PROJECT

FaceEmotionRecognition

SoniaMehra, SouvikPaul

31*AUG* 2020

ABSTRACT

In this Project we have used Convolutional Neural Network to recognise emotion of a face from its image. We have classified the emotions as ‘*Angry*’, '*Disgust*', '*Fear*', '*Happy*', '*Sad*', '*Surprise*' and '*Neutral*'. ‘**FER2013**’ data, consists of 35887 observations/images with emotion labels, is used. Firstly, the data is divided into training set, validation set and test set with ratio *8:1:1*. Then from the training set we found out the number of observations corresponding to each label. Next, we trained the training data on different models and based on accuracy from the Validation set , we have chosen the best model and found its test accuracy. Lastly, we test our model with some real examples.Here, we have only provided the best of all the models which we have tried.

INTRODUCTION

In the era of **Artificial Intelligence** many unbelievable things become possible. Using a machine we can find out the person in front of you are telling lie or truth, just seeing a image of person **Facebook** can detect its name and details, hearing our voice **Google** understands and shows us the relevant websites etc. **CNN** in *Deep Learning* is such a tool to achieve perfection in image recognition. Significant additional impacts in image or object recognition were felt from 2011 to 2012. Although *CNN*s trained by backpropagation had been around for decades, and GPU implementations of NNs for years, including CNNs, fast implementations of *CNN*s with *max-pooling* on GPUs in the style of *Ciresan* and colleagues were needed to progress on computer vision. In 2011, this approach achieved for the first time superhuman performance in a visual pattern recognition contest. Also in 2011, it won the **ICDAR** Chinese handwriting contest, and in May 2012, it won the **ISBI** image segmentation contest. Until 2011, CNNs did not play a major role at computer vision conferences, but in June 2012, a paper by *Ciresan et al.* at the leading conference **CVPR** showed how *max-pooling CNN*s on GPU can dramatically improve many vision benchmark records. In November 2012, *Ciresan* et al.'s system also won the **ICPR** contest on analysis of large medical images for **cancer** detection.

Here we also used CNN to recognise face emotion by seeing a picture of a person.

OBJECTIVE

Objective of the project is to develop a face emotion recogniser based on 36K samples.

DATA INFORMATION

This ‘**FER2013**’ data is collected from Kaggle. There are 35887 examples in the data and each example is **48 x 48 x 1** grayscale image. Data is divided into 80%, 10% and 10% respectively for train set, dev set and test set. Thus we get 28709 examples in the train set and 3589 examples in both the dev set and test set. This data was firstly used in Kaggle Challenge and the winner got 71.161% test-accuracy. The data consists of 3 columns and 35888 rows (with first row as heading). First column is for *label* , second column is for *pixels* (but all pixels are in a single excel cell separated by space for a single observation) and third column is for name of the *emotion* corresponding to *label*. In our training set there are 441 ‘*disgust*’ images (lowest) and 7152 ‘*angry*’ images. Let’s do the analysis.

ANALYSIS

In the whole analysis various models are used. These models vary by **hyperparameters** such as *number of convnet layers*, *number of fully connected layers*, *number of filters*, *size of filter*, *padding*, *strides*, *activation function*. As there are seven categories , ‘*softmax*’ activation is in the final layer instead of ‘*sigmoid*’ (which is used for binary category). *Keras* is used in the analysis with *Tensorflow* at the backend.

Necessary modules to Import

In [33]:
```
import tensorflow as tf
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import keras
import random
from sklearn.model_selection import train_test_split
import seaborn as sns
import os
import sys
import math
```

The function below is to import data in such a way that the *pixel* values for each observation can be elements of an array.

In [34]:
```
def getData(filname):
    Y = []
    X = []
    first = True
    for line in open(filname):
        if first:
            first = False
        else:
            row = line.split(',')
            Y.append(int(row[0]))
            X.append([int(p) for p in row[1].split()])

    X, Y = np.array(X) / 255.0, np.array(Y)
    return X, Y
```

In [35]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/c
ontent/drive", force_remount=True).
```

In [36]:
```
X,Y=getData('/content/drive/My Drive/fer2013.csv')
```

Let's check the data.

In [37]:
```
print(X)
print("Type of X is:\n",type(X))
print(Y)
print("Type of Y is:\n",type(Y))
```

```
[[0.2745098  0.31372549 0.32156863 ... 0.41568627 0.42745098 0.32156863]
 [0.59215686 0.58823529 0.57647059 ... 0.75686275 0.71764706 0.72156863]
 [0.90588235 0.83137255 0.61176471 ... 0.34509804 0.43137255 0.59607843]
 ...
 [0.06666667 0.06666667 0.0627451  ... 0.60392157 0.52156863 0.44313725]
 [0.11764706 0.10980392 0.10980392 ... 0.1372549  0.11764706 0.10980392]
 [0.0745098  0.05098039 0.05490196 ... 0.74117647 0.78039216 0.78823529]]
Type of X is:
 <class 'numpy.ndarray'>
[0 0 2 ... 0 3 2]
Type of Y is:
 <class 'numpy.ndarray'>
```

In [38]:
```
X=X.reshape(35887,48,48,1)
print("Shape of X is:",X.shape)
print("Shape of Y is:",Y.shape)
```

```
Shape of X is: (35887, 48, 48, 1)
Shape of Y is: (35887,)
```

Here shape of **X** is $(m, n_H, n_W, 1)$ = (35887, 48, 48, 1) where m is the number of examples, n_H is number pixels in height, n_W is number of pixels in width of an image and 1 due to single layer grayimage. Label **Y** is a vector of 35887 elements.

In [39]:
```
X_train, X_test_val, y_train, y_test_val = train_test_split(X, Y, test_size=0.2,random_state
=0)
X_test, X_val, y_test, y_val = train_test_split(X_test_val, y_test_val, test_size=0.5,random
_state=0,shuffle=False)
```

In [40]:
```
print("Size of X_train:",X_train.shape[0])
print("Size of X_val:",X_val.shape[0])
print("Size of X_test:",X_test.shape[0])
```

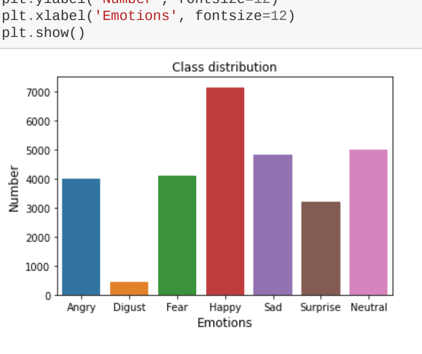
```
Size of X_train: 28709
Size of X_val: 3589
Size of X_test: 3589
```

Frequency distribution of each *emotion* in training set.

In [41]:
```
emotion_map = {0: 'Angry', 1: 'Digust', 2: 'Fear', 3: 'Happy', 4: 'Sad', 5: 'Surprise', 6:
'Neutral'}
A,B=np.unique(y_train,return_counts=True)
emotion_counts=pd.DataFrame({'emotion':A,'number':B})
emotion_counts['emotion'] = emotion_counts['emotion'].map(emotion_map)
print(emotion_counts)
```

```
    emotion  number
0     Angry    3992
1    Digust     441
2      Fear    4115
3     Happy    7152
4       Sad    4816
5  Surprise    3203
6   Neutral    4990
```

In [42]:
```
plt.figure(figsize=(6,4))
sns.barplot(emotion_counts.emotion, emotion_counts.number)
plt.title('Class distribution')
plt.ylabel('Number', fontsize=12)
plt.xlabel('Emotions', fontsize=12)
plt.show()
```



Importing necessary modules for *Keras*.

In [43]:
```
from keras.models import Sequential, Model
from keras.layers import Dense , Activation , Dropout ,Flatten, ZeroPadding2D,  Input
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D, AveragePooling2D
from keras.metrics import categorical_accuracy
from keras.models import model_from_json
from keras.callbacks import ModelCheckpoint
from keras.optimizers import *
from keras.layers.normalization import BatchNormalization
from keras.preprocessing import image
import pydot
```

In [44]:
```
random.seed(13)
```

Model

The structure of this model is shown below :-

Conv2D0 -> Conv2D1 -> Conv2D2 -> Batchnormalization0 -> Maxpool0 -> Dropout -> Cov2D3 -> Conv2D4 -> batchnormalization -> Maxpool1 -> Dropout -> Conv2D5 -> Conv2d6 -> Batchnormalization -> Maxpool2 -> Dropout -> FC0 -> Dropout -> FC1 ->Softmax

Note - (i) First arguement in ZeroPadding2D is pad size.

(ii) First, second arguement in Conv2D is number of filters and filter size.

(iii) In the arguement in *Avtivation* , function *relu* is used.

(iv) Arguement in *MaxPooling2D* is pool size

(v) *Flatten* converts the Convnet into simple nn layer and *Dense(7)* fully connects this simple layer with another simple nn layer with 7 neurons

(vi) *Batchnormalization* is technique to normalize the input layer by re-centering and re-scaling

(vii) *Dropout* is a regularization technique. *Dropout(p)* randomly removes px100% neurons in a layer.

In [45]:
```
def mymodel():
    input_shape=(48,48,1)
    X_input=Input(input_shape)

    X=Conv2D(64,(3,3),activation='relu',name='conv0')(X_input)
    X=Conv2D(64,(3,3),activation='relu',name='conv1')(X)
    X=Conv2D(64,(3,3),activation='relu',name='conv2')(X)
    X=BatchNormalization()(X)
    X=MaxPooling2D((2,2),name='maxpool0')(X)
    X=Dropout(0.5)(X)

    X=Conv2D(128,(3,3),activation='relu',name='conv3')(X)
    X=Conv2D(128,(3,3),activation='relu',name='conv4')(X)
    X=BatchNormalization()(X)
    X=MaxPooling2D((2,2),name='maxpool1')(X)
    X=Dropout(0.5)(X)

    X=Conv2D(256,(3,3),activation='relu',name='conv5')(X)
    X=Conv2D(256,(3,3),activation='relu',name='conv6')(X)
    X=BatchNormalization()(X)
    X=MaxPooling2D((2,2),name='maxpool2')(X)
    X=Dropout(0.5)(X)

    X=Flatten()(X)
    X=Dense(1024,activation='relu')(X)
    X=Dropout(0.4)(X)
    X=Dense(7, activation='softmax')(X)

    model=Model(inputs=X_input, outputs=X, name='mymodel')
    return model
```

In [46]:
```
face_model=mymodel()
face_model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
y1=y_train
y_train=(np.arange(7)==y_train[:,None]).astype(np.float32)
```

In [47]:
```
y_val=(np.arange(7)==y_val[:,None]).astype(np.float32)
```

In [68]:
```
history=face_model.fit(x=X_train, y=y_train, epochs=50, batch_size=64,verbose=1, validation_
data=(X_val,y_val) )
```

```
Epoch 1/50
  1/449 [..............................] - ETA: 11s - loss: 0.5333 - accuracy: 0.8125WARNING:
tensorflow:Callbacks method `on_train_batch_end` is slow compared to the batch time (batch ti
me: 0.0163s vs `on_train_batch_end` time: 0.0265s). Check your callbacks.
449/449 [==============================] - 14s 31ms/step - loss: 0.6448 - accuracy: 0.7571 -
val_loss: 1.0789 - val_accuracy: 0.6590
Epoch 2/50
449/449 [==============================] - 14s 31ms/step - loss: 0.6342 - accuracy: 0.7642 -
val_loss: 1.0932 - val_accuracy: 0.6598
Epoch 3/50
449/449 [==============================] - 14s 31ms/step - loss: 0.6361 - accuracy: 0.7648 -
val_loss: 1.1347 - val_accuracy: 0.6514
Epoch 4/50
449/449 [==============================] - 14s 30ms/step - loss: 0.6238 - accuracy: 0.7673 -
val_loss: 1.1123 - val_accuracy: 0.6595
Epoch 5/50
449/449 [==============================] - 14s 30ms/step - loss: 0.6126 - accuracy: 0.7733 -
val_loss: 1.0685 - val_accuracy: 0.6601
Epoch 6/50
449/449 [==============================] - 14s 30ms/step - loss: 0.6110 - accuracy: 0.7723 -
val_loss: 1.0802 - val_accuracy: 0.6604
Epoch 7/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5880 - accuracy: 0.7806 -
val_loss: 1.1135 - val_accuracy: 0.6525
Epoch 8/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5895 - accuracy: 0.7760 -
val_loss: 1.1178 - val_accuracy: 0.6601
Epoch 9/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5870 - accuracy: 0.7795 -
val_loss: 1.1154 - val_accuracy: 0.6623
Epoch 10/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5826 - accuracy: 0.7841 -
val_loss: 1.0907 - val_accuracy: 0.6431
Epoch 11/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5685 - accuracy: 0.7880 -
val_loss: 1.0851 - val_accuracy: 0.6500
Epoch 12/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5632 - accuracy: 0.7913 -
val_loss: 1.1757 - val_accuracy: 0.6617
Epoch 13/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5523 - accuracy: 0.7947 -
val_loss: 1.1139 - val_accuracy: 0.6573
Epoch 14/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5403 - accuracy: 0.7963 -
val_loss: 1.1491 - val_accuracy: 0.6498
Epoch 15/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5440 - accuracy: 0.7968 -
val_loss: 1.1447 - val_accuracy: 0.6567
Epoch 16/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5407 - accuracy: 0.7972 -
val_loss: 1.1037 - val_accuracy: 0.6634
Epoch 17/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5241 - accuracy: 0.8062 -
val_loss: 1.1826 - val_accuracy: 0.6634
Epoch 18/50
449/449 [==============================] - 14s 30ms/step - loss: 0.5274 - accuracy: 0.8040 -
val_loss: 1.1169 - val_accuracy: 0.6484
Epoch 19/50
449/449 [==============================] - 14s 30ms/step - loss: 0.5181 - accuracy: 0.8076 -
val_loss: 1.1055 - val_accuracy: 0.6525
Epoch 20/50
449/449 [==============================] - 14s 30ms/step - loss: 0.5073 - accuracy: 0.8115 -
val_loss: 1.1427 - val_accuracy: 0.6434
Epoch 21/50
449/449 [==============================] - 14s 31ms/step - loss: 0.5129 - accuracy: 0.8107 -
val_loss: 1.1674 - val_accuracy: 0.6567
Epoch 22/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4993 - accuracy: 0.8148 -
val_loss: 1.1699 - val_accuracy: 0.6523
Epoch 23/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4971 - accuracy: 0.8175 -
val_loss: 1.1180 - val_accuracy: 0.6590
Epoch 24/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4869 - accuracy: 0.8218 -
val_loss: 1.1674 - val_accuracy: 0.6498
Epoch 25/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4836 - accuracy: 0.8229 -
val_loss: 1.1939 - val_accuracy: 0.6584
Epoch 26/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4795 - accuracy: 0.8233 -
val_loss: 1.1169 - val_accuracy: 0.6578
Epoch 27/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4694 - accuracy: 0.8273 -
val_loss: 1.1866 - val_accuracy: 0.6528
Epoch 28/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4719 - accuracy: 0.8263 -
val_loss: 1.1049 - val_accuracy: 0.6567
Epoch 29/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4667 - accuracy: 0.8289 -
val_loss: 1.1485 - val_accuracy: 0.6562
Epoch 30/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4607 - accuracy: 0.8285 -
val_loss: 1.1464 - val_accuracy: 0.6634
Epoch 31/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4606 - accuracy: 0.8325 -
val_loss: 1.1464 - val_accuracy: 0.6506
Epoch 32/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4588 - accuracy: 0.8304 -
val_loss: 1.1249 - val_accuracy: 0.6562
Epoch 33/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4402 - accuracy: 0.8396 -
val_loss: 1.1393 - val_accuracy: 0.6648
Epoch 34/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4390 - accuracy: 0.8399 -
val_loss: 1.1623 - val_accuracy: 0.6634
Epoch 35/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4444 - accuracy: 0.8394 -
val_loss: 1.1985 - val_accuracy: 0.6428
Epoch 36/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4344 - accuracy: 0.8397 -
val_loss: 1.1912 - val_accuracy: 0.6539
Epoch 37/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4218 - accuracy: 0.8455 -
val_loss: 1.2578 - val_accuracy: 0.6584
Epoch 38/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4267 - accuracy: 0.8419 -
val_loss: 1.2191 - val_accuracy: 0.6551
Epoch 39/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4248 - accuracy: 0.8451 -
val_loss: 1.1816 - val_accuracy: 0.6578
Epoch 40/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4209 - accuracy: 0.8461 -
val_loss: 1.1813 - val_accuracy: 0.6581
Epoch 41/50
449/449 [==============================] - 14s 30ms/step - loss: 0.4355 - accuracy: 0.8462 -
val_loss: 1.1339 - val_accuracy: 0.6503
Epoch 42/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4196 - accuracy: 0.8476 -
val_loss: 1.1933 - val_accuracy: 0.6539
Epoch 43/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4054 - accuracy: 0.8545 -
val_loss: 1.1759 - val_accuracy: 0.6598
Epoch 44/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4051 - accuracy: 0.8522 -
val_loss: 1.1697 - val_accuracy: 0.6567
Epoch 45/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4043 - accuracy: 0.8524 -
val_loss: 1.1805 - val_accuracy: 0.6542
Epoch 46/50
449/449 [==============================] - 14s 30ms/step - loss: 0.3961 - accuracy: 0.8550 -
val_loss: 1.1957 - val_accuracy: 0.6525
Epoch 47/50
449/449 [==============================] - 14s 31ms/step - loss: 0.3977 - accuracy: 0.8561 -
val_loss: 1.2307 - val_accuracy: 0.6517
Epoch 48/50
449/449 [==============================] - 14s 31ms/step - loss: 0.4012 - accuracy: 0.8568 -
val_loss: 1.1904 - val_accuracy: 0.6587
Epoch 49/50
449/449 [==============================] - 14s 31ms/step - loss: 0.3859 - accuracy: 0.8598 -
val_loss: 1.1788 - val_accuracy: 0.6604
Epoch 50/50
449/449 [==============================] - 14s 31ms/step - loss: 0.3779 - accuracy: 0.8631 -
val_loss: 1.2094 - val_accuracy: 0.6648
```

Model is run two times with *epoch=50*, so total *epoch=100*

In [69]:
```
preds=face_model.evaluate(x=X_test, y= (np.arange(7)==y_test[:,None]).astype(np.float32))
print('loss = '+str(preds[0]))
print('test accuracy = '+str(preds[1]))
```

```
113/113 [==============================] - 1s 6ms/step - loss: 1.2331 - accuracy: 0.6500
loss = 1.2331316471099854
test accuracy = 0.6500418186187744
```

We train the model using 50 *epochs* with *mini batch* size = 64. We get the training accuracy 85.91% and validation accuracy 64.73%.This model is pretty good among all the model which we have used behind the scenes.

Experiment with some Real Examples

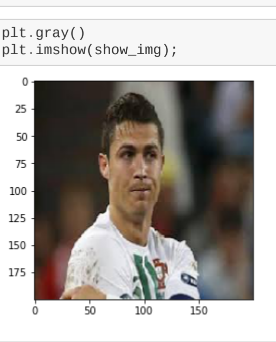
In [84]:
```
from skimage import io

img = image.load_img('/content/drive/My Drive/ronaldo.jpg', grayscale=True, target_size=(48,
48));

x = image.img_to_array(img)
x = np.expand_dims(x, axis = 0)
x/=255
custom = face_model.predict(x)
x = np.array(x, 'float32')
x = x.reshape([48, 48]);
```

```
/usr/local/lib/python3.6/dist-packages/keras_preprocessing/image/utils.py:107: UserWarning: g
rayscale is deprecated. Please use color_mode = "grayscale"
  warnings.warn('grayscale is deprecated. Please use '
```

In [85]:
```
show_img=image.load_img('/content/drive/My Drive/ronaldo.jpg', grayscale=False, target_size=
(200, 200))
```

In [86]:
```
m=0.000000000000000000001
a=custom[0]
for i in range(0,len(a)):
    if a[i]>m:
        m=a[i]
        ind=i
objects = ['angry', 'disgust', 'fear', 'happy', 'sad', 'surprise', 'neutral']
```

In [87]:
```
plt.gray()
plt.imshow(show_img);
```

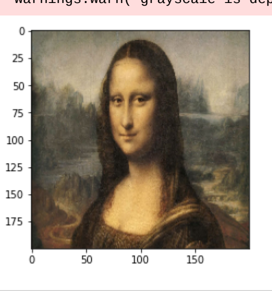


In [88]:
```
print('Expression Prediction:',objects[ind])
```

```
Expression Prediction: fear
```

In [77]:
```
img = image.load_img('/content/drive/My Drive/monalisa.jpg', grayscale=True, target_size=(48
, 48));

x = image.img_to_array(img)
x = np.expand_dims(x, axis = 0)
x/=255
custom = face_model.predict(x)
x = np.array(x, 'float32')
x = x.reshape([48, 48]);
show_img=image.load_img('/content/drive/My Drive/monalisa.jpg', grayscale=False, target_size
=(200, 200))
m=0.000000000000000000001
a=custom[0]
for i in range(0,len(a)):
    if a[i]>m:
        m=a[i]
        ind=i
plt.gray()
plt.imshow(show_img);
```

```
/usr/local/lib/python3.6/dist-packages/keras_preprocessing/image/utils.py:107: UserWarning: g
rayscale is deprecated. Please use color_mode = "grayscale"
  warnings.warn('grayscale is deprecated. Please use '
```

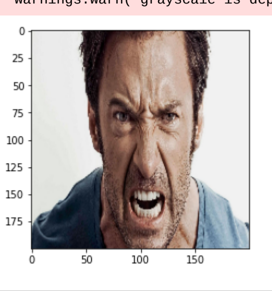


In [78]:
```
print('Expression Prediction:',objects[ind])
```

```
Expression Prediction: sad
```

In [89]:
```
img = image.load_img('/content/example.jpg', grayscale=True, target_size=(48, 48));

x = image.img_to_array(img)
x = np.expand_dims(x, axis = 0)
x/=255
custom = face_model.predict(x)
x = np.array(x, 'float32')
x = x.reshape([48, 48]);
show_img=image.load_img('/content/example.jpg', grayscale=False, target_size=(200, 200))
m=0.000000000000000000001
a=custom[0]
for i in range(0,len(a)):
    if a[i]>m:
        m=a[i]
        ind=i
plt.gray()
plt.imshow(show_img);
```

```
/usr/local/lib/python3.6/dist-packages/keras_preprocessing/image/utils.py:107: UserWarning: g
rayscale is deprecated. Please use color_mode = "grayscale"
  warnings.warn('grayscale is deprecated. Please use '
```

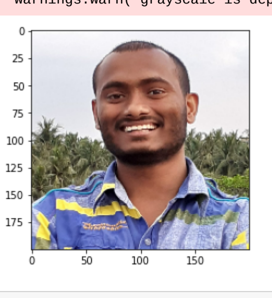


In [90]:
```
print('Expression Prediction:',objects[ind])
```

```
Expression Prediction: angry
```

In [99]:
```
img = image.load_img('/content/example22.jpeg', grayscale=True, target_size=(48, 48));

x = image.img_to_array(img)
x = np.expand_dims(x, axis = 0)
x/=255
custom = face_model.predict(x)
x = np.array(x, 'float32')
x = x.reshape([48, 48]);
show_img=image.load_img('/content/example22.jpeg', grayscale=False, target_size=(200, 200))
m=0.000000000000000000001
a=custom[0]
for i in range(0,len(a)):
    if a[i]>m:
        m=a[i]
        ind=i
plt.gray()
plt.imshow(show_img);
```

```
/usr/local/lib/python3.6/dist-packages/keras_preprocessing/image/utils.py:107: UserWarning: g
rayscale is deprecated. Please use color_mode = "grayscale"
  warnings.warn('grayscale is deprecated. Please use '
```

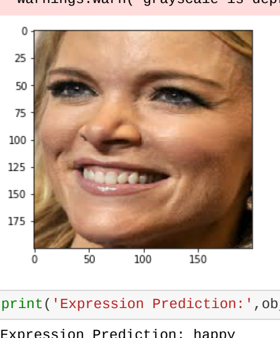


In [100]:
```
print('Expression Prediction:',objects[ind])
```

```
Expression Prediction: happy
```

In [97]:
```
img = image.load_img('/content/example2.jpg', grayscale=True, target_size=(48, 48));

x = image.img_to_array(img)
x = np.expand_dims(x, axis = 0)
x/=255
custom = face_model.predict(x)
x = np.array(x, 'float32')
x = x.reshape([48, 48]);
show_img=image.load_img('/content/example2.jpg', grayscale=False, target_size=(200, 200))
m=0.000000000000000000001
a=custom[0]
for i in range(0,len(a)):
    if a[i]>m:
        m=a[i]
        ind=i
plt.gray()
plt.imshow(show_img);
```

```
/usr/local/lib/python3.6/dist-packages/keras_preprocessing/image/utils.py:107: UserWarning: g
rayscale is deprecated. Please use color_mode = "grayscale"
  warnings.warn('grayscale is deprecated. Please use '
```



In [98]:
```
print('Expression Prediction:',objects[ind])
```

```
Expression Prediction: happy
```

Normalized Confusion Matrix of test data

In [75]:
```
from sklearn.metrics import classification_report, confusion_matrix
y_test=(np.arange(7)==y_test[:,None]).astype(np.float32)
test_true = np.argmax(y_test, axis=1)
test_pred = np.argmax(face_model.predict(X_test), axis=1)
```

In [76]:
```
def plot_confusion_matrix(y_true, y_pred, classes,
                          normalize=False,
                          title=None,
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if not title:
        if normalize:
            title = 'Normalized confusion matrix'
        else:
            title = 'Confusion matrix, without normalization'

    # Compute confusion matrix
    cm = confusion_matrix(y_true, y_pred)
    # Only use the labels that appear in the data
    classes = classes
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        #print("Normalized confusion matrix")
    #else:
        #print('Confusion matrix, without normalization')

    #print(cm)

    fig, ax = plt.subplots(figsize=(12,6))
    im = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    ax.figure.colorbar(im, ax=ax)
    # We want to show all ticks...
    ax.set(xticks=np.arange(cm.shape[1]),
           yticks=np.arange(cm.shape[0]),
           # ... and label them with the respective list entries
           xticklabels=classes, yticklabels=classes,
           title=title,
           ylabel='True label',
           xlabel='Predicted label')

    # Rotate the tick labels and set their alignment.
    plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
             rotation_mode="anchor")

    # Loop over data dimensions and create text annotations.
    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            ax.text(j, i, format(cm[i, j], fmt),
                    ha="center", va="center",
                    color="white" if cm[i, j] > thresh else "black")
    fig.tight_layout()
    return ax
plot_confusion_matrix(test_true, test_pred, classes=['Angry', 'Disgust', 'Fear', 'Happy', 'S
ad', 'Surprise', 'Neutral'],
                      normalize=True, title='Normalized confusion matrix')
plt.show()
```