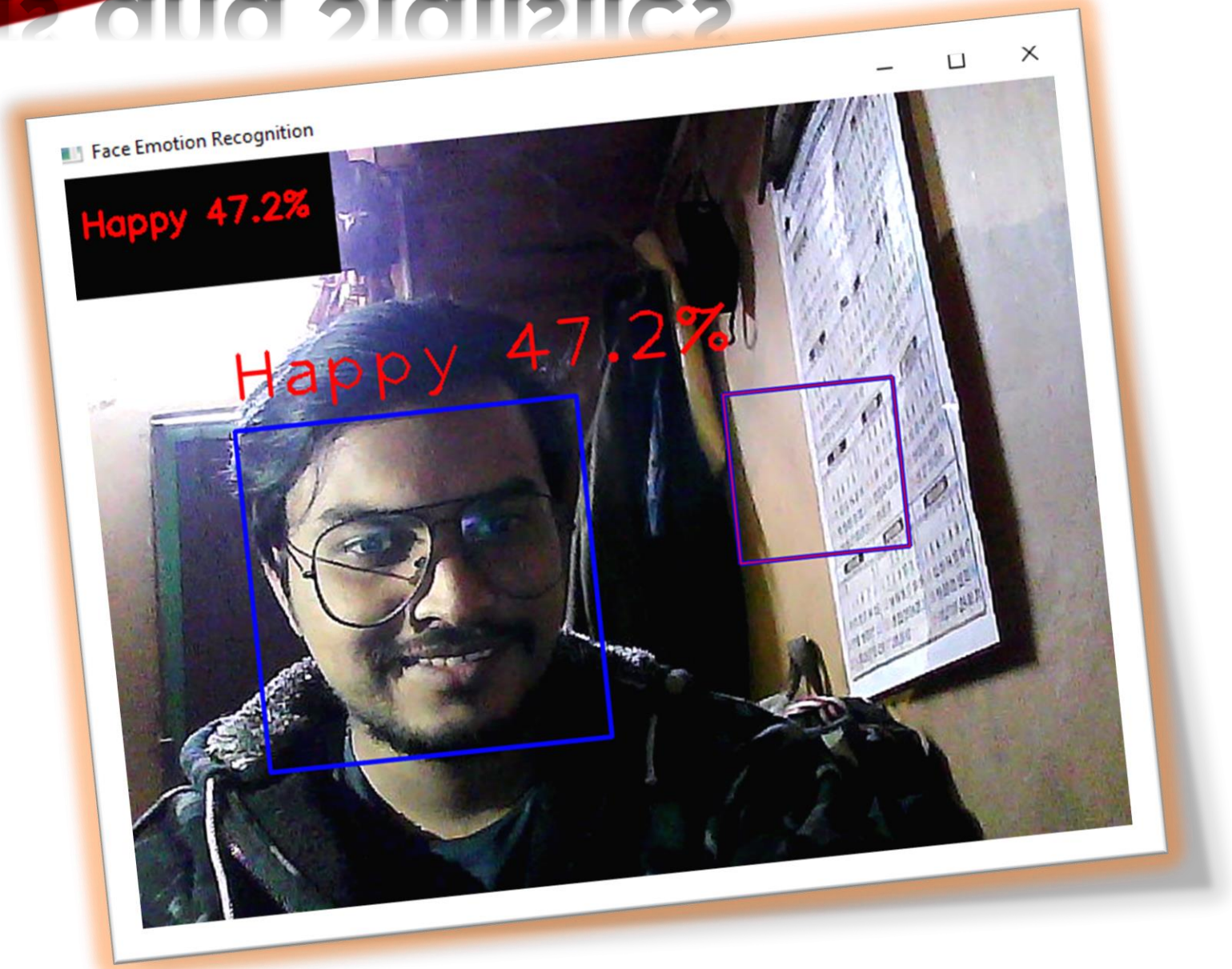
```
225/225 [==============================] - 1430s 6s/step - loss: 0.2893 - accuracy: 0.8989
Epoch 17/25
225/225 [==============================] - 1429s 6s/step - loss: 0.2931 - accuracy: 0.8997
Epoch 18/25
225/225 [==============================] - 1427s 6s/step - loss: 0.2278 - accuracy: 0.9202
Epoch 19/25
225/225 [==============================] - 1428s 6s/step - loss: 0.2226 - accuracy: 0.9238
Epoch 20/25
225/225 [==============================] - 1427s 6s/step - loss: 0.2146 - accuracy: 0.9262
Epoch 21/25
225/225 [==============================] - 1429s 6s/step - loss: 0.2068 - accuracy: 0.9281
Epoch 22/25
225/225 [==============================] - 1428s 6s/step - loss: 0.1716 - accuracy: 0.9430
Epoch 23/25
225/225 [==============================] - 1430s 6s/step - loss: 0.1815 - accuracy: 0.9356
Epoch 24/25
225/225 [==============================] - 1426s 6s/step - loss: 0.1494 - accuracy: 0.9492
Epoch 25/25
225/225 [==============================] - 1426s 6s/step - loss: 0.1496 - accuracy: 0.9507
```

Outputs and Statistics

Outputs and Statistics



Outputs and Statistics



THANK YOU