Emotion Detection from Images using Machine Learning

AUTHOR : KASHIF MOIN

Index

Introduction

The data consists of **48x48** pixel grayscale images of faces. The faces have been automatically registered so that the face is more or less centred and occupies about the same amount of space in each image. *The task is to categorize each face based on the emotion shown in the facial expression into **one of seven** categories →*

0 : Angry
1 : Disgust
2 : Fear
3 : Happy
4 : Sad
5 : Surprise
6 : Neutral

train.csv contains two columns, "**emotion**" and "**pixels**".

The "***emotion***" column contains a numeric code ranging from *0 to 6*, inclusive, for the emotion that is present in the image.

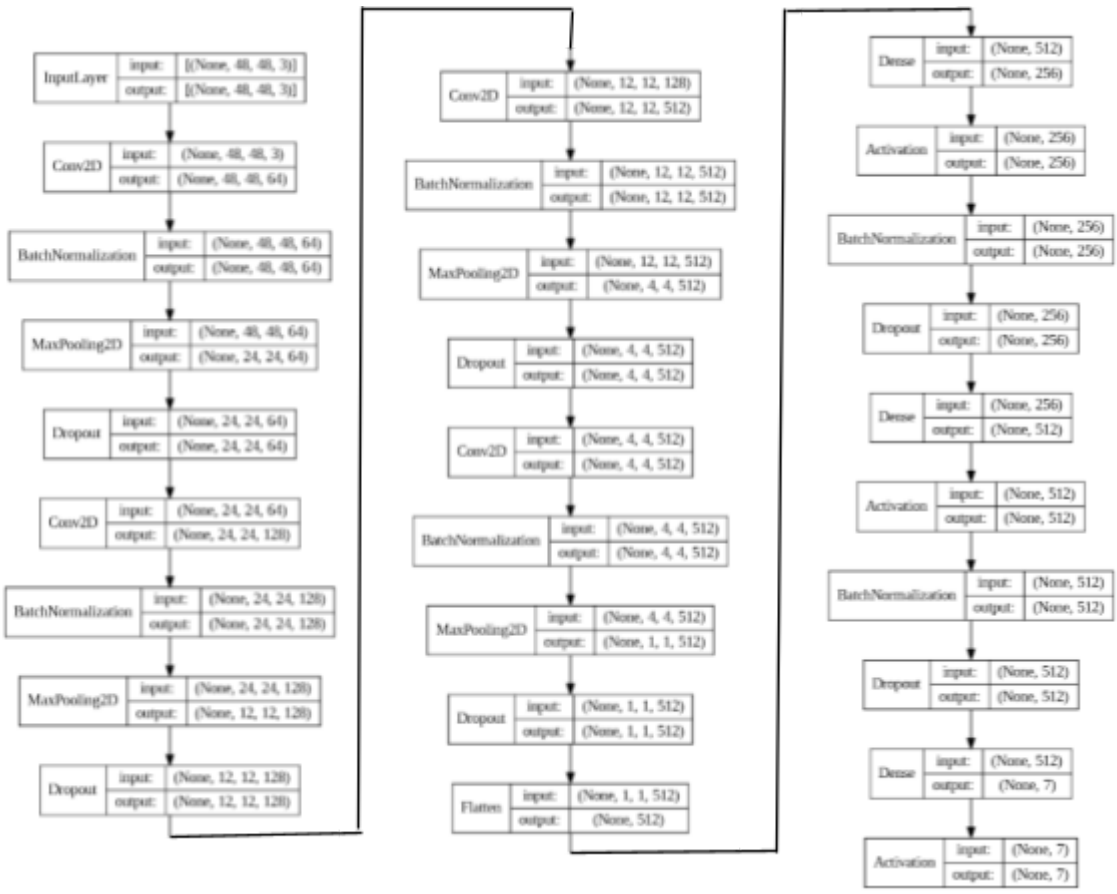
The "***pixels***" column contains a string surrounded by quotes for each image. The contents of this string a space-separated pixel values in row-major order.

The ***training set*** consists of **32,298 images**.

The ***test set*** consists of **3,589 images**.

Model Utilized

Sequential Model is used to specify a neural network, precisely, sequentially: from input to output, passing through a series of neural layers, one after the other.



Libraries

For the completion of this model, we have used various Python libraries, namely, *pandas, numpy, plotly, sklearn, tensorflow, scikitplot, and keras.*

Importing Important Libraries

```
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings('ignore')

# plotting
import plotly.express as px
import matplotlib.pyplot as plt
from sklearn import metrics
from sklearn.metrics import confusion_matrix
import scikitplot
from sklearn.metrics import classification_report


# preprocessing
import cv2
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras.utils import to_categorical

# Data splitting
from sklearn.model_selection import train_test_split

# Deep Learning Libraries
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.layers import Dense, Input, Dropout, GlobalAveragePooling2D, Flatten, Conv2D, BatchNormalization, Activ
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.optimizers import Adam, SGD, RMSprop
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, ReduceLROnPlateau
```

Data Preprocessing

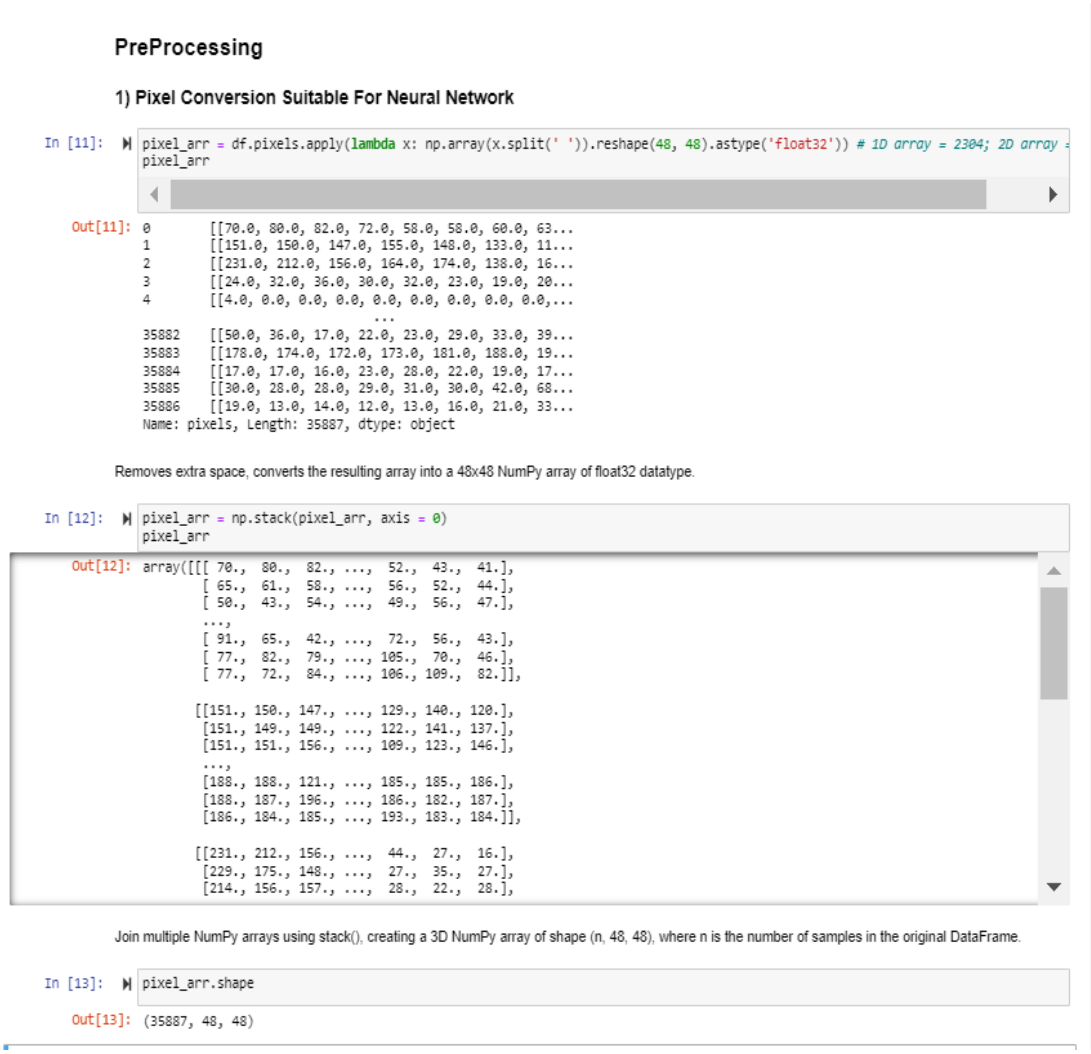
1. Pixel Conversion suitable for Neural Network

In this step, we convert the pixel of each image [df['pixels']] into a numpy array of size **48x48**, and datatype **float**.

Further, we remove the **whitespaces** present in this column

After converting each row into a numpy array, we use **numpy.stack()** function to join these arrays and create a three-dimensional numpy array of shape **(n, 48, 48)**; where ***n*** is the number of samples in the original dataframe.

Note → The reason for creating a 2D numpy array of size (48, 48) is that each row in a single dimension was of size (2304,). Hence, converting the 1D array into a 2D array 48x48 (= 2304) was the only solution.



PreProcessing

1) Pixel Conversion Suitable For Neural Network

```
In [11]: pixel_arr = df.pixels.apply(lambda x: np.array(x.split(' ')).reshape(48, 48).astype('float32')) # 1D array = 2304; 2D array =
         pixel_arr
```

```
Out[11]: 0        [[70.0, 80.0, 82.0, 72.0, 58.0, 58.0, 60.0, 63...
         1        [[151.0, 150.0, 147.0, 155.0, 148.0, 133.0, 11...
         2        [[231.0, 212.0, 156.0, 164.0, 174.0, 138.0, 16...
         3        [[24.0, 32.0, 36.0, 30.0, 32.0, 23.0, 19.0, 20...
         4        [[4.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
                                      ...
         35882    [[50.0, 36.0, 17.0, 22.0, 23.0, 29.0, 33.0, 39...
         35883    [[178.0, 174.0, 172.0, 173.0, 181.0, 188.0, 19...
         35884    [[17.0, 17.0, 16.0, 23.0, 28.0, 22.0, 19.0, 17...
         35885    [[30.0, 28.0, 28.0, 29.0, 31.0, 30.0, 42.0, 68...
         35886    [[19.0, 13.0, 14.0, 12.0, 13.0, 16.0, 21.0, 33...
         Name: pixels, Length: 35887, dtype: object
```

Removes extra space, converts the resulting array into a 48x48 NumPy array of float32 datatype.

```
In [12]: pixel_arr = np.stack(pixel_arr, axis = 0)
         pixel_arr
```

```
Out[12]: array([[[ 70.,  80.,  82., ...,  52.,  43.,  41.],
                 [ 65.,  61.,  58., ...,  56.,  52.,  44.],
                 [ 50.,  43.,  54., ...,  49.,  56.,  47.],
                 ...,
                 [ 91.,  65.,  42., ...,  72.,  56.,  43.],
                 [ 77.,  82.,  79., ..., 105.,  70.,  46.],
                 [ 77.,  72.,  84., ..., 106., 109.,  82.]],

                [[151., 150., 147., ..., 129., 140., 120.],
                 [151., 149., 149., ..., 122., 141., 137.],
                 [151., 151., 156., ..., 109., 123., 146.],
                 ...,
                 [188., 188., 121., ..., 185., 185., 186.],
                 [188., 187., 196., ..., 186., 182., 187.],
                 [186., 184., 185., ..., 193., 183., 184.]],

                [[231., 212., 156., ...,  44.,  27.,  16.],
                 [229., 175., 148., ...,  27.,  35.,  27.],
                 [214., 156., 157., ...,  28.,  22.,  28.],
```

Join multiple NumPy arrays using stack(), creating a 3D NumPy array of shape (n, 48, 48), where n is the number of samples in the original DataFrame.

```
In [13]: pixel_arr.shape
```

```
Out[13]: (35887, 48, 48)
```

2. Converting Numpy Arrays to RGB format

After the conversion of the pixels into the 3D array, we first convert the array into the ***grayscale*** format and then into a 4D array in ***RGB*** format, which is *suitable for neural networks*.

It is not always mandatory to convert a 3D array of images into a 4D array in RGB format. But, many CNN architectures require input images to have RGB format instead of grayscale format. If let's say, we provide grayscale images instead of RGB images, many CNN architectures may not be able to learn relevant features from the input images, leading to poor performance. Therefore, we are converting our input images from grayscale format to RGB format.

2) Converting images to RGB format

It is not always mandatory to convert a 3D array images into a 4D array of RGB format. But, many CNN architectures require input images to have RGB format instead of grayscale format. If let's say we provide grayscale images instead of RGB images, many CNN architectures may not be able to learn relevant features from the input images, leading to poor performance. Therefore, we are converting our input images from grayscale format to RGB format.

In [14]:
```
def RGB(pixel):
    image = []
    for i in range(len(pixel)):
        pix = cv2.cvtColor(pixel[i], cv2.COLOR_GRAY2RGB)
        image.append(pix)
    image = np.array(image)
    return image
```

In [15]:
```
pixel_rgb  = RGB(pixel_arr)
```

In [16]:
```
pixel_rgb.shape
```

Out[16]:
```
(35887, 48, 48, 3)
```

In [17]:
```
pixel_rgb[2].astype(np.uint8)
```

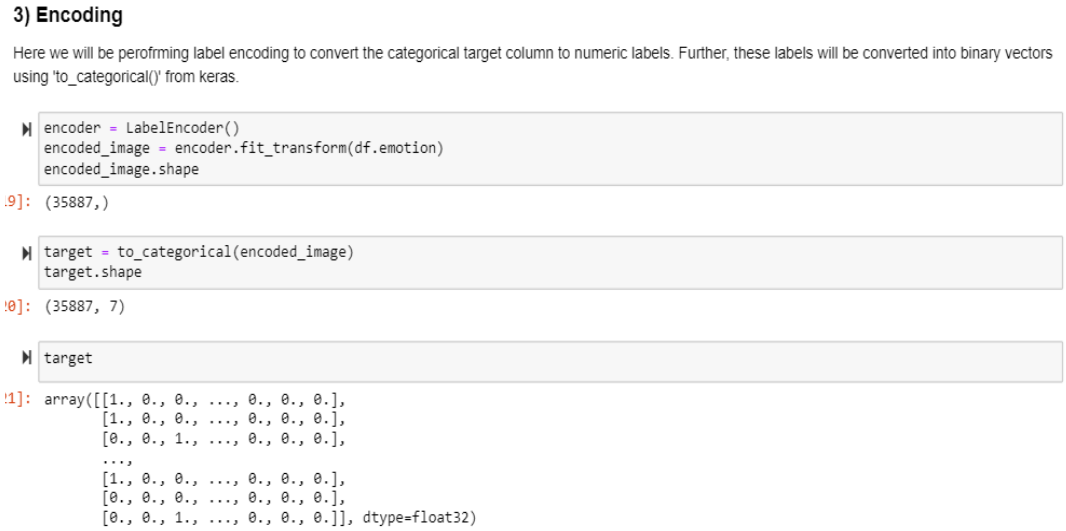
Out[17]:
```
array([[[231, 231, 231],
        [212, 212, 212],
        [156, 156, 156],
        ...,
        [ 44,  44,  44],
        [ 27,  27,  27],
        [ 16,  16,  16]],

       [[229, 229, 229],
        [175, 175, 175],
        [148, 148, 148],
        ...,
        [ 27,  27,  27],
        [ 35,  35,  35],
        [ 27,  27,  27]],

       [[214, 214, 214],
        [156, 156, 156],
        [157, 157, 157],
```

In [18]:
```
plt.figure(1, (3, 3))
plt.imshow(pixel_rgb[2].astype(np.uint8))
plt.tight_layout()
```

3. Encoding

Since our model is a *multiclass classification model*, it will have more than two labels as the target value. There are many Machine Learning/Deep Learning models that only take numeric labels.

So, in this step, we convert our categorical target column to numeric labels using ***LabelEncoding()***. Later, these labels will be converted into binary vectors by using ***to_categorical()*** from **keras**.



3) Encoding

Here we will be perofrming label encoding to convert the categorical target column to numeric labels. Further, these labels will be converted into binary vectors using 'to_categorical()' from keras.

```
encoder = LabelEncoder()
encoded_image = encoder.fit_transform(df.emotion)
encoded_image.shape
```

```
(35887,)
```

```
target = to_categorical(encoded_image)
target.shape
```

```
(35887, 7)
```

```
target
```

```
array([[1., 0., 0., ..., 0., 0., 0.],
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       ...,
       [1., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.]], dtype=float32)
```

Data Splitting

We are split our dataset into two parts :

- Train - (32298, 48, 48, 3)
- Test - (3589, 48, 48, 3)

For splitting the data we use ***train_test_split()*** from **sklearn.metrics**.

Data Splitting

Train - Test Split

```
X_train, X_cv, y_train, y_cv = train_test_split(pixel_rgb, target, shuffle = True, stratify = target,
                                                test_size=0.1, random_state=42)
```

stratifying preserves the proportion of how data is distributed in the target column

```
print("TRAIN SIZE")
print("X-Train:",X_train.shape)
print("Y-Train:",y_train.shape[0])
```

```
TRAIN SIZE
X-Train: (32298, 48, 48, 3)
Y-Train: 32298
```

```
print("CV SIZE")
print("X-CV:",X_cv.shape)
print("Y-CV:",y_cv.shape[0])
```

```
CV SIZE
X-CV: (3589, 48, 48, 3)
Y-CV: 3589
```

Data Normalisation

These datset images are of 48×48 pixels are represented as an array of numbers whose values range from **[0, 255]**. Therefore to normalize the dataset, we divide the data by 255.0 as neural networks are very sensitive to unnormalized data.

Data Normalization

These datset images are of 48×48 pixels are represented as an array of numbers whose values range from [0, 255]. There fore to normalize the dataset, we will divide the data by 255.0 as neural networks are very sensitive to unnormalized data.

```
X_train /= 255.0
X_cv /= 255.0
```

Modeling

1. Model Creation

Sequential Model is used to specify a neural network, precisely, sequentially: from input to output, passing through a series of neural layers, one after the other.

In this sequential model, we use various layers such as Flatten, Conv2D, BatchNormalisation, MaxPooling2D, Dropout, Activation and Dense.

→**Conv2D Layer :**
1] filters: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution). During network training, the filters are updated in a way that minimizes the loss.

2] kernal_size: This parameter determines the dimensions of the kernel. Common dimensions include 1×1, 3×3, 5×5, and 7×7 which can be passed as (1, 1), (3, 3), (5, 5), or (7, 7) tuples.

3] Activation: Activations can either be used through an Activation layer, or through the activation argument. There are various types of activation functions, they are:

^ relu function: Applies the rectified linear unit activation function.
^ sigmoid function: Applies the sigmoid activation function. For small values (<-5), sigmoid returns a value close to zero, and for large values (>5) the result of the function gets close to 1. Sigmoid function always returns a value between 0 and 1.
^ softmax function: Softmax converts a vector of values to a probability distribution. The elements of the output vector are in range (0, 1) and sum to 1.
^ tanh function: Hyperbolic tangent activation function.
^ selu function:
^ elu function: Exponential Linear Unit. ELUs have negative values which pushes the mean of the activations closer to zero. Mean activations that are closer to zero enable faster learning as they bring the gradient closer to the natural gradient. ELUs saturate to a negative value when the argument gets smaller. Saturation means a small derivative which decreases the variation and the information that is propagated to the next layer.

4] Padding: The padding parameter of the Keras Conv2D class can take one of two values: ‘valid’ or ‘same’.
"valid" means no padding.
“Same” means same padding.

5] input_shape: *input_shape = (width,height,depth)*

→**BatchNormalization Layer:**
Layer that normalizes its inputs.Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1.

→**MaxPooling2D Layer:**

We need to define the pool size and what it does is, it extracts important information from that area of that image.

→Dropout Layer:

The dropout layer is used to prevent the model from overfitting. The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting.

→Flatten Layer:

Flatten is used to flatten the input. For example, if flatten is applied to layer having input shape as (batch_size, 2,2), then the output shape of the layer will be (batch_size, 4). Flatten is used to transform the data into 1D array.

Modeling

1) Model Creation

```
model = Sequential()

#1st CNN Layer
model.add(Conv2D(filters = 64, kernel_size=(3, 3), padding = 'same', activation = 'relu', input_shape = (width,height,depth)))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(0.25))

#2nd CNN Layer
model.add(Conv2D(filters = 128, kernel_size=(5, 5), padding = 'same', activation = 'relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2,2)))
model.add(Dropout(0.25))

#3rd CNN Layer
model.add(Conv2D(filters = 512, kernel_size=(5, 5), padding = 'same', activation = 'relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (3, 3)))
model.add(Dropout(0.25))

#4th CNN Layer
# model.add(Conv2D(filters = 256, kernel_size=(3, 3), padding = 'same', activation = 'relu'))
# model.add(BatchNormalization())
# model.add(MaxPooling2D(pool_size = (3, 3)))
# model.add(Dropout(0.25))

#5th CNN Layer
model.add(Conv2D(filters = 512, kernel_size=(5, 5), padding = 'same', activation = 'relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (3, 3)))
model.add(Dropout(0.25))

model.add(Flatten())

model.add(Dense(256))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Dropout(0.25))

model.add(Dense(512))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Dropout(0.25))

#Output Layer
model.add(Dense(labels))
model.add(Activation('softmax'))

model.compile(optimizer= Adam(lr=0.0003), loss = "categorical_crossentropy",  metrics=['accuracy'])
model.summary()
```

2. Model Fitting

Before training our model, we have to define certain parameters : *early_stopping, lr_reduction, epochs, batchsize.*

early_stopping is a regularization technique for deep neural networks that stops training when parameter updates no longer begin to yield improves on a validation set.

lr_reduction is a technique for deep neural networks for reducing the learning rate when a metric has stopped improving.

epoch is basically the number of cycles for which our model has to be trained but for some reason, if our model starts overfitting, early stopping will terminate the further execution.

batchsize depicts the number of samples that propagate through the neural network before updating the model parameters. As a rule of thumb, we use batchsize as 32.

2) Model Fitting

```
early_stopping = EarlyStopping(monitor='val_accuracy', min_delta=0, patience=7, verbose=1, restore_best_weights=True)
lr_reduction = ReduceLROnPlateau(monitor='val_accuracy', factor=0.5, patience=5, min_lr=1e-7, verbose=1)

callbacks = [early_stopping, lr_reduction]
epochs = 50
batch_size = 32
```

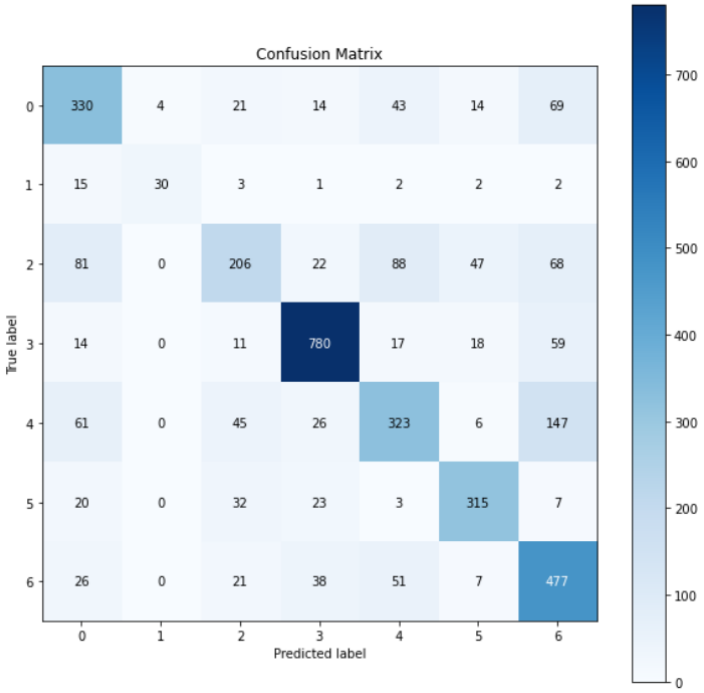
Observation and Conclusion

Confusion Matrix

The **confusion matrix** gives a comparison between actual and predicted values. It is used for the optimization of machine learning models. The confusion matrix is a N x N matrix, where **N** is the number of classes or outputs.

```
scikitplot.metrics.plot_confusion_matrix(y_test, y_pred, figsize=(10,10))
```

```
<AxesSubplot:title={'center':'Confusion Matrix'}, xlabel='Predicted label', ylabel='True label'>
```



F1 Score and Accuracy

For evaluating the classification data, we have the confusion matrix where we classify our data into **TP**, **TN, FP**, and **FN**.
In binary classification we consider the **F1 score** to be a measure of the model's accuracy and is calculated by using the:

F1 = 2*(precision*recall)/(precision+recall)

Where, **precision = TP/(TP+FP)**
And, **recall = TP/(TP+FN)**
We see scores for F1 between 0 and 1, where **0** is the *worst score* and **1** is the *best score.*

While **precision** refers to the percentage of your results which are relevant, **recall** refers to the percentage of total relevant results correctly classified by your algorithm.

Higher precision means that an algorithm returns more relevant results than irrelevant ones, and *high recall* means that an algorithm returns most of the relevant results (whether or not irrelevant ones are also returned).

Classification accuracy is a metric that summarizes the performance of a classification model as the number of correct predictions divided by the total number of predictions.

	precision	recall	f1-score	support
0	0.60	0.67	0.63	495
1	0.88	0.55	0.67	55
2	0.61	0.40	0.48	512
3	0.86	0.87	0.87	899
4	0.61	0.53	0.57	608
5	0.77	0.79	0.78	400
6	0.58	0.77	0.66	620
accuracy			0.69	3589
macro avg	0.70	0.65	0.67	3589
weighted avg	0.69	0.69	0.68	3589

*From this model, we **conclude** that the dataset is heavily imbalanced. There is not enough images for emotions like Disgust.*

Out of 3589 images in our test data, the model was able to correctly predict 2461 images, with an accuracy of 69%.

Bonus Challenge

Overview -

Additionally, as a bonus challenge, if time permits, you can work on the implementation of emotion detection inference for video inputs instead of single images.

Implementation -

For this task, we have used various python libraries like load_model from keras, img_to_array, image from keras.preprocessing, cv2, numpy.

```
from keras.models import load_model
from time import sleep
from keras.preprocessing.image import img_to_array
from keras.preprocessing import image
import cv2
import numpy as np
```

cv2.CascadeClassifier is a machine learning-based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images. In our case, we have used ***haarcascade_frontalface_default.xml***. It uses the function detectMultiScale to detect the upper left corner of the rectangle (x, y) on the face as well as width and height of the rectangle.

Load_model : You can save a model with model.save() or keras.models.save_model() (which is equivalent). You can load it back with keras.models.load_model() .

```
face_classifier = cv2.CascadeClassifier(r"C:\Users\kashi\ML Facial Emotion\haarcascade_frontalface_default.xml")
classifier =load_model(r"C:\Users\kashi\ML Facial Emotion\cnnmodel.h5")
```

For capturing the image we have used **cv2.VideoCapture(0)**. Here 0 is a parameter that means the default web camera of the laptop.

```
emotion_labels = ['Angry','Disgust','Fear','Happy','Neutral', 'Sad', 'Surprise']

cap = cv2.VideoCapture(0)
```

After capturing the image from our camera, it is first converted into a grayscale format.

Once the image has been converted into grayscale, **face_classifier.detectMultiScale(gray)** is used to detect faces in our captured image.

Each image can have multiple faces. So for that, we will use the for loop and **cv2.rectangle()** that will draw a rectangle around each face. This region inside the yellow rectangle is our region of interest. We are also resizing this region of interest into a (48,48) size image because the model is trained only on images of size (48,48).

Next, we are normalizing our images by dividing them by 255.0 and converting the image into an array using **img_to_array(roi)**. Once the image has been converted into an array, we are converting our images into RGB format.

Once all the preprocessing is done, we can now make the predictions. The output will be in the form of numerical values that will be mapped with the list of emotions that we created earlier. For some reason, if our camera is not able to detect any face, it will display "No Faces"

At last, we are using **cv2.imshow('Emotion Detector',frame)** for displaying the output.

```
while True:
    _, frame = cap.read()
    labels = []
    gray = cv2.cvtColor(frame,cv2.COLOR_BGR2GRAY)
    faces = face_classifier.detectMultiScale(gray)

    for (x,y,w,h) in faces:
        cv2.rectangle(frame,(x,y),(x+w,y+h),(0,255,255),2)
        roi_gray = gray[y:y+h,x:x+w]
        roi_gray = cv2.resize(roi_gray,(48,48),interpolation=cv2.INTER_AREA)



        if np.sum([roi_gray])!=0:
            roi = roi_gray.astype('float')/255.0
            roi = img_to_array(roi)
            roi = np.expand_dims(roi,axis=0)
            roi  = RGB(roi)

            prediction = classifier.predict(roi)[0]
            label=emotion_labels[prediction.argmax()]
            label_position = (x,y-10)
            cv2.putText(frame,label,label_position,cv2.FONT_HERSHEY_SIMPLEX,1,(0,255,0),2)
        else:
            cv2.putText(frame,'No Faces',(30,80),cv2.FONT_HERSHEY_SIMPLEX,1,(0,255,0),2)
    cv2.imshow('Emotion Detector',frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cap.release()
cv2.destroyAllWindows()
```

Output -