Digital Forensics course, A.A. 2020/2021

Costantini Lorenzo

Student ID 2029018

07th September 2021

Project 2 “Face Detection” Report

Index

Introduction

The target of this Project was described as:

“Starting from the lab experience n.4, build an emotion recognition strategy using the dataset provided at the address:

https://www.kaggle.com/deadskull7/fer2013 “.

I used Python as programming language, in particular with the help of Google Colaboratory, in order to execute the code in a faster and safer way.

Chapter 1

Emotion Recognition

In this Chapter, I will present a little more in detail the Challenge, with some code presentation and a description of the architecture that I choose to use.

1.1 Description of the Project

In the previous Introduction, I have briefly described the target of this project, as a "Emotion Recognition Strategy".
First of all, we start by choosing and defining a dataset. In this case, we used fer2013.csv, made by more than 30000 values divided in 3 different columns:

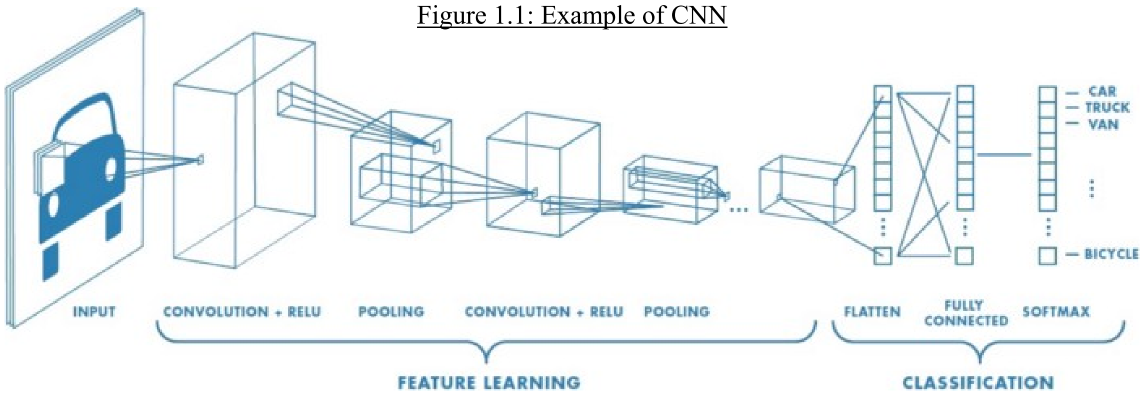
- The first one, defined as "emotion", is composed by 7 different numbers (from 0 to 6), each one, in our representation, stays for a particular emotion. In our case, 0=Angry, 1=Disgust, 2=Fear, 3=Happy, 4=Sad, 5=Surprise, 6=Neutral;
- The second column, called "pixels", contains a list of integers, that represents a specific image;
- The last one, called Usage, describes the effective usage of the dataset, divided in 3 different categories: Training, PublicTest and PrivateTest.

In the instruction of the Project, it was given a free choice in what architecture could be used, like SVM, CNN etc….
Personally, I tried both SVM and CNN, and I find better results with this last one. Obviously, I tried several combinations of different CNN-based architecture until I found the one that satisfy the requirements of a good precision and, not for a second choice, a good efficiency.

1.2 General Description of CNNs

The Convolutional Neural Networks are a particular type of Deep Network, introduced in 1988 by Y.LeCun [1].
They are often used for tasks like image recognition, objects or, in particular, face recognition.
Their working phase is similar to the Deep Networks: they receive some input data (for example, images), process it and then, they label it and associate it to a particular pre-defined class.
To do this, the algorithm have to receive the image as a pixels array, distributed in the three-dimensions (height, width and depth).
During the working process, every input data has to pass through a series of different levels (figure 1.1), each one with a different functionality. The first level is called Convolutional, characterized by the respective input data, that defines some of its features. It is often associated with the activation function ReLU (Rectified Linear Unit), that helps in nullifying the negative values. In this way, all the CNN is faster and efficient, without impact in the accuracy.
After this level, we have the Pooling one, that helps in reducing the number of parameters.
Then, we have a Flatten level, followed by a Fully Connected one, that essentially connects all the previous neurons in order to establish the different classes. This is done by taking the output of the previous level and by generating an N-dimensional vector, where N is the number of classes. Inside this vector, there are all the probability that each input data belongs to a particular class.

Figure 1.1: Example of CNN



1.3 Approach

First of all, I have imported the dataset fer2013.csv from Google Drive, where I have previously saved it.
Differently from usual, the dataset was not directly composed by images, but from raw pixels, so I need to implement a pre-processing phase.
I have split the Dataset in 3 parts: Training set, Validation set and Test set.
As usual, the Training set is used to train the model, while the Validation one is used to set and decide the values for the hyperparameters (for example, by testing on it different combinations of hyperparameters). Finally, the Test set is used to get the final results and to evaluate the real performances of the model.
After that, I've found a very interesting function, that has helped me in pre-processing data. I have re-adapted it to be useful in my model, and it is able to convert the strings (pixels) in a list of integers, then it has reshaped and encoded the obtained data.
Now, I have finally defined all that I need for the Training phase.
As said before, I decided to use a CNN-based model.
In particular, a first module made by a 2D Convolutional layer, with dimension 128, kernel size 3x3 and input shape equal to the dimension of a single image (48x48). I choose ReLU activation, for the reasons described in the previous paragraph.
Then, I decide to use a normalization layer, to normalize the input and scale it. In this way, the model becomes easier to train.
The second module is similar to the previous, the convolutional layer has the same dimension (128), but with two additional layers: after the normalization one, a MaxPooling layer (that, as said in the previous paragraph, reduces the number of parameters in order to get an easier way to train the model).
Finally, a Dropout layer, that helps to prevent overfitting, by randomly setting inputs units to 0 (it drops out some neurons at random during each epoch, in order to regularize), with rate 0.25.

The third module is similar to the second, but now the convolutional layer has dimension 64.

The fourth module is similar to the second one, but the convolutional layer now has dimension 32.

I have done different trials, adding and removing the Pooling and the Dropout layer in the last 3 modules. I find that they are useful in each of the three modules. If I remove these 2 layers only on one module (for example, in the middle one (Third)), the difference is slightly noticeable in terms of accuracy or time. If I remove them from more than a module, the risk of incurring in overfitting increase, while the time needed is almost the same. So, thanks to these trials, I decided to implement them in all the three modules, also giving a sort of symmetry.

After that, I have a Flatten layer, followed by three equals modules made by a Dense layer (of dimension 128), a normalization layer and a dropout one (with rate 0.5).

Finally, I defined the output layer as a Dense layer, with dimension 7 (number of classes/emotions) and softmax as activation (because it provides a categorical output).

After that, I have plotted the accuracy and loss graphs, followed by the confusion matrices, first without normalization, then with normalization.

Chapter 2

Code Review

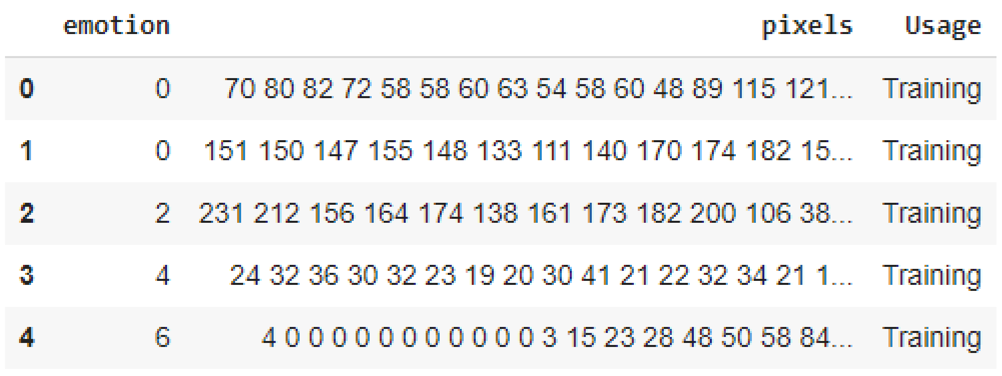
In this chapter, I will review the code, with some explanations and some clarifications.

2.1 Imports and Dataset

```
from google.colab import drive
drive.mount('/content/drive')
data_path = '/content/drive/MyDrive/dataset/fer2013.csv'
data=pd.read_csv(data_path)
data.head()
```

In this cell, I have imported the Dataset from my Google Drive folder. The variable data path should be changed for every user, in order to import correctly the dataset. The command data.head() show me the first 5 rows of the Dataset, in order to have an idea of what is inside and how it is organized (Figure 2.1).

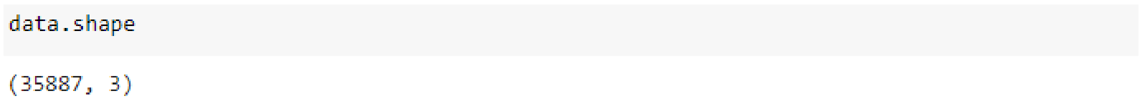
Figure 2.1: Output of “data.head()



	emotion	pixels	Usage
0	0	70 80 82 72 58 58 60 63 54 58 60 48 89 115 121...	Training
1	0	151 150 147 155 148 133 111 140 170 174 182 15...	Training
2	2	231 212 156 164 174 138 161 173 182 200 106 38...	Training
3	4	24 32 36 30 32 23 19 20 30 41 21 22 32 34 21 1...	Training
4	6	4 0 0 0 0 0 0 0 0 0 0 0 0 3 15 23 28 48 50 58 84...	Training

Then, I used some commands in order to have a better idea of the dataset:

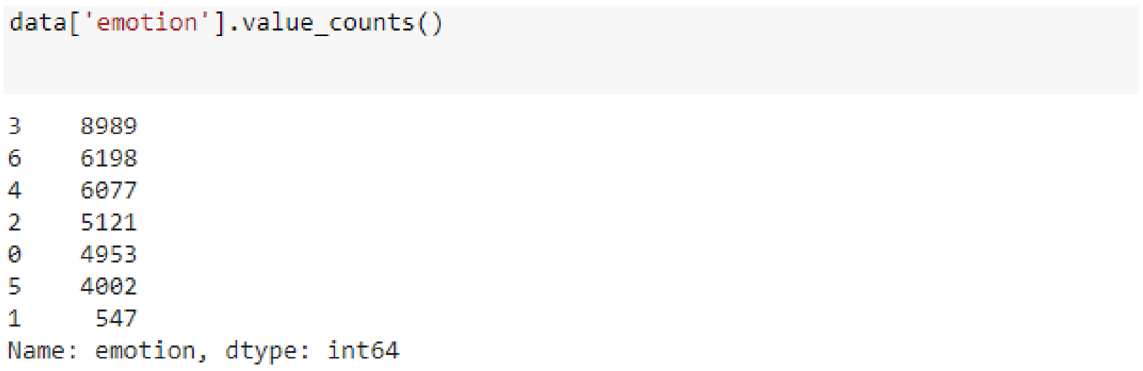
- With data.shape, I showed the dimensions of the dataset:



```
data.shape
```

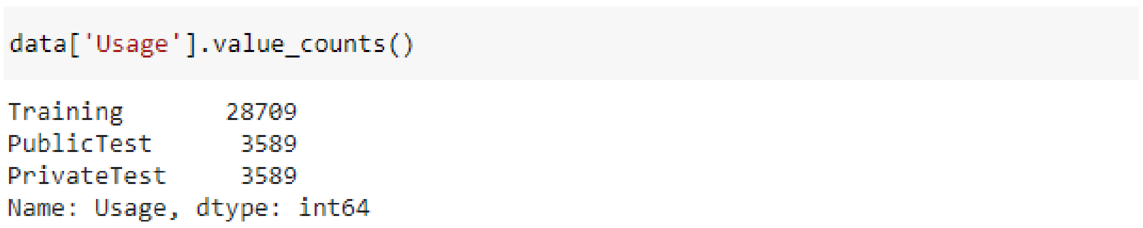
```
(35887, 3)
```

- Now, I showed how many examples are represented for each emotion (class) and the disposal of the dataset:



```
data['emotion'].value_counts()
```

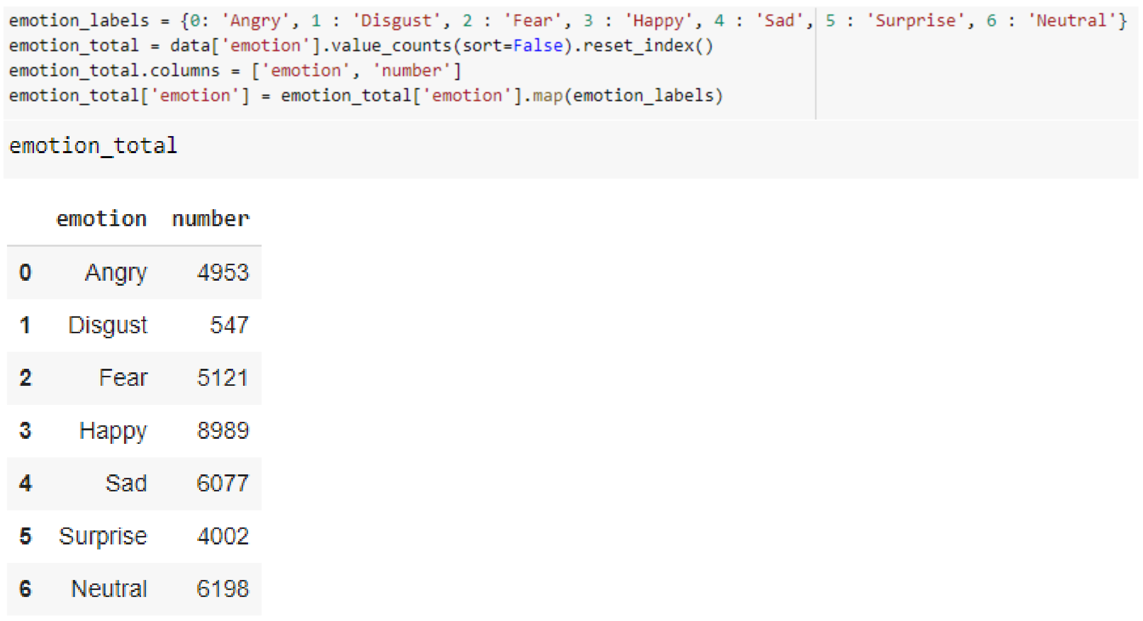
```
3    8989
6    6198
4    6077
2    5121
0    4953
5    4002
1     547
Name: emotion, dtype: int64
```



```
data['Usage'].value_counts()
```

```
Training       28709
PublicTest      3589
PrivateTest     3589
Name: Usage, dtype: int64
```

- Then, I defined the label for each emotion, by associating each number to a particular class. After that, I showed more clearly the table shown before (with emotion labels):



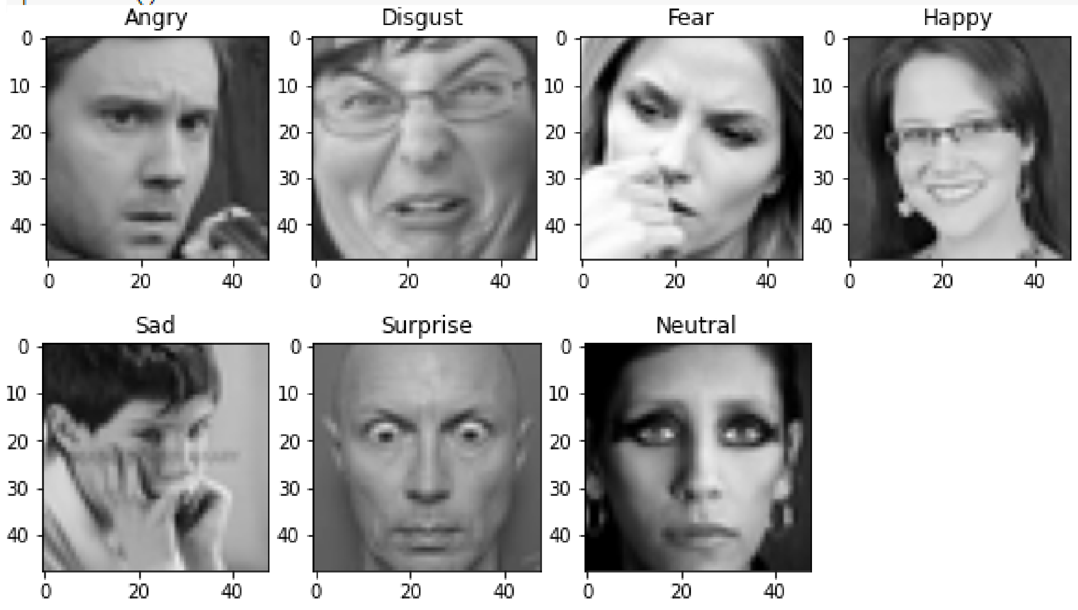
```
emotion_labels = {0: 'Angry', 1 : 'Disgust', 2 : 'Fear', 3 : 'Happy', 4 : 'Sad', 5 : 'Surprise', 6 : 'Neutral'}
emotion_total = data['emotion'].value_counts(sort=False).reset_index()
emotion_total.columns = ['emotion', 'number']
emotion_total['emotion'] = emotion_total['emotion'].map(emotion_labels)
```

```
emotion_total
```

	emotion	number
0	Angry	4953
1	Disgust	547
2	Fear	5121
3	Happy	8989
4	Sad	6077
5	Surprise	4002
6	Neutral	6198

- Finally, I printed some examples of images in the dataset, one for each possible classes, by putting pixels in an array of integers and reshaping, in order to get an image from raw pixels:

```
def imageshow(row):
    pixels = row['pixels']
    emotion = emotion_labels[row['emotion']]
    img = np.array(pixels.split())
    img = img.reshape(48,48)
    image = np.zeros((48,48,3))
    image[:,:,0] = img
    image[:,:,1] = img
    image[:,:,2] = img
    return np.array([image.astype(np.uint8), emotion])

plt.figure(figsize=(10,10))

for i in range(1,8):
    face = data[data['emotion'] == i-1].iloc[0]
    img = imageshow(face)
    plt.subplot(2,4,i)
    plt.imshow(img[0])
    plt.title(img[1])

plt.show()
```



2.2 Pre-Processing

Now, I have to split the Dataset in 3 different parts: one for Training, one for Validation and the last one for Test.

I've done this by using the different labels that were included in the original Dataset, so that the Training set is extracted from what it was labelled as "Training", the Validation set from "PublicTest" and the Test set from "PrivateTest".

After that, I've showed the shape of these 3 sets as a check of the correctness.

```
data_train = data[data['Usage']=='Training']
data_val   = data[data['Usage']=='PublicTest']
data_test  = data[data['Usage']=='PrivateTest']
print("train shape = ", data_train.shape)
print("validation shape = ", data_val.shape)
print("test shape = ", data_test.shape)
```

```
train shape =  (28709, 3)
validation shape =  (3589, 3)
test shape =  (3589, 3)
```

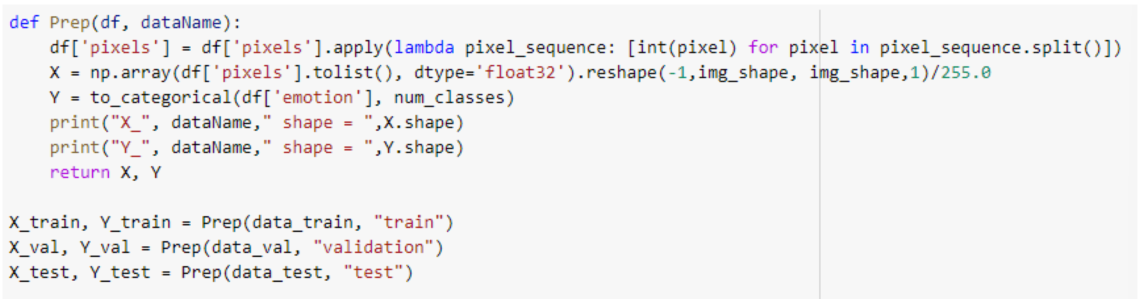
As we could see from the shapes, the Training set is almost an 80% of the total dataset.

After that, I've defined some parameters that I will need in the next steps.

```
num_classes = 7
img_shape = 48
num_epochs = 50
batch_size = 128
```

The number of classes represents the number of emotions (7 in our case), the image dimension is 48x48, so that img_shape is defined as 48. I've increased the number of epochs from 10 to 50, in order to obtain a better accuracy, stopping before overfitting. I choose a batch size of 128, after some trials with 64 and 256. I think that 128 is a good compromise in terms of efficiency and correctness.

Now, I’ve implemented a useful function, found in a documented similar task, that, as said in the previous chapter, it converts the strings (pixels) in a list of integers, then it reshapes them and encode the obtained data.

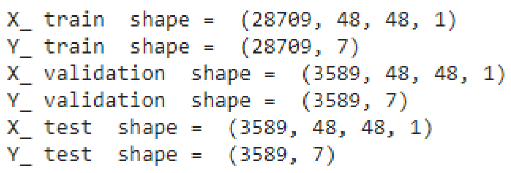


```python
def Prep(df, dataName):
    df['pixels'] = df['pixels'].apply(lambda pixel_sequence: [int(pixel) for pixel in pixel_sequence.split()])
    X = np.array(df['pixels'].tolist(), dtype='float32').reshape(-1,img_shape, img_shape,1)/255.0
    Y = to_categorical(df['emotion'], num_classes)
    print("X_", dataName," shape = ",X.shape)
    print("Y_", dataName," shape = ",Y.shape)
    return X, Y

X_train, Y_train = Prep(data_train, "train")
X_val, Y_val = Prep(data_val, "validation")
X_test, Y_test = Prep(data_test, "test")
```

As it is visible, in the first two rows it operates the conversion and, in particular in the second row, it stores the values in a numpy array, right after having them reshaped.

The third row uses the function “to_categorical”, applied to the numpy array defined before. This function converts a numpy array, which has integers that represent different categories, to a numpy array which has binary values and columns equals to the number of categories in the data. This helps in the analysis and in the processing of data. Finally, I’ve showed the shapes of the obtained outputs (Figure 2.2).

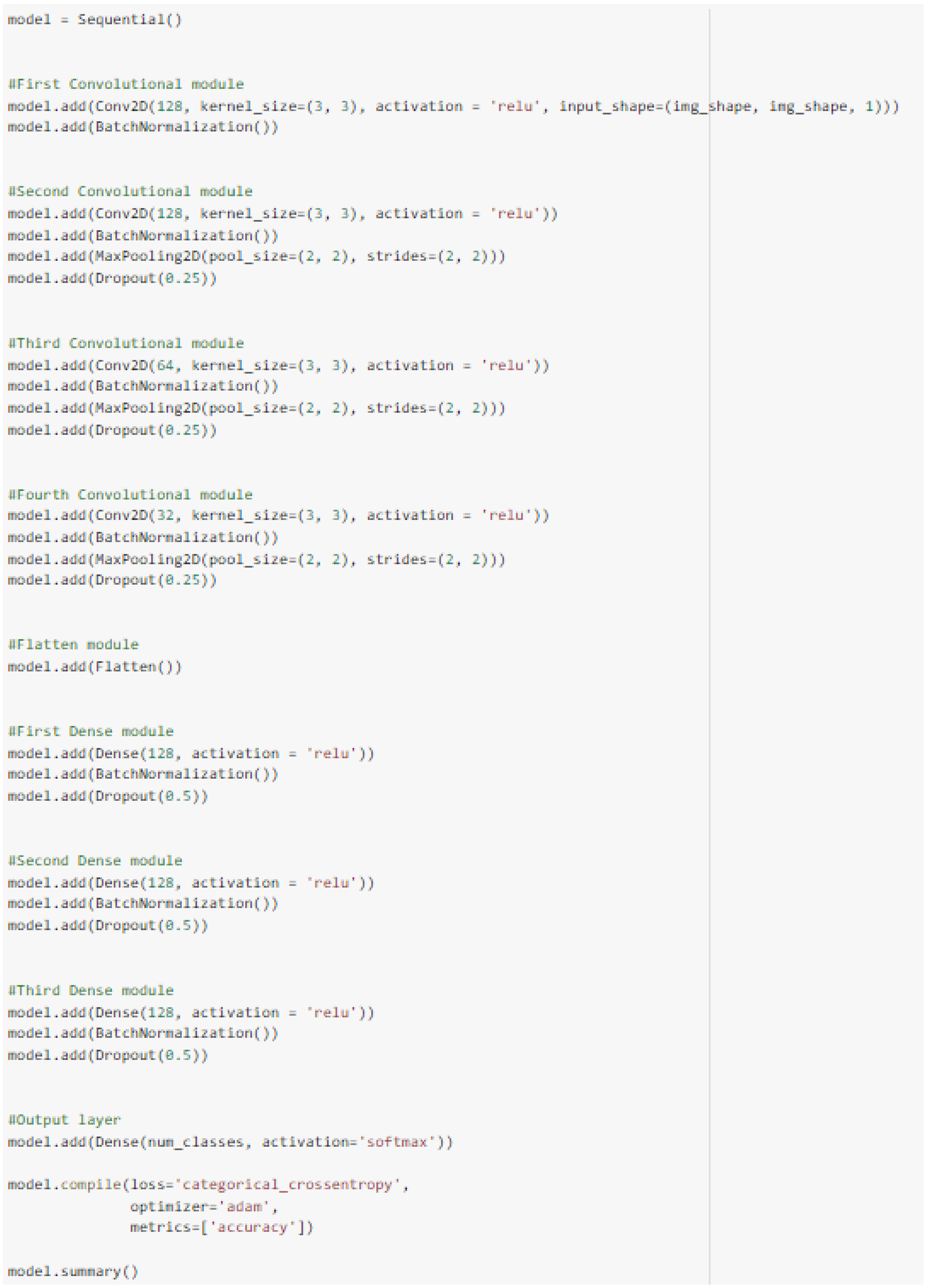
Figure 2.2: Shapes



```
X_ train  shape =  (28709, 48, 48, 1)
Y_ train  shape =  (28709, 7)
X_ validation  shape =  (3589, 48, 48, 1)
Y_ validation  shape =  (3589, 7)
X_ test  shape =  (3589, 48, 48, 1)
Y_ test  shape =  (3589, 7)
```

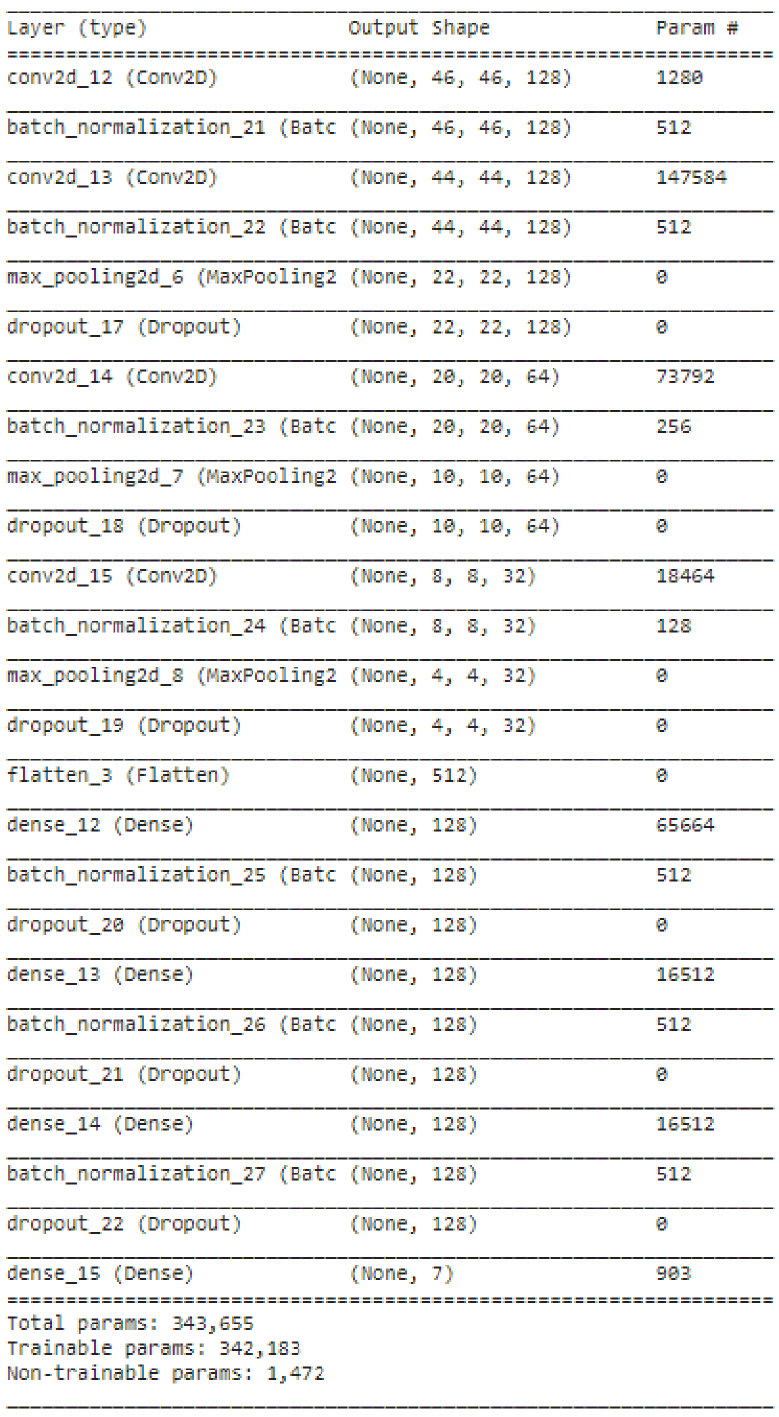
2.3 Definition of the Model

The CNN-based model that I’ve used was described clearly in the previous chapter. In the code visible right here, the different modules are visible and numbered, with their parameters and a final summary (Figure 2.3).



```
model = Sequential()


#First Convolutional module
model.add(Conv2D(128, kernel_size=(3, 3), activation = 'relu', input_shape=(img_shape, img_shape, 1)))
model.add(BatchNormalization())


#Second Convolutional module
model.add(Conv2D(128, kernel_size=(3, 3), activation = 'relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.25))


#Third Convolutional module
model.add(Conv2D(64, kernel_size=(3, 3), activation = 'relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.25))


#Fourth Convolutional module
model.add(Conv2D(32, kernel_size=(3, 3), activation = 'relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.25))


#Flatten module
model.add(Flatten())


#First Dense module
model.add(Dense(128, activation = 'relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))


#Second Dense module
model.add(Dense(128, activation = 'relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))


#Third Dense module
model.add(Dense(128, activation = 'relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))


#Output layer
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.summary()
```

Figure 2.3: Model Summary



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_12 (Conv2D)           (None, 46, 46, 128)       1280      
_________________________________________________________________
batch_normalization_21 (Batc (None, 46, 46, 128)       512       
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 44, 44, 128)       147584    
_________________________________________________________________
batch_normalization_22 (Batc (None, 44, 44, 128)       512       
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 22, 22, 128)       0         
_________________________________________________________________
dropout_17 (Dropout)         (None, 22, 22, 128)       0         
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 20, 20, 64)        73792     
_________________________________________________________________
batch_normalization_23 (Batc (None, 20, 20, 64)        256       
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 10, 10, 64)        0         
_________________________________________________________________
dropout_18 (Dropout)         (None, 10, 10, 64)        0         
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 8, 8, 32)          18464     
_________________________________________________________________
batch_normalization_24 (Batc (None, 8, 8, 32)          128       
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 4, 4, 32)          0         
_________________________________________________________________
dropout_19 (Dropout)         (None, 4, 4, 32)          0         
_________________________________________________________________
flatten_3 (Flatten)          (None, 512)               0         
_________________________________________________________________
dense_12 (Dense)             (None, 128)               65664     
_________________________________________________________________
batch_normalization_25 (Batc (None, 128)               512       
_________________________________________________________________
dropout_20 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_13 (Dense)             (None, 128)               16512     
_________________________________________________________________
batch_normalization_26 (Batc (None, 128)               512       
_________________________________________________________________
dropout_21 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_14 (Dense)             (None, 128)               16512     
_________________________________________________________________
batch_normalization_27 (Batc (None, 128)               512       
_________________________________________________________________
dropout_22 (Dropout)         (None, 128)               0         
_________________________________________________________________
dense_15 (Dense)             (None, 7)                 903       
=================================================================
Total params: 343,655
Trainable params: 342,183
Non-trainable params: 1,472
_________________________________________________________________
```

The time required for Training the model is around 25/30 minutes (around 30/32 seconds for epoch) that is, for a similar task, an acceptable performance. The various test that I performed with/without the normalization, dropout or pooling layers didn't give me much better results in term of time, but they were less precise, that can be noted especially in the confusion matrices that I will show later.

2.4 Graphs and Confusion Matrices

First of all, I plotted the Accuracy graph, including the accuracy calculated on the Training and on the Validation sets (Figure 2.4).
Then, I plotted the loss graph, on Training and on Validation sets too (Figure 2.5).
Both the graphs have in the X-axis the number of epochs (in this case 50) and in the Y-axis the accuracy and the loss.

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'validation'], loc='upper left')
plt.show()
```

Figure 2.4: Accuracy Graph

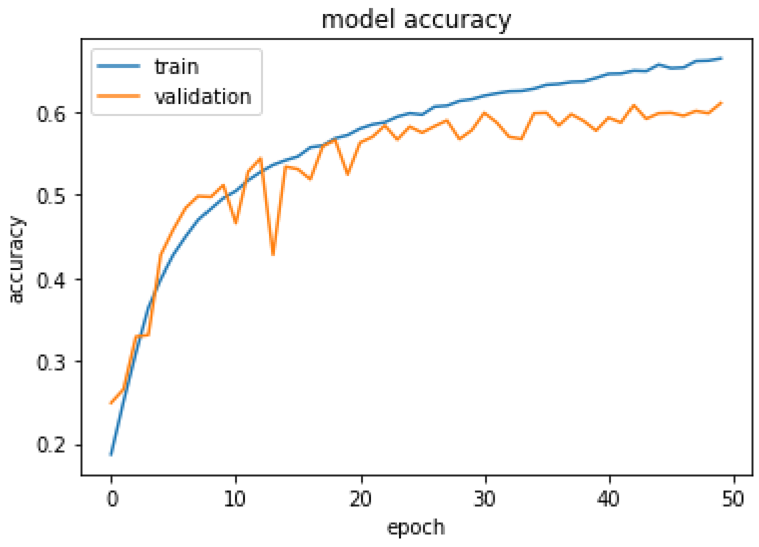
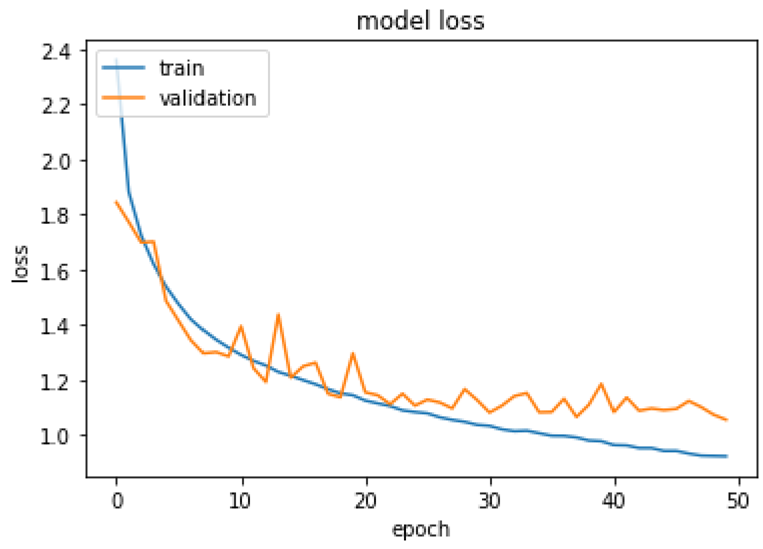


Figure 2.5: Loss Graph



From Figure 2.4, we can see that the model doesn't overfit, as the difference between the train set accuracy curve and the validation set one is not too much high. We can also see that the train reach approximately 0.66 for the training set and 0.61 for the validation one, that is a good result for our task.

From Figure 2.5, we can notice that the loss is correctly decreasing in both cases. Its lower value is around 0.9 for the train set and 1.0 for the validation set, that is acceptable for the kind of task that we are performing.

Now, I evaluated the model on the test set, to see the accuracy on it.

```
test_loss, test_acc = model.evaluate(X_test, Y_test)
print('Test accuracy:', test_acc)
```

```
113/113 [==============================] - 2s 14ms/step - loss: 1.0097 - accuracy: 0.6252
Test accuracy: 0.6252437829971313
```

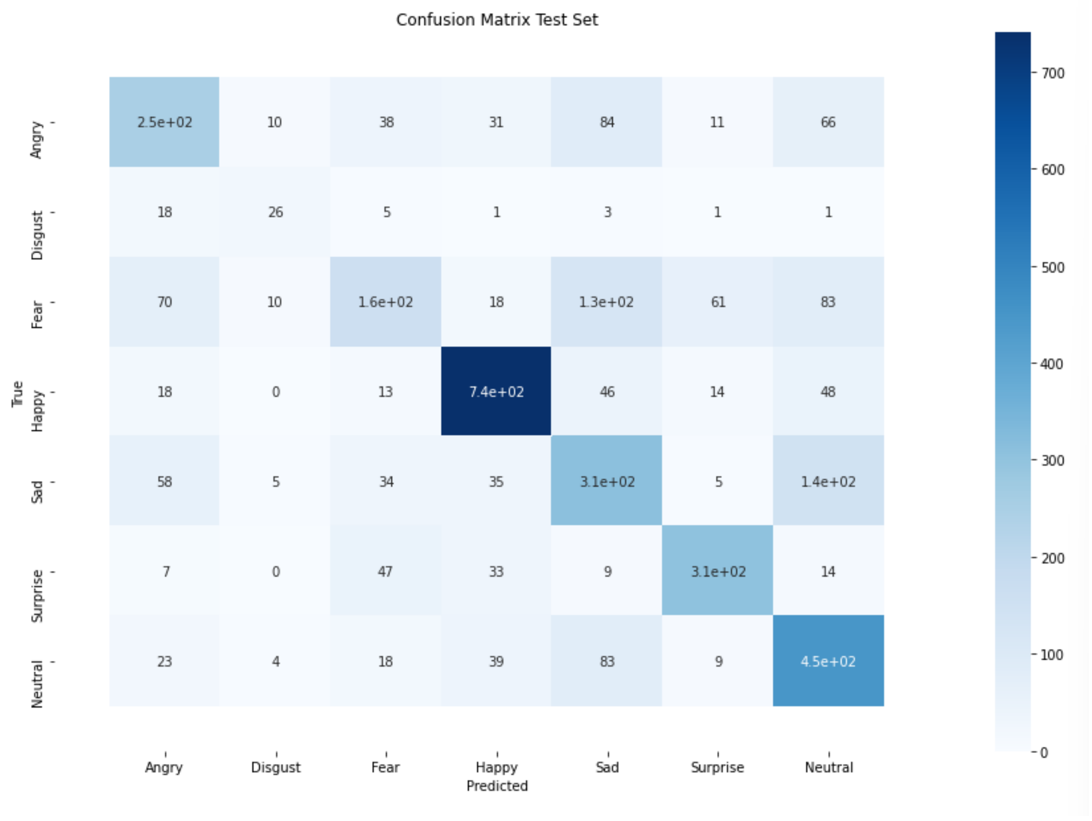
As we can see, the results are not that far from the one obtained for the training/validation set, so we can consider the model as performing.

The final proof of the model is left to the confusion matrices, that give us a simply and immediate result of the work of the model.

First of all, I showed an un-normalized matrix, to show how well the model perform (Figure 2.6).

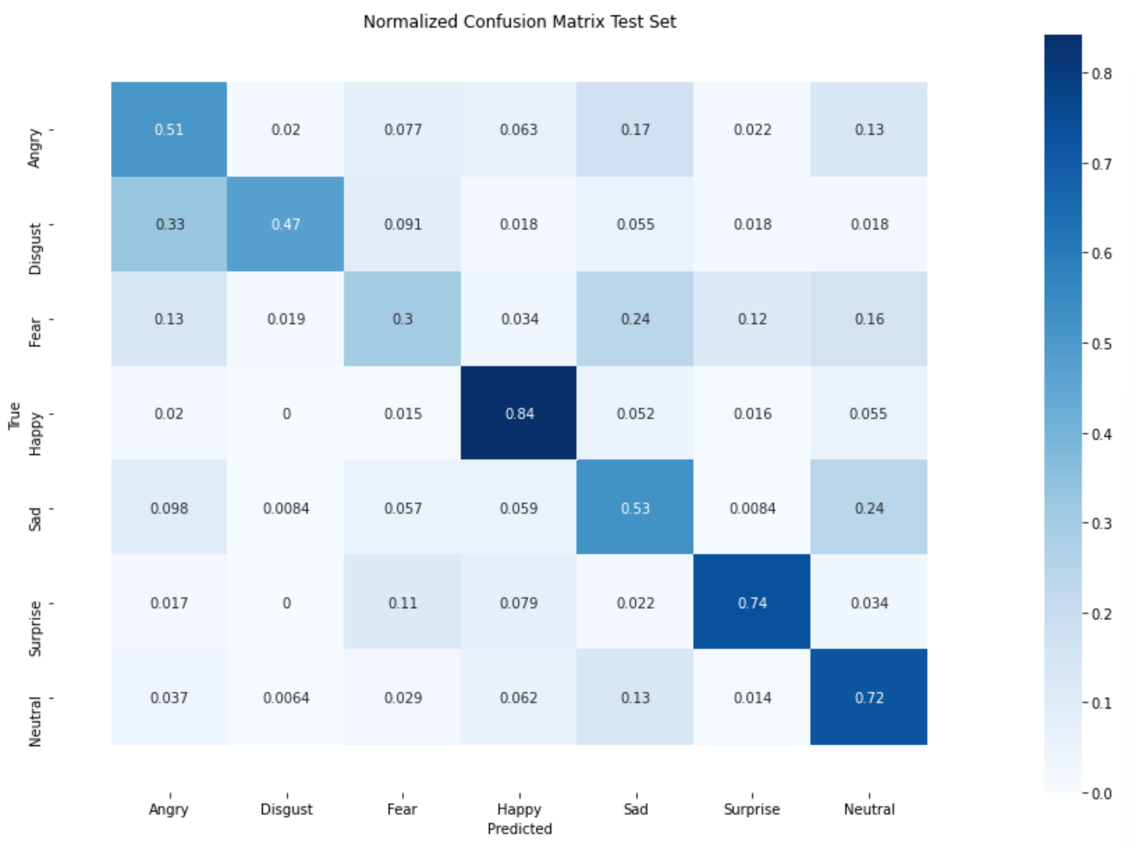
```
def print_confusion_matrix_pandas(model, images, labels):
    num_classes = 7
    # Get the predicted classifications for the test-set.
    predictions = model.predict(images)
    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(np.argmax(labels, axis = 1), np.argmax(predictions,axis=1))
    # Plot the confusion matrix as an image.
    class_names = labels_confusion_matrix.keys()
    df_cm = pd.DataFrame(cm, index = class_names, columns = class_names)
    plt.figure(figsize = (15,10))
    sn.heatmap(df_cm, annot=True, cmap='Blues')
    plt.axis([-0.5, 7.5, 7.5, -0.5])
    plt.title('Confusion Matrix Test Set')
    plt.xlabel('Predicted')
    plt.ylabel('True')
    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

```
print_confusion_matrix_pandas(model, X_test, Y_test)
```



```python
def print_normalized_confusion_matrix_pandas(model, images, labels):
    num_classes = 7
    # Get the predicted classifications for the test-set.
    predictions = model.predict(images)
    # Get the confusion matrix using sklearn.
    cm = confusion_matrix(np.argmax(labels, axis = 1), np.argmax(predictions,axis=1))
    cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
    # Plot the confusion matrix as an image.
    class_names = labels_confusion_matrix.keys()
    df_cm = pd.DataFrame(cm, index = class_names, columns = class_names)
    plt.figure(figsize = (15,10))
    sn.heatmap(df_cm, annot=True, cmap='Blues')
    plt.axis([-0.5, 7.5, 7.5, -0.5])
    plt.title('Normalized Confusion Matrix Test Set')
    plt.xlabel('Predicted')
    plt.ylabel('True')
    # Ensure the plot is shown correctly with multiple plots
    # in a single Notebook cell.
    plt.show()
```

```python
print_normalized_confusion_matrix_pandas(model, X_test, Y_test)
```



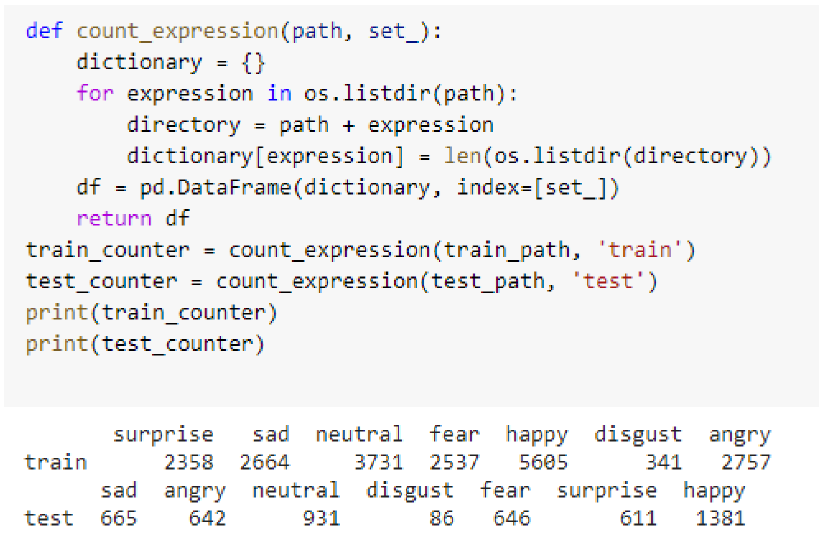
Chapter 3

Extra

In this Chapter, I'm going to present the Extra trial required, that consist in using the same fer2013 dataset, but with masked faces.

3.1 Masked Emotion Recognition

To mask the faces, I have used the software recommended in the instructions, called MaskTheFace (https://github.com/aqeelanwar/ MaskTheFace).
In this case, the pre-processing of data was different from the previous case, as in this case I have to work directly with the images. The software used has some defects. In fact, it reduced the dataset of almost 30% (25449 examples instead of 35887), because of some images that could not being processed in order to put a mask (for example, some images that have hands on the face).
This is the total number of images for class (emotion):

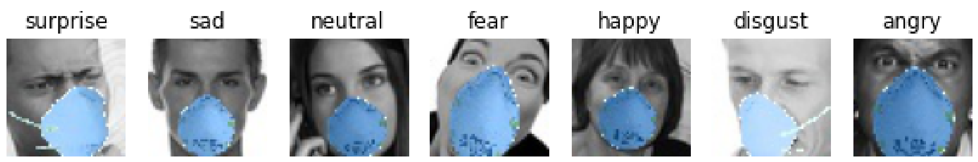


```python
def count_expression(path, set_):
    dictionary = {}
    for expression in os.listdir(path):
        directory = path + expression
        dictionary[expression] = len(os.listdir(directory))
    df = pd.DataFrame(dictionary, index=[set_])
    return df
train_counter = count_expression(train_path, 'train')
test_counter = count_expression(test_path, 'test')
print(train_counter)
print(test_counter)
```

```
       surprise   sad  neutral  fear  happy  disgust  angry
train      2358  2664     3731  2537   5605      341   2757
      sad  angry  neutral  disgust  fear  surprise  happy
test  665    642      931       86   646       611   1381
```

Now, I have showed some examples of the masked dataset, as done with the normal one.

```python
plt.figure(figsize=(10,10))
d = 1
for i in os.listdir(train_path):
    img = load_img((train_path + i +'/'+ os.listdir(train_path + i)[1]))
    plt.subplot(1,7,d)
    plt.imshow(img)
    plt.title(i)
    plt.axis('off')
    d += 1
plt.show()
```



The pre-processing starts by using ImageDataGenerator, that generates batches of tensor images data with real-time data augmentation. With Data Augmentation, we refer to a technique of creating new data from existing one, by applying some transformations (like rotation, zooms etc.).

```python
train_datagen = ImageDataGenerator(rescale=1./255,horizontal_flip=True)

train_set = train_datagen.flow_from_directory(train_path, batch_size=64, target_size=(48,48), color_mode='grayscale',class_mode='categorical')

test_datagen = ImageDataGenerator(rescale=1./255)
test_set = test_datagen.flow_from_directory(test_path, batch_size=64, target_size=(48,48), color_mode='grayscale', class_mode='categorical')
```

```
Found 19993 images belonging to 7 classes.
Found 4962 images belonging to 7 classes.
```

To have a fair comparison between the trials, I have used the same CNN-based model as before. The time required for the training is higher, in particular for the first epoch.

As predictable, the accuracy (both on the training and on the validation/test set) is much lower. These are the graphs of the Accuracy (Figure 3.1) and of the Loss (Figure 3.2) of the model:

Figure 3.1: Accuracy Graph

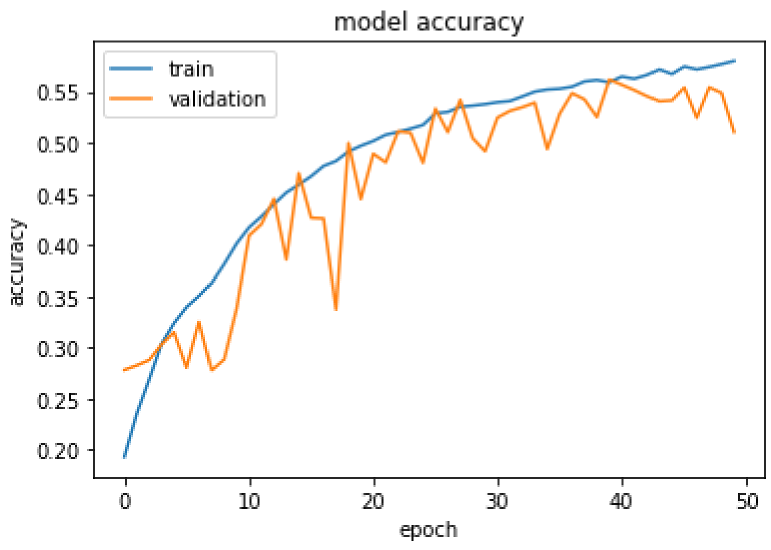
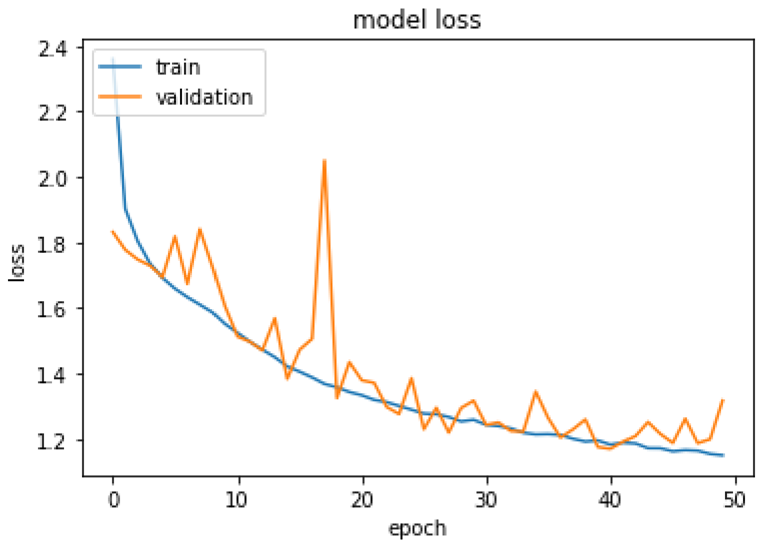


Figure 3.2: Loss Graph

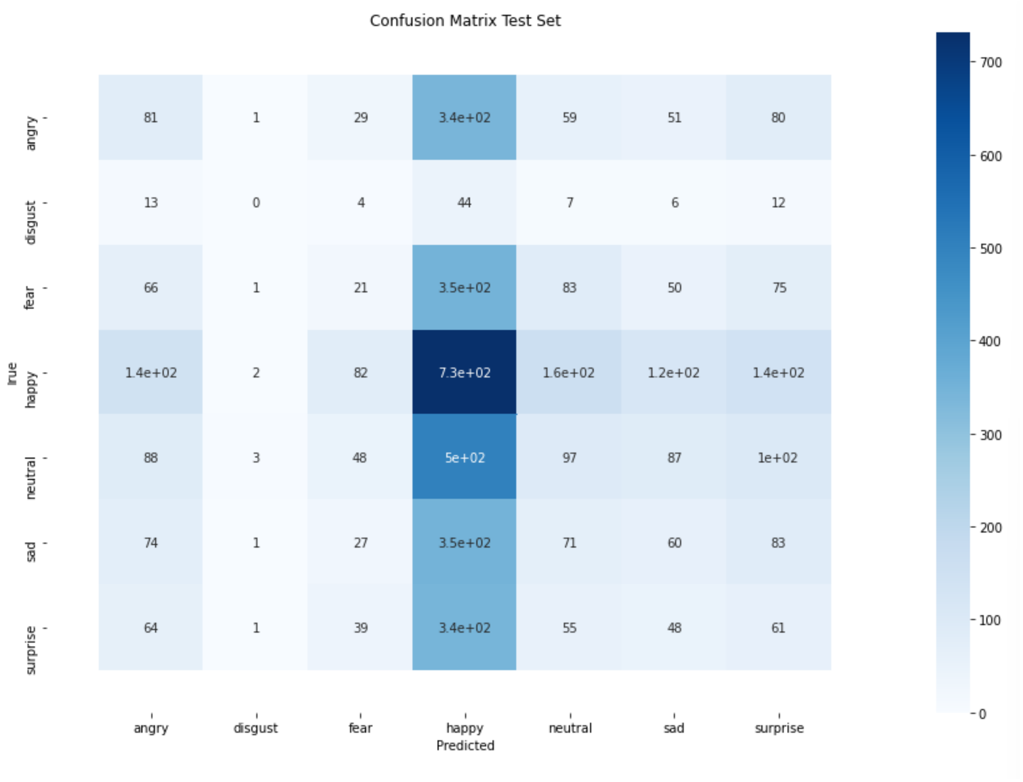


```
test_loss, test_acc = model.evaluate(test_set)
print('Test accuracy:', test_acc)
```

```
78/78 [==============================] - 9s 114ms/step - loss: 1.3172 - accuracy: 0.5107
Test accuracy: 0.51068115234375
```

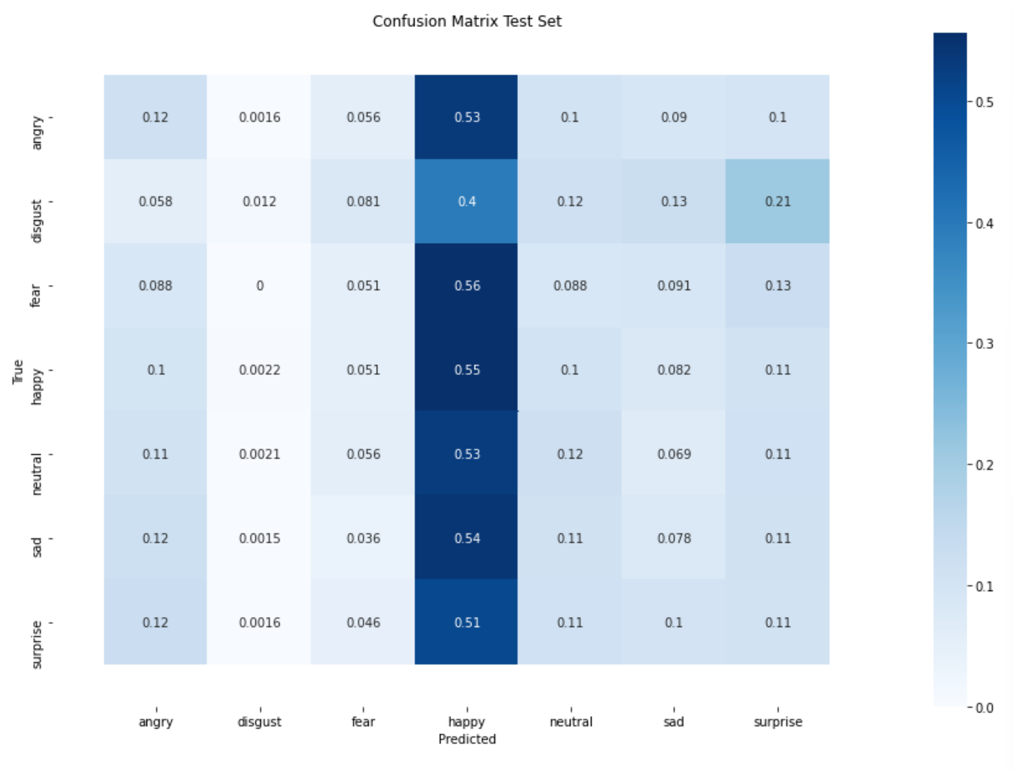
As predictable, the accuracy is lower than the first model, same for the loss.

The final proof is done with the Confusion matrices, that shown the classification errors made by the algorithm.


Confusion Matrix Test Set

True \ Predicted	angry	disgust	fear	happy	neutral	sad	surprise
angry	81	1	29	3.4e+02	59	51	80
disgust	13	0	4	44	7	6	12
fear	66	1	21	3.5e+02	83	50	75
happy	1.4e+02	2	82	7.3e+02	1.6e+02	1.2e+02	1.4e+02
neutral	88	3	48	5e+02	97	87	1e+02
sad	74	1	27	3.5e+02	71	60	83
surprise	64	1	39	3.4e+02	55	48	61

True
Predicted
700
600
500
400
300
200
100
0


Confusion Matrix Test Set

True \ Predicted	angry	disgust	fear	happy	neutral	sad	surprise
angry	0.12	0.0016	0.056	0.53	0.1	0.09	0.1
disgust	0.058	0.012	0.081	0.4	0.12	0.13	0.21
fear	0.088	0	0.051	0.56	0.088	0.091	0.13
happy	0.1	0.0022	0.051	0.55	0.1	0.082	0.11
neutral	0.11	0.0021	0.056	0.53	0.12	0.069	0.11
sad	0.12	0.0015	0.036	0.54	0.11	0.078	0.11
surprise	0.12	0.0016	0.046	0.51	0.11	0.1	0.11

True
Predicted
0.5
0.4
0.3
0.2
0.1
0.0

Chapter 4

Results

In this chapter, I will discuss the obtained results, giving in particular a focus on the confusion matrices.

4.1 Results Discussion

In figure 2.4 and 2.5, I have shown the accuracy and loss graph. As discussed before, the results are not excellent (accuracy around 0.66 and loss around 0.92 for the training set), but the task was not that easy, as most of the emotion are difficult to label even for humans. By taking as reference the "Challenge in Representation Learning: Facial Expression Recognition Challenge" [1], with an accuracy on the test set of around 0.625, the approach that I used will classify in the first 10 places.

From these graphs, we can notice that the model is trained in a good way, as it doesn't overfit over training data (really important in order to obtain a good model) and they don't "saturate" on some values, but the curves are good, both in increasing and in decreasing.

[1]: https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/leaderboard

By observing the confusion matrices, we could notice more "practical" aspects and defects about the algorithm.

Now, focusing on the Normalized matrix (Figure 2.7), we could notice that, in some cases the emotions are well and clearly labelled, in other there's some indecision.

For example, "Happy" is correctly labelled in most of the cases (0.84), similar for "Surprise" (0.74) and "Neutral" (0.72).

Then, there are some cases in which the emotion is correctly labelled in most of the cases (>0.5), but not as accurate as before. This is the case of "Angry" (0.51) and "Sad" (0.53). This last one, in particular, has a conflict with Neutral (0.24), that is not negligible.

Finally, we have the more undecided case. These are "Disgust", that is correctly recognized with 0.47, but it has a possibility of being misclassified with Angry of 0.33, that is not negligible. In particular, we could see the difficulty of recognizing this emotion also in the Un-normalized matrix, where the numbers are lower than the other cases (it can also depend on an asymmetry of the dataset, in fact the number of "Disgust" examples in the training set are much less than the others).

The worst case is "Fear". In this case, the emotion is correctly recognized only with the 0.3, but the algorithm can misclassify it with different other emotions, like "Sad" (0.24), "Neutral" (0.16), "Fear" (0.13) and "Surprise" (0.12).

I think that the reasons of these results are traceable also in humans. For example, there are some emotions that can be expressed differently from people to people. While it's almost easy for us to recognize a happy face, it's tough sometimes to distinguish between fear or surprise, or between angry and surprised.

These defects were highlighted in every trial that I made, with all the different CNN-based architectures that I have implemented.

Finally, the masked trial shows the difficult in recognize emotions with mask on. The results were predictable because, without seeing half of the face (and in particular the area between nose and mouth), it becomes difficult even for humans to recognize the possible emotion expressed.

Other than that, as said in the introduction of the trial, the dataset was cut by almost the 30%, that affect a little the total accuracy of the model.

From the analysis of the Confusion matrices, it is visible that the algorithm tends to classify the undecided images as happy one, probably because is the emotion that includes more different physical traits.