UNIVERSITÀ DI PISA

Department of Information Engineering
MSc in Artificial Intelligence and Data Engineering

Emotion detection from images

Computational Intelligence and Deep Learning Project

Irene Cantini
Ludovica Cocchella

Academic Year 2022/2023

Contents

1 Introduction

1.1 Project description

Emotion recognition from facial images is an increasingly significant task across various domains. The objective of this paper is to tackle the emotion recognition classification problem using facial images. Deep Learning methodologies like Transfer learning and custom-trained Neural Networks, are employed to address this task. Subsequently, the paper compares the outcomes of these techniques. The classifier aims to distinguish the following emotions: happiness, neutrality, sadness, anger, surprise, disgust, and fear. Emotion Detection using images is quite useful for identification. Examples of situations where we can use this type of technology are driver's drowsiness detection, students behavior detection or marketing, in fact companies can employ these neural networks to assess consumers' emotional responses to advertising campaigns, products, or services.

2 Dataset

The dataeset used to develop this project is taken by Kaggle. It's details are:

- Source: https://www.kaggle.com/datasets/msambare/fer2013
- Size: 28,709 examples
- Image Type: 48x48 pixel grayscale images of human faces
- Classes: 7 emotion classes (Angry, Disgust, Fear, Happy, Sad, Surprise, Neutral)
- Total Size: 56.51 MB

The dataset will be divided as follows:

- **Training Set**: This subset will be used to train our emotion detection models. It will consist of a portion of the combined dataset.
- **Validation Set**: The validation set will be employed to fine-tune the model during training. It will aid in selecting hyperparameters and preventing overfitting.
- **Test Set**: The final performance evaluation of our models will be carried out on this subset.

In the following, one example for each emotion, is reported.


Angry


Happy


Sad


fear


Disgust

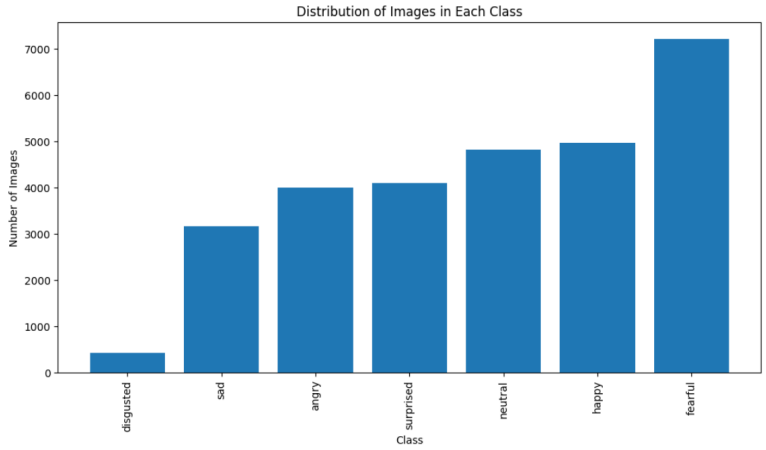

Neutral

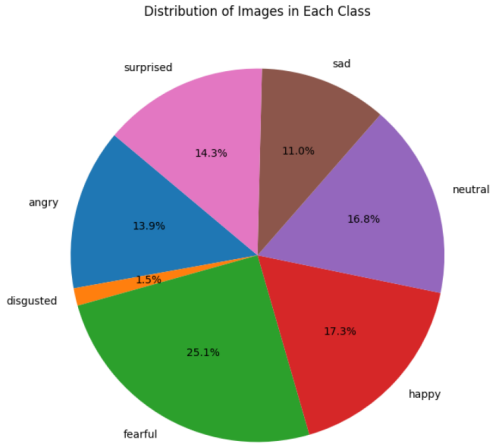

Surprise

Figure 1: Emotions

2.1 Prepare the dataset

The images are organized in two principal folders that are train and test. Then there are, in each of them, other 7 different folders each one related to a class and containing the corresponding images. The classes have been named as 0, 1, 2, 3, 4, 5, 6, corresponding to the following emotions: angry, disgusted, fearful, happy, neutral, sad, surprised. The initial distribution of the train set is:



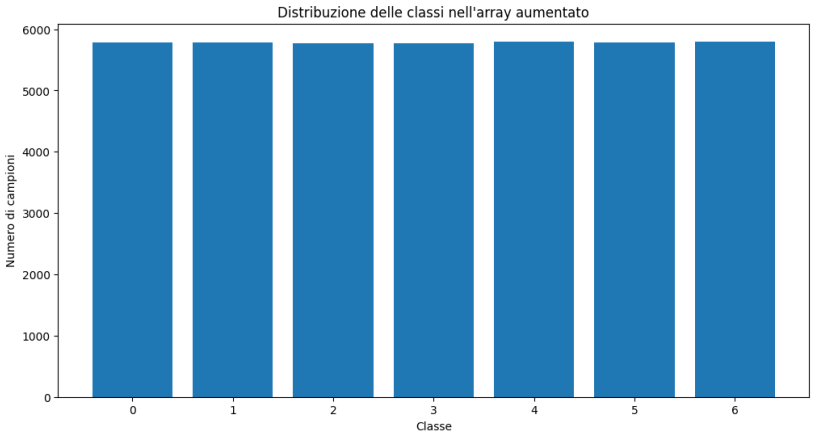


As we can see from the previous images, we have an imbalanced dataset. So, we have decided to use two different approaches for training our networks and then evaluate which one performs better. The first approach involves not making any changes to the dataset and addressing data imbalance by using class weight during the training phase, while the second approach involves balancing the dataset by performing data augmentation on the minority classes.

To prepare the dataset for the first approach, in order to use it for training the CNN, we read images from each class and store them in a NumPy array along with

their respective labels. This is a typical step in data preparation for training machine learning models, where images and labels are prepared for training a neural network. After the construction of this array we created the train and validation sets and to do this we mixed the arrays to avoid a class predominance when creating the train and validation sets. In this way all the classes have the same probability to be present in the two sets. We used a split ratio of 80-20. We need a validation set because the primary goal of machine learning is to develop models that can generalize well to unseen data. The validation set is used to assess how well the model learned during training in order to generalize data it has never seen before. This helps prevent overfitting, where the model memorizes the training set instead of learning the true underlying patterns in the data. After that we divided the pixel values by 255 to normalize them. In our image data, pixel values range from 0 to 255, where 0 represents black and 255 represents white. By dividing all pixel values by 255, we scale the values to a range between 0 and 1. Normalization is crucial because it ensures that all input features (pixel values in this case) have similar scales. Neural networks tend to perform better when the input data is in a consistent range. Since we are dealing with a classification problem, converting labels to categorical format is necessary. In particular we converted integer class labels into one-hot encoded categorical labels.

The dataset preparation for the second approach closely aligns with the procedures outlined for the first approach, except for one key distinction: after dividing the dataset into training and validation sets, we exclusively applied data augmentation to the minority classes within the training dataset. This was carried out to ensure an equal number of images for each class. The distribution obtained is the following:



3 Experiments using class weight

3.1 CNN from Scratch

In this section a CNN (*Convolutional Neural Network*) is built to construct a classifier able to recognize facial emotion starting from the dataset described in the section above. The CNN is used because it is a deep network that finds out what are the best features that are necessary and sufficient to solve a problem. It is developed from scratch to implement all the necessary components without pre-existing deep learning frameworks or libraries. It is developed using Google Collab and Python as programming language.

3.1.1 CNN Structure

To find out the best network which satisfies the facial emotion classification problem we started building different CNN networks from the smallest one (with 3 convolutional layers) to the most complex. We added at each experiment a single layer of the network analysing the final accuracy obtained. All the CNNs are built using the Keras API with TensorFlow as backend. From the different trials we achieved the following results:

Table 1: Experiments results

N	Network	Accuracy
1	3 convolutional block	0.55
2	4 convolutional block	0.59
3	5 convolutional block	0.58
4	4 convolutional block (first composed by two layers)	0.57
5	4 convolutional block (first/second composed by two layers)	0.64
6	4 convolutional block (first/second/third composed by two layers)	0.62
7	4 convolutional block (first/second/third/fourth composed by two layers)	0.61
8	4 convolutional block (first composed by three layers, second/third/fourth composed by two layers)	0.63

After these results we decided to use the network obtained in the fifth experiment because it has the highest accuracy with a simpler structure than the last one. It's interesting to note that the network with the best accuracy wasn't one of the most complex ones that were trained, showing that a larger network doesn't always lead to better results, and it's more important to calibrate the parameters well.

3.1.2 CNN Construction

Below the code used to build the network is shown:

```
from tensorflow.keras import layers
from tensorflow.keras import models
from keras.layers import BatchNormalization
from keras.layers import Dropout
from tensorflow import keras

# Define the CNN model

model = models.Sequential()

# First convolutional layer
model.add(layers.Conv2D(32, (3, 3), ...
    activation='relu',padding='same', input_shape=(48, 48, 1)))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(32, (3, 3), activation='relu', ...
    padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.2))

# Third convolutional layer
model.add(layers.Conv2D(64, (3, 3), activation='relu', ...
    padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(64, (3, 3), activation='relu', ...
    padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.2))

# fifth convolutional layer
model.add(layers.Conv2D(128, (3, 3), activation='relu', ...
    padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.2))

# seventh convolutional layer
model.add(layers.Conv2D(256, (3, 3), activation='relu', ...
    padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.2))

# Flattening Layer
model.add(layers.Flatten())

# first dense layer
model.add(layers.Dense(256, activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.5))
```

```
# third dense layer (output)
model.add(layers.Dense(7, activation='softmax'))

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

The summary of the model:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 48, 48, 32)        320

 batch_normalization (BatchN  (None, 48, 48, 32)       128
 ormalization)

 conv2d_1 (Conv2D)           (None, 48, 48, 32)        9248

 max_pooling2d (MaxPooling2D  (None, 24, 24, 32)       0
 )

 batch_normalization_1 (Batc  (None, 24, 24, 32)       128
 hNormalization)

 dropout (Dropout)           (None, 24, 24, 32)        0

 conv2d_2 (Conv2D)           (None, 24, 24, 64)        18496

 batch_normalization_2 (Batc  (None, 24, 24, 64)       256
 hNormalization)

 conv2d_3 (Conv2D)           (None, 24, 24, 64)        36928

 max_pooling2d_1 (MaxPooling  (None, 12, 12, 64)       0
 2D)

 batch_normalization_3 (Batc  (None, 12, 12, 64)       256
 hNormalization)

 dropout_1 (Dropout)         (None, 12, 12, 64)        0

 conv2d_4 (Conv2D)           (None, 12, 12, 128)       73856

 max_pooling2d_2 (MaxPooling  (None, 6, 6, 128)        0
 2D)

 batch_normalization_4 (Batc  (None, 6, 6, 128)        512
 hNormalization)

 dropout_2 (Dropout)         (None, 6, 6, 128)         0
```

```
conv2d_5 (Conv2D)           (None, 6, 6, 256)         295168

max_pooling2d_3 (MaxPooling  (None, 3, 3, 256)        0
2D)

batch_normalization_5 (Batc  (None, 3, 3, 256)        1024
hNormalization)

dropout_3 (Dropout)         (None, 3, 3, 256)         0

flatten (Flatten)           (None, 2304)              0

dense (Dense)               (None, 256)               590080

batch_normalization_6 (Batc  (None, 256)              1024
hNormalization)

dropout_4 (Dropout)         (None, 256)               0

dense_1 (Dense)             (None, 7)                 1799

=================================================================
Total params: 1,029,223
Trainable params: 1,027,559
Non-trainable params: 1,664
_________________________________________________________________
```

The model's structure is constructed sequentially, step by step, with various convolutional, batch normalization, and fully connected layers to learn to extract meaningful features from input images and classify them accurately. Below is a detailed overview of the process:

- Importing Libraries: The necessary libraries are imported, including modules from TensorFlow Keras for model definition and layer creation.
- Model Definition: An empty sequential model is created, allowing layers to be added one after another in sequence.
- Convolutional Layers: Multiple convolutional layers are added, each followed by a ReLU activation to introduce non-linearity. These layers progressively extract information from the input data. The use of 'same' padding maintains dimensions so that the output retains the same dimensions as the input. Batch normalization is also applied to enhance training stability. In each layer a number of **filters** is defined and their quantity is increased from 32 to 256. Each one has a dimension of 3x3 and convolves the input image and form a **Feature map**.
- Max Pooling: After each pair of convolutional layers, max pooling is performed to reduce the dimension of the extracted feature maps while preserving salient information.

- Dropout: Regularization using dropout is applied to reduce overfitting. This prevents the network from becoming overly dependent on the training data.
- Flattening Layer: The output of the final convolutional layer is "flattened" into a one-dimensional vector, preparing it for input to the fully connected layers.
- Densely Connected (Dense) Layers: Fully connected layers are added. Each layer has a ReLU activation function to introduce non-linearity. Batch normalization and dropout are used to mitigate overfitting.
- Output Layer: The last layer is a fully connected layer with a softmax activation function. This layer performs the classification of images into one of the seven possible categories.
- Model Compilation: The model is compiled by specifying the "adam" optimizer, "categorical crossentropy" loss function (suitable for multi-class classification), and evaluation metrics such as accuracy.
- Model Summary: A detailed summary of the model is printed, displaying the number of parameters and tensor dimensions in each of the layers.

3.1.3 CNN Training

In the next step we train the network thanks to *model.fit()*, a method in TensorFlow Keras. This method handles the entire training process, including iterating through training epochs and calculating losses and evaluation metrics on training and validation data. We decided to set an initial number of epoch equal to 200 in order to allow the network to train until the early_stopping condition stops it.

The code used is shown below:

```
# Compute class weights
class_weights = class_weight.compute_class_weight(
    class_weight='balanced',
    classes=np.unique(label_train),
    y=label_train)
class_weight_dict = dict(enumerate(class_weights))

# Create ImageDataGenerator object with data augmentation
datagen = ImageDataGenerator(
    rotation_range=10,
    zoom_range=0.15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.15,
    horizontal_flip=False,
    vertical_flip=False,
    fill_mode="nearest")

# Create an instance of EarlyStopping callback
early_stopping = EarlyStopping(
```

```
    monitor='val_loss',     # Monitor validation loss
    patience=10,              # Stop if no improvement after 10 epochs
    restore_best_weights=True  # Restore the model with the best ...
        weights
)

checkpoint = ...
    tf.keras.callbacks.ModelCheckpoint('/content/drive/Shareddrives
            /Cocchella_Cantini/models/CnnModelFromScratch.h5',
                        monitor='val_accuracy',
                        save_best_only=True)

# Train the model with data augmentation and class weights
history = model.fit(
    datagen.flow(img_train, categorical_label_train, batch_size=64),
    epochs=200,
    callbacks=[early_stopping, checkpoint],
    validation_data=(img_val, categorical_label_val),
    class_weight=class_weight_dict,
    shuffle=True)
```

During training, the model.fit() method performs the following operations:

- For each epoch, it divides the training data into batches of defined size.
- For each batch, it calculates gradients of the loss function with respect to the model's weights.
- It applies weight updates using optimization algorithms like the "adam" optimizer.
- It calculates specified metrics (e.g., accuracy) on both training and validation data to monitor performance.

In addition to the basic training steps, in this case model.fit() have to incorporate the following features:

- Data Augmentation: Data augmentation involves applying random transformations to training images, such as rotations, flips, and shifts. This technique increases the diversity of the training data, reducing overfitting and improving generalization to new data.
- Early Stopping: The EarlyStopping callback monitors a specified metric (e.g., validation loss) during training and halts training if the metric stops improving. This prevents overfitting and allows stopping when the model's performance doesn't improve anymore.
- Class Weights: Imbalanced datasets can lead to biased training. The class_weight parameter allows to assign higher weights to underrepresented classes during training, giving them more influence on weight updates and balancing the training process.

The results obtained by the last 10 epochs are reported below:

```
Epoch 55/200
359/359 [==============================] - 14s 38ms/step - loss: ...
    0.9209 - accuracy: 0.6181 - val_loss: 0.9936 - val_accuracy: 0.6388
Epoch 56/200
359/359 [==============================] - 14s 38ms/step - loss: ...
    0.9163 - accuracy: 0.6208 - val_loss: 1.0050 - val_accuracy: 0.6266
Epoch 57/200
359/359 [==============================] - 14s 39ms/step - loss: ...
    0.9085 - accuracy: 0.6257 - val_loss: 0.9971 - val_accuracy: 0.6320
Epoch 58/200
359/359 [==============================] - 14s 39ms/step - loss: ...
    0.9135 - accuracy: 0.6254 -val_loss: 1.0048 - val_accuracy: 0.6210
Epoch 59/200
359/359 [==============================] - 14s 38ms/step - loss: ...
    0.9157 - accuracy: 0.6250 - val_loss: 1.0050 - val_accuracy: 0.6310
Epoch 60/200
359/359 [==============================] - 14s 39ms/step - loss: ...
    0.9007 - accuracy: 0.6291 - val_loss: 1.0379 - val_accuracy: 0.6209
Epoch 61/200
359/359 [==============================] - 14s 38ms/step - loss: ...
    0.9028 - accuracy: 0.6318 - val_loss: 1.0112 - val_accuracy: 0.6282
Epoch 62/200
359/359 [==============================] - 14s 39ms/step - loss: ...
    0.9057 - accuracy: 0.6267 - val_loss: 1.0682 - val_accuracy: 0.6104
Epoch 63/200
359/359 [==============================] - 14s 39ms/step - loss: ...
    0.8925 - accuracy: 0.6345 - val_loss: 1.0048 - val_accuracy: 0.6301
Epoch 64/200
359/359 [==============================] - 16s 43ms/step - loss: ...
    0.9098 - accuracy: 0.6316 - val_loss: 1.0064 - val_accuracy: 0.6308
Epoch 65/200
359/359 [==============================] - 14s 40ms/step - loss: ...
    0.9040 - accuracy: 0.6348 -val_loss: 1.0244 - val_accuracy: 0.6191
```

3.1.4 CNN Results

After the training of the model we obtained the results that we can see in the plots. Each emotion is identified by a number following this legend:

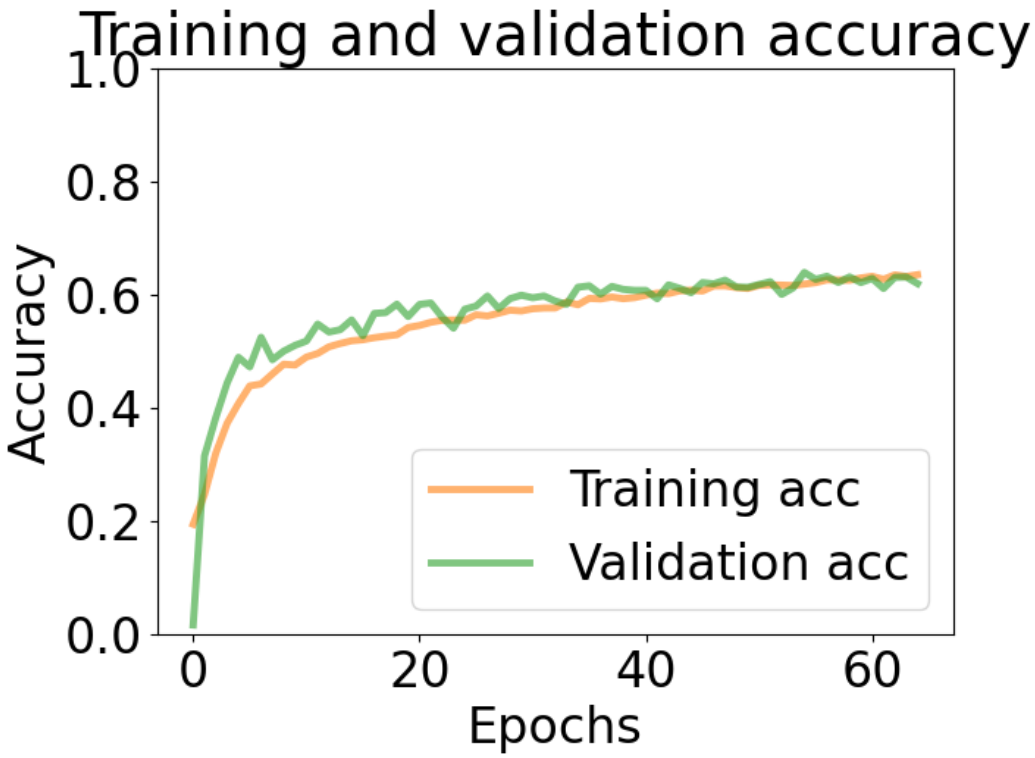
- 0: 'angry',
- 1: 'disgusted',
- 2: 'fearful',
- 3: 'happy',
- 4: 'neutral',
- 5: 'sad',

- 6: 'surprised'

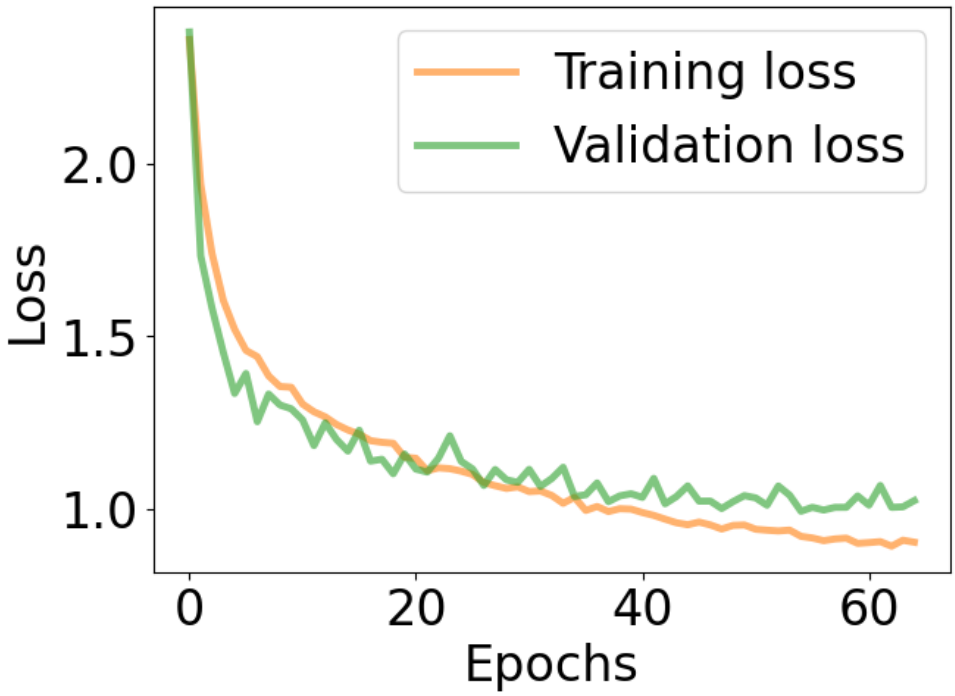
The value obtained for each metric are:

	precision	recall	f1-score	support
0	0.57	0.56	0.56	958
1	0.47	0.76	0.58	111
2	0.58	0.32	0.41	1024
3	0.85	0.84	0.85	1774
4	0.51	0.75	0.61	1233
5	0.54	0.46	0.50	1247
6	0.76	0.80	0.78	831
accuracy			0.64	7178
macro avg	0.61	0.64	0.61	7178
weighted avg	0.65	0.64	0.63	7178

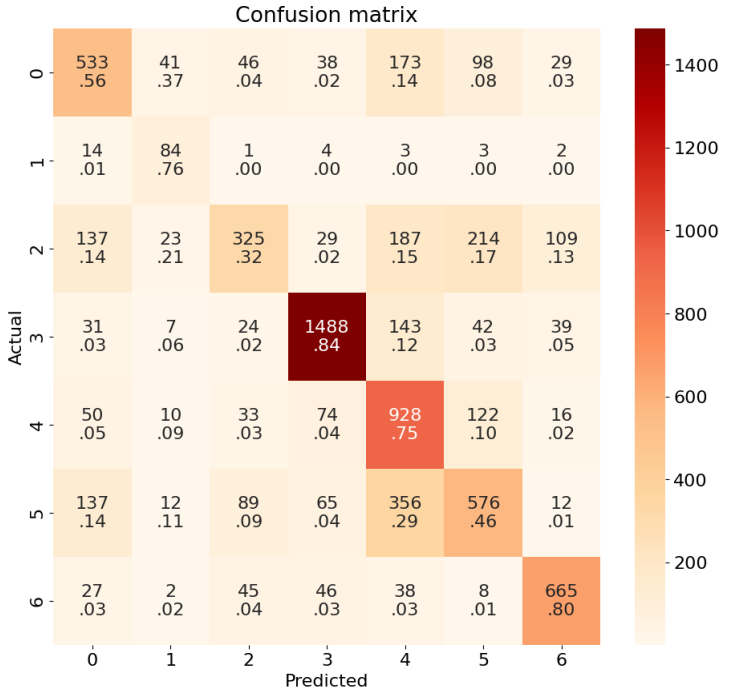
Plot of training and validation accuracy:



Plot of training and validation loss:



Confusion matrix:

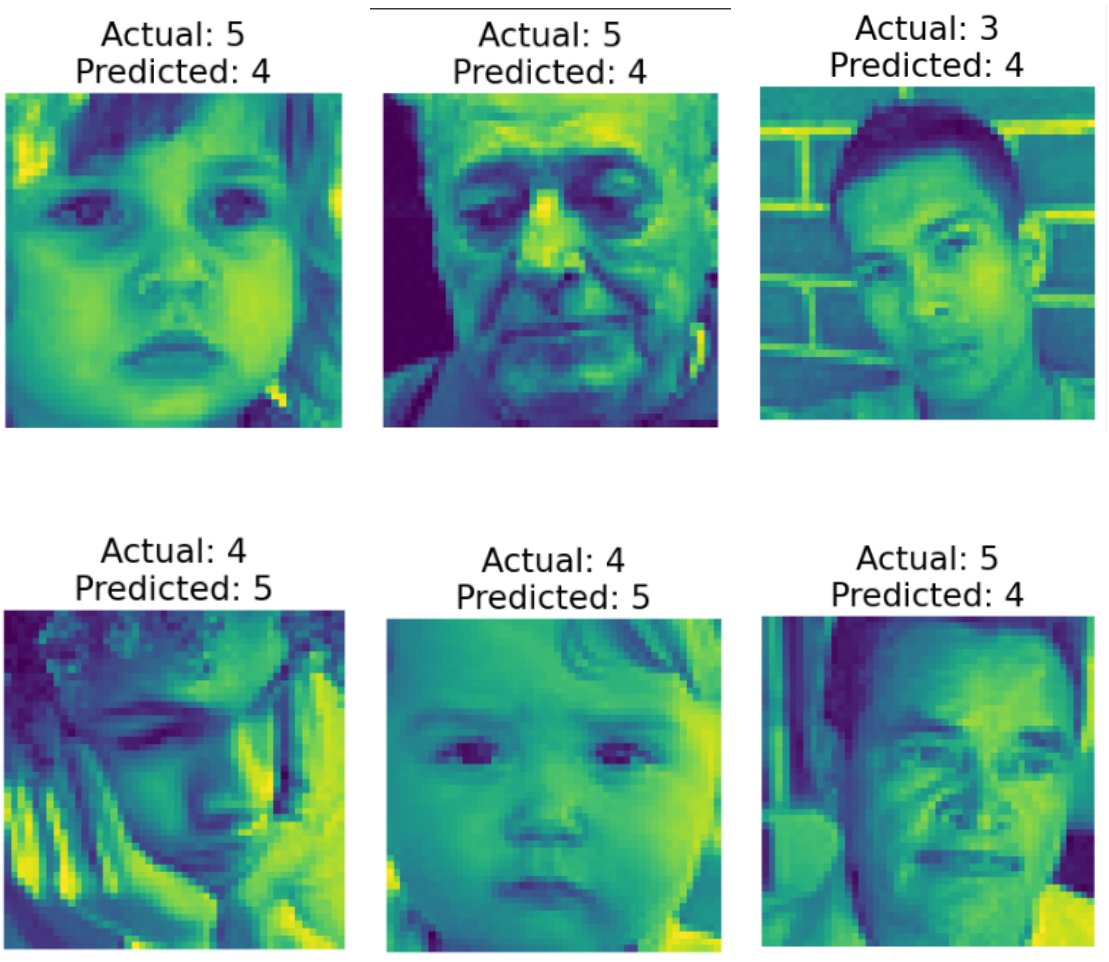


3.1.5 Consideration

Our model has achieved an accuracy of 0.64, which we can consider satisfactory for a network defined heuristically from scratch designed for a 7-class problem. To evaluate the results we obtained, we conducted a series of online searches on similar analyses carried out on our own dataset. We have seen that the highest accuracy achieved with a CNN network trained from scratch was 71.2% and the human level accuracy is only at 65±5%. This confirms that the results we have obtained are consistent for the analyzed topic.

After analyzing the test results obtained, we can observe that what poses challenges in accurate classification is the 'neutral' class. Analyzing the training dataset for the 'neutral' class, we have noticed that it includes numerous images that could potentially belong to other classes. This likely poses issues during the network's training phase, leading to prediction errors during the testing phase.

By observing the confusion matrix, we investigated the high misclassification of sad images as neutral and vice versa. Below, we present a series of examples of misclassifications related to the 'neutral' (4) and 'sad' (5) classes:



From these examples of misclassification, what we can notice is the presence of ambiguous situations between the two classes, making it difficult to objectively determine the type of emotions represented by the expressions in question. This highlights one of the central challenges of emotion detection, which is the subjective and often unclear interpretation of expressions.

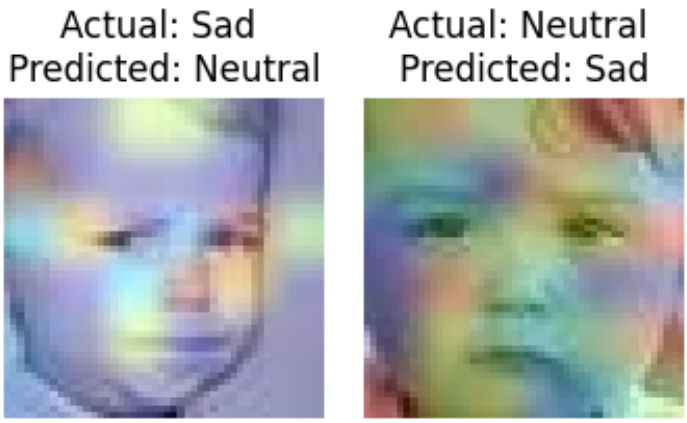
3.1.6 Explainability of the model

To better understand our network's behavior, we employed various methods. One of these methods is the visualization of heatmaps: this helps to figure out which parts of an image are identified as belonging to a given class. It helps us understand what the network relies on to classify things and what could potentially cause errors in its decisions. Below, we display some examples of heatmaps for each class:

We observed that the network has learned to concentrate its attention on specific facial features for different emotions. For 'angry' images, it focuses on the side of the mouth and the upper part of the eyes, including the eyebrows. When predicting 'disgusted,' it pays attention to the nose. 'Fearful' expressions prompt the network to consider the eyes and mouth, whereas 'happy' expressions primarily involve the mouth. 'Neutral' images draw the network's focus to both eyes, the nose, and partially the central part of the mouth. 'Sad' images lead the network to concentrate on the area below the eyes and the chin. Lastly, for 'surprised' images, the network emphasizes the cheekbones and the lower part of the forehead near the eyes.

Once we understand which features the network captures for each class, we can investigate certain misclassification scenarios between the 'neutral' and 'sad' classes

to derive more technical conclusions. Below, we present two heatmaps representing instances of misclassification:



These images clearly illustrate why our network made these two misclassifications. In the first image, the network's focus aligns with the characteristics typically associated with the 'neutral' class, while in the second image, its attention converges on regions more indicative of the 'sad' class.

3.2 Transfer Learning

This is a fundamental step for training complex CNN, specially for problem that requires a labeled training set of considerable size. In the project, described in this report, the available dataset is not of huge dimension so we decided to implement also this type of learning strategy. In order to adapt pretrained model to a different problem domain we used two techniques: features extraction and fine tuning.

3.2.1 Features extraction

Instead of training the entire neural network from scratch, we keep the pretrained model's weights frozen, excluding the final classification layers. The output of the model's layers just before the classification layers is used as feature vectors for our new dataset. We removed the top (fully connected) layers of the pretrained model and replace them with custom layers tailored to our specific task. Initially we performed this phase on three different pretrained network in order to discover which of them were the most useful for our purpose. The results obtained are reported in the following table:

Table 2: Results of feature extraction

Pretrained network	Train Accuracy	Val Accuracy	Test Accuracy
VGG16	0.3076	0.3548	0.38
VGG19	0.2750	0.3394	0.20
ResNet50	0.1381	0.1437	0.19

We noticed that all the pretrained networks have achieved unsatisfactory results but it could be caused by the fact that the dataset, with which they are trained, is very different from the data used for our problem. Despite that, VGG16, seems to have slightly better performance. Below we reported the code used.
Added Layers:

```
model.add(pretrained_model)
model.add(layers.Flatten())
model.add(layers.Dense(256)) #activation relu
model.add(Dropout(0.5))
# Output layer
model.add(layers.Dense(7, activation='softmax'))
# Compile the model
model.compile(optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy'])
model.summary()
```

Epochs during training:

```
Epoch 15/200
359/359 [==============================] - 28s 78ms/step - loss: ...
    1.7256 - accuracy: 0.3016 - val_loss: 1.6381 - val_accuracy: ...
    0.3581 - lr: 2.5000e-04
Epoch 16/200
359/359 [==============================] - 27s 75ms/step - loss: ...
    1.7270 - accuracy: 0.3026 - val_loss: 1.6638 - val_accuracy: ...
    0.3427 - lr: 2.5000e-04
Epoch 17/200
359/359 [==============================] - ETA: 0s - loss: 1.7229 ...
    - accuracy: 0.3036
Epoch 17: ReduceLROnPlateau reducing learning rate to ...
    0.0001250000059371814.
359/359 [==============================] - 27s 75ms/step - loss: ...
    1.7229 - accuracy: 0.3036 - val_loss: 1.6547 - val_accuracy: ...
    0.3520 - lr: 2.5000e-04
Epoch 18/200
359/359 [==============================] - ETA: 0s - loss: 1.7219 ...
    - accuracy: 0.3072Restoring model weights from the end of the ...
    best epoch: 11.
359/359 [==============================] - 28s 77ms/step - loss: ...
    1.7219 - accuracy: 0.3072 - val_loss: 1.6884 - val_accuracy: ...
    0.3304 - lr: 1.2500e-04
Epoch 18: early stopping
```

3.2.2 Fine Tuning

This phase involves training not only the custom layers added for our task but also some of the layers from the pretrained model. Instead of keeping all the weights frozen, we selectively unfreeze and update the weights of certain layers in the pre-trained model. We unfreeze some of the top layer of the base network while keeping the lower-level layers, which capture more abstract features, frozen. Top layers are fine-tuned to adapt to the specific characteristics of our dataset, while the lower layers retain their general feature extraction capabilities. Also in this phase we performed different tests based on the different number of unfreeze layers:

Table 3: Fine tuning Results

unfreeze layers	Train Accuracy	Val Accuracy	Test Accuracy
3	0.5487	0.5310	0.54
6	0.7071	0.6021	0.61
9	0.5039	0.5061	0.51

We also tried lowering the learning rate in early stopping so that the network would attempt to train for a longer duration and adding two hidden dense layers, but we did not achieve any improvements.

Here we reported the code used:

```
pretrained_model = keras.applications.vgg16.VGG16(include_top=False,
                                        input_shape=(IMG_WIDTH, ...
                                            IMG_HEIGHT, 3),classes=7,
                                        weights='imagenet')

pretrained_model.trainable = True

set_trainable = False
for layer in pretrained_model.layers:
    if layer.name == 'block4_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False

model = keras.Sequential()
model.add(pretrained_model)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation = 'relu'))
model.add(Dropout(0.5))
# Output layer
model.add(layers.Dense(7, activation='softmax'))
# Compile the model
model.compile(optimizer='adam',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

```
# Compute class weights
class_weights = class_weight.compute_class_weight(
    class_weight='balanced',
    classes=np.unique(label_train),
    y=label_train)
class_weight_dict = dict(enumerate(class_weights))

#earlystopping
early_stopping = EarlyStopping(
    monitor = 'val_accuracy',
    min_Δ = 0.00001,
    patience = 15,
    restore_best_weights = True,
    verbose = 1
)

checkpoint = ...
   tf.keras.callbacks.ModelCheckpoint('/content/drive/Shareddrives
                    /Cocchella_Cantini/models/FineTuningVGG16.h5',
                                            monitor='val_accuracy',
                                            save_best_only=True)

history = model.fit(aug_train.flow(img_train_norm, ...
    categorical_label_train, batch_size=64),
                    validation_data=generator_val.flow(img_val_norm, ...
                        categorical_label_val, batch_size=64),
                    epochs=200,
                    callbacks=[early_stopping, checkpoint],
                    class_weight = class_weight_dict,
                    shuffle = True)
```

Epochs during training:

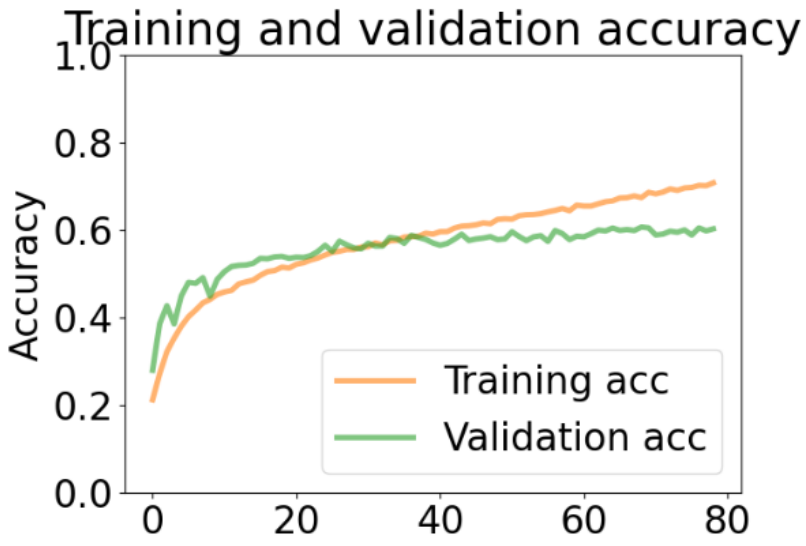
```
Epoch 70/200
359/359 [==============================] - 30s 84ms/step - loss: ...
    0.7740 - accuracy: 0.6855 - val_loss: 1.1163 - val_accuracy: 0.6043
Epoch 71/200
359/359 [==============================] - 31s 85ms/step - loss: ...
    0.7717 - accuracy: 0.6820 - val_loss: 1.1774 - val_accuracy: 0.5879
Epoch 72/200
359/359 [==============================] - 30s 84ms/step - loss: ...
    0.7677 - accuracy: 0.6860 - val_loss: 1.1523 - val_accuracy: 0.5900
Epoch 73/200
359/359 [==============================] - 30s 85ms/step - loss: ...
    0.7495 - accuracy: 0.6933 - val_loss: 1.1473 - val_accuracy: 0.5967
Epoch 74/200
359/359 [==============================] - 30s 83ms/step - loss: ...
    0.7512 - accuracy: 0.6901 - val_loss: 1.1669 - val_accuracy: 0.5944
Epoch 75/200
359/359 [==============================] - 34s 93ms/step - loss: ...
    0.7348 - accuracy: 0.6955 - val_loss: 1.1413 - val_accuracy: 0.5996
Epoch 76/200
359/359 [==============================] - 31s 85ms/step - loss: ...
    0.7348 - accuracy: 0.6967 - val_loss: 1.1647 - val_accuracy: 0.5878
```

```
Epoch 77/200
359/359 [==============================] - 30s 84ms/step - loss: ...
    0.7290 - accuracy: 0.7013 - val_loss: 1.1376 - val_accuracy: 0.6043
Epoch 78/200
359/359 [==============================] - 31s 87ms/step - loss: ...
    0.7239 - accuracy: 0.7006 - val_loss: 1.1629 - val_accuracy: 0.5975
Epoch 79/200
359/359 [==============================] - ETA: 0s - loss: 0.7089 ...
    - accuracy: 0.7071Restoring model weights from the end of the ...
    best epoch: 69.
359/359 [==============================] - 31s 86ms/step - loss: ...
    0.7089 - accuracy: 0.7071 - val_loss: 1.1604 - val_accuracy: 0.6021
Epoch 79: early stopping
```

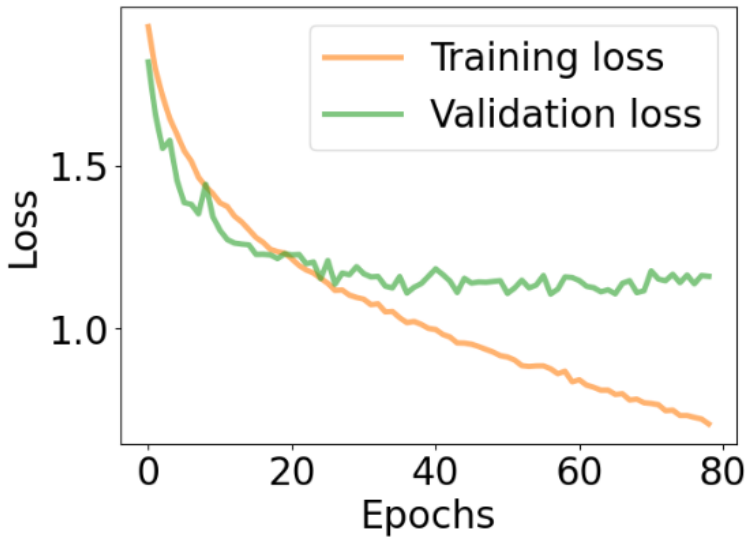
The value obtained for each metric are:

	precision	recall	f1-score	support
0	0.47	0.58	0.52	958
1	0.62	0.60	0.61	111
2	0.49	0.39	0.44	1024
3	0.81	0.81	0.81	1774
4	0.52	0.61	0.56	1233
5	0.53	0.43	0.47	1247
6	0.73	0.76	0.74	831
accuracy			0.61	7178
macro avg	0.60	0.60	0.59	7178
weighted avg	0.61	0.61	0.61	7178

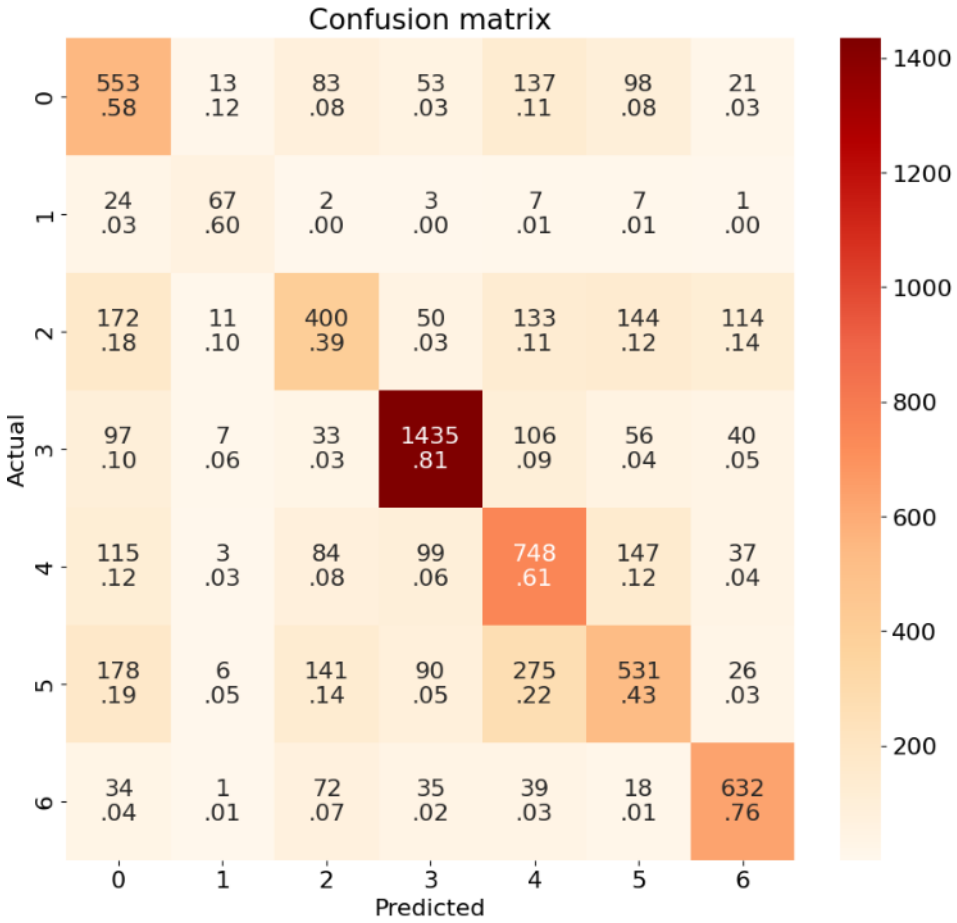
Plot of training and validation accuracy:



Plot of training and validation loss:



Confusion matrix:



3.2.3 Consideration and future implementation

As we can see, we obtained a result that is slightly worse compared to the Convolutional Neural Network (CNN) built from scratch. This disparity in performance can be attributed to the fact that the pretrained networks we employed were originally trained using a dataset comprising 1000 different classes, including objects such as keyboards, mice, pencils, and various animals. These classes are vastly different from the content of our dataset, which primarily consists of human faces.

The divergence in dataset content plays a pivotal role in the performance of pretrained networks. When a pretrained network is fine-tuned or used for transfer learning on a new dataset, its ability to recognize and extract features highly depends on the similarity between the original training data and the target data. In our case, human faces possess distinctive features, textures, and nuances that significantly differ from the objects and animals present in the original pretrained dataset. As a result, the pretrained network might not be effectively capturing the facial features we require for our task.

An intriguing avenue for future research lies in exploring the use of ad-hoc pretrained networks that are specifically designed and trained for facial recognition tasks. Some notable examples of such networks include OpenFace, FaceNet, and VGGFace. These networks have been meticulously crafted to excel in the domain of facial analysis, and they leverage large-scale facial datasets during their training processes. By utilizing these pretrained networks as a foundation for our task, we may harness their specialized ability to extract facial features, which could lead to improved performance and accuracy in our facial recognition task.

4 Experiments with balanced dataset

In order to improve the network's performance, we undertook further investigation after obtaining the results of the previous analysis. In this section, we outline the steps we took during the previous analysis, with the significant distinction that this time, we are working with a balanced dataset and do not require the use of class weights.

4.1 CNN from scratch

The goal remains unchanged, which is to build a facial emotion recognition classifier, this time starting with a pre-balanced dataset. For this analysis, we employed the same type of network that was used in the previous analysis.

4.1.1 CNN structure

After multiple attempts to create custom CNN network structures from scratch, we found that, in this case as well, the network that delivered the best performance was identical to the one selected in the previous analysis.

4.1.2 CNN Construction

Below, we show the code executed to build the selected network:

```
# Define the CNN model

model = models.Sequential()

# First convolutional layer
model.add(layers.Conv2D(32, (3, 3), ...
    activation='relu',padding='same', input_shape=(48, 48, 1)))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(32, (3, 3), activation='relu', ...
    padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.2))

# Third convolutional layer
model.add(layers.Conv2D(64, (3, 3), activation='relu', ...
    padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Conv2D(64, (3, 3), activation='relu', ...
    padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.2))

# fifth convolutional layer
model.add(layers.Conv2D(128, (3, 3), activation='relu', ...
    padding='same'))
```

```
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.2))

# seventh convolutional layer
model.add(layers.Conv2D(256, (3, 3), activation='relu', ...
    padding='same'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.2))

# Flattening Layer
model.add(layers.Flatten())

# first dense layer
model.add(layers.Dense(256, activation='relu'))
model.add(layers.BatchNormalization())
model.add(layers.Dropout(0.5))

# third dense layer (output)
model.add(layers.Dense(7, activation='softmax'))

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

Below we show the summery of the model:

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 48, 48, 32)        320

 batch_normalization (Batch  (None, 48, 48, 32)        128
 Normalization)

 conv2d_1 (Conv2D)           (None, 48, 48, 32)        9248

 max_pooling2d (MaxPooling2  (None, 24, 24, 32)        0
 D)

 batch_normalization_1 (Bat  (None, 24, 24, 32)        128
 chNormalization)

 dropout (Dropout)           (None, 24, 24, 32)        0

 conv2d_2 (Conv2D)           (None, 24, 24, 64)        18496

 batch_normalization_2 (Bat  (None, 24, 24, 64)        256
 chNormalization)

 conv2d_3 (Conv2D)           (None, 24, 24, 64)        36928
```

```
 max_pooling2d_1 (MaxPoolin  (None, 12, 12, 64)        0
 g2D)

 batch_normalization_3 (Bat  (None, 12, 12, 64)        256
 chNormalization)

 dropout_1 (Dropout)         (None, 12, 12, 64)        0

 conv2d_4 (Conv2D)           (None, 12, 12, 128)       73856

 max_pooling2d_2 (MaxPoolin  (None, 6, 6, 128)         0
 g2D)

 batch_normalization_4 (Bat  (None, 6, 6, 128)         512
 chNormalization)

 dropout_2 (Dropout)         (None, 6, 6, 128)         0

 conv2d_5 (Conv2D)           (None, 6, 6, 256)         295168

 max_pooling2d_3 (MaxPoolin  (None, 3, 3, 256)         0
 g2D)

 batch_normalization_5 (Bat  (None, 3, 3, 256)         1024
 chNormalization)

 dropout_3 (Dropout)         (None, 3, 3, 256)         0

 flatten (Flatten)           (None, 2304)              0

 dense (Dense)               (None, 256)               590080

 batch_normalization_6 (Bat  (None, 256)               1024
 chNormalization)

 dropout_4 (Dropout)         (None, 256)               0

 dense_1 (Dense)             (None, 7)                 1799

=================================================================
Total params: 1029223 (3.93 MB)
Trainable params: 1027559 (3.92 MB)
Non-trainable params: 1664 (6.50 KB)
_________________________________________________________________
```

Epochs during training:

```
Epoch 43/200
633/633 [==============================] - 25s 40ms/step - loss: ...
    0.5083 - accuracy: 0.8128 - val_loss: 0.9946 - val_accuracy: 0.6309
Epoch 44/200
633/633 [==============================] - 25s 40ms/step - loss: ...
    0.5026 - accuracy: 0.8167 - val_loss: 1.0108 - val_accuracy: 0.6302
```

```
Epoch 45/200
633/633 [==============================] - 25s 40ms/step - loss: ...
    0.5000 - accuracy: 0.8176 - val_loss: 1.0222 - val_accuracy: 0.6292
Epoch 46/200
633/633 [==============================] - 25s 40ms/step - loss: ...
    0.4980 - accuracy: 0.8161 - val_loss: 0.9527 - val_accuracy: 0.6497
Epoch 47/200
633/633 [==============================] - 25s 40ms/step - loss: ...
    0.4907 - accuracy: 0.8196 - val_loss: 1.0012 - val_accuracy: 0.6414
Epoch 48/200
633/633 [==============================] - 25s 40ms/step - loss: ...
    0.4896 - accuracy: 0.8195 - val_loss: 0.9988 - val_accuracy: 0.6302
Epoch 49/200
633/633 [==============================] - 25s 39ms/step - loss: ...
    0.4796 - accuracy: 0.8228 - val_loss: 0.9811 - val_accuracy: 0.6428
Epoch 50/200
633/633 [==============================] - 25s 40ms/step - loss: ...
    0.4840 - accuracy: 0.8216 - val_loss: 0.9803 - val_accuracy: 0.6407
Epoch 51/200
633/633 [==============================] - 25s 40ms/step - loss: ...
    0.4826 - accuracy: 0.8213 - val_loss: 0.9660 - val_accuracy: 0.6482
Epoch 52/200
633/633 [==============================] - 26s 40ms/step - loss: ...
    0.4754 - accuracy: 0.8244 - val_loss: 0.9739 - val_accuracy: 0.6509
Epoch 53/200
633/633 [==============================] - 25s 39ms/step - loss: ...
    0.4734 - accuracy: 0.8254 - val_loss: 0.9904 - val_accuracy: 0.6351
Epoch 54/200
633/633 [==============================] - 24s 39ms/step - loss: ...
    0.4714 - accuracy: 0.8267 - val_loss: 0.9815 - val_accuracy: 0.6440
Epoch 55/200
633/633 [==============================] - 25s 40ms/step - loss: ...
    0.4699 - accuracy: 0.8273 - val_loss: 0.9707 - val_accuracy: 0.6504
Epoch 56/200
633/633 [==============================] - 25s 40ms/step - loss: ...
    0.4657 - accuracy: 0.8291 - val_loss: 1.0166 - val_accuracy: 0.6262
```

4.1.3 CNN Training

After constructing the network, we initiated the training process by executing the 'model.fit' command. Below, you'll find the code used, along with the various hyperparameters that were configured:

```
# Create an instance of EarlyStopping callback
early_stopping = EarlyStopping(
    monitor='val_loss',     # Monitor validation loss
    patience=10,              # Stop if no improvement after 10 epochs
    restore_best_weights=True  # Restore the model with the best ...
        weights
)

# Create ImageDataGenerator object with data augmentation
datagen = ImageDataGenerator(
```

```
    rotation_range=10,
    zoom_range=0.15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.15,
    horizontal_flip=False,
    vertical_flip=False,
    fill_mode="nearest")

checkpoint = ...
   tf.keras.callbacks.ModelCheckpoint('/content/drive/Shareddrives
            /Cocchella_Cantini/models/BalancedCnnModelFromScratch.h5',
                                               monitor='val_accuracy',
                                               save_best_only=True)

# Train the model with data augmentation and class weights
history = model.fit(
    datagen.flow(img_train_norm, categorical_label_train, ...
       batch_size=64),
    epochs=200,
    callbacks=[early_stopping],
    validation_data=(img_val_norm, categorical_label_val),
    shuffle=True)
```

Here lies the difference compared to what was done in the previous analysis, as we can observe that class weights were not utilized.

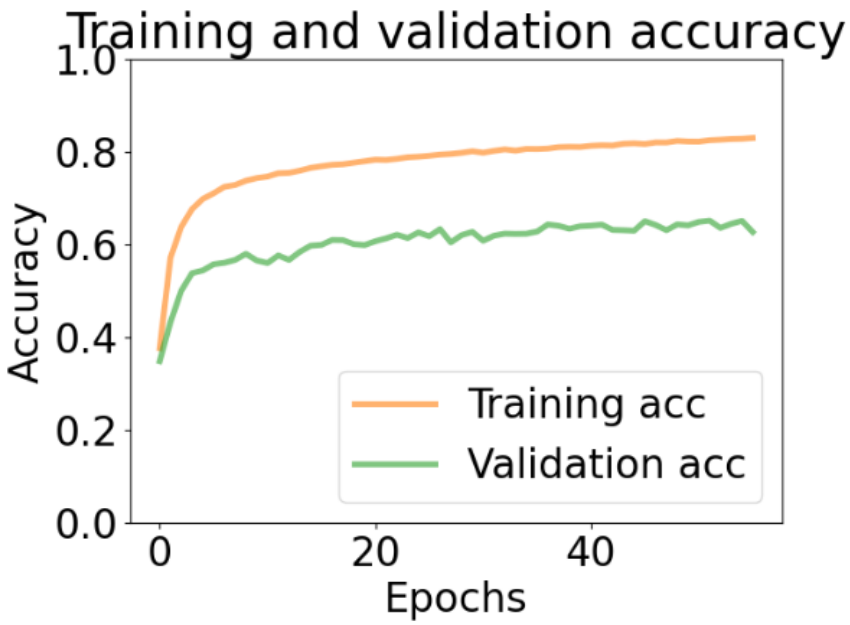
4.1.4 CNN Results

Once the network was trained, we proceeded to test it with the test dataset, and the performance results obtained are as follows:

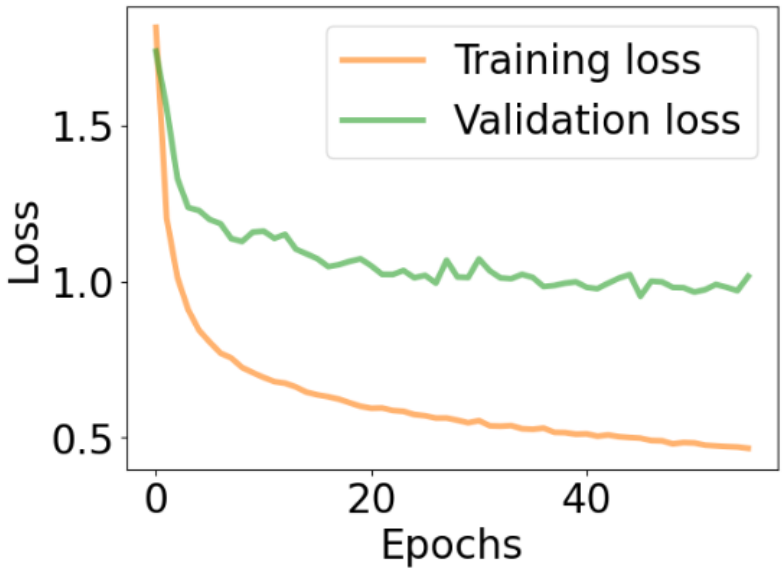
- The value obtained for each metric are:

	precision	recall	f1-score	support
0	0.56	0.58	0.57	958
1	0.84	0.37	0.51	111
2	0.57	0.35	0.43	1024
3	0.85	0.87	0.86	1774
4	0.56	0.67	0.61	1233
5	0.50	0.55	0.53	1247
6	0.77	0.77	0.77	831
accuracy			0.65	7178
macro avg	0.66	0.59	0.61	7178
weighted avg	0.65	0.65	0.64	7178

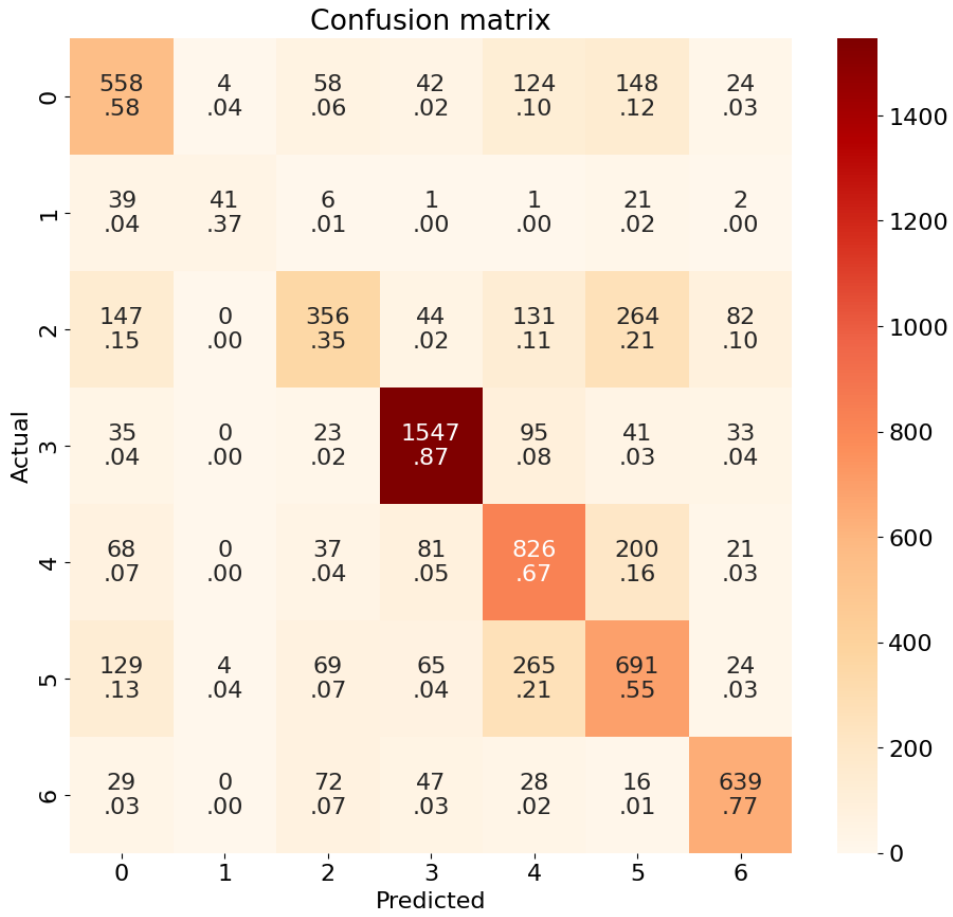
- Plot of training and validation accuracy:



- Plot of training and validation loss:



- Confusion matrix:



4.1.5 Consideration

Despite the final accuracy obtained being very close to that achieved in the analysis with the unbalanced dataset, we can observe a significant difference from the previously conducted analysis in the graphs above. Both training loss and validation loss are decreasing during training, but the validation loss remains significantly higher than the training loss, This fact may deal with a situation where our model is susceptible to overfitting. This discrepancy between the two losses can indicate that the model is learning very well from the training data but is struggling to generalize correctly to unseen data.

4.2 Transfer Learning

Regarding transfer learning, the techniques we carried out were identical to those in the previous analysis. The goal was to assess the performance achieved with the new balanced dataset.

4.2.1 Features extraction

The results obtained during the feature extraction phase using the balanced dataset are as follows:

Table 4: Results of feature extraction

Pretrained network	Train Accuracy	Val Accuracy	Test Accuracy
VGG16	0.7229	0.4361	0.43
VGG19	0.6277	0.3893	0.39
ResNet50	0.1424	0.1428	0.12

Similarly, in this case, VGG16, seems to have slightly better performance. Below we reported the code and the results obtained.

- Added Layers:

```
        model.add(pretrained_model)
        model.add(layers.Flatten())
        model.add(layers.Dense(256, activation='relu'))
        model.add(Dropout(0.5))
        # Output layer
        model.add(layers.Dense(7, activation='softmax'))
        # Compile the model
        model.compile(optimizer='adam',
                      loss='categorical_crossentropy',
                      metrics=['accuracy'])
        model.summary()
```

- Epochs during training:

```
            Epoch 64/200
633/633 [==============================] - 48s 75ms/step - ...
    loss: 1.0120 - accuracy: 0.6261 - val_loss: 1.5253 - ...
    val_accuracy: 0.4110
Epoch 65/200
633/633 [==============================] - 47s 74ms/step - ...
    loss: 1.0233 - accuracy: 0.6250 - val_loss: 1.5276 - ...
    val_accuracy: 0.4086
Epoch 66/200
633/633 [==============================] - 48s 76ms/step - ...
    loss: 1.0165 - accuracy: 0.6246 - val_loss: 1.5301 - ...
    val_accuracy: 0.4136
Epoch 67/200
```

```
633/633 [==============================] - 47s 75ms/step - ...
    loss: 1.0152 - accuracy: 0.6281 - val_loss: 1.5372 - ...
    val_accuracy: 0.4067
Epoch 68/200
633/633 [==============================] - 49s 77ms/step - ...
    loss: 1.0166 - accuracy: 0.6261 - val_loss: 1.5372 - ...
    val_accuracy: 0.4124
Epoch 69/200
633/633 [==============================] - 48s 75ms/step - ...
    loss: 1.0126 - accuracy: 0.6255 - val_loss: 1.5268 - ...
    val_accuracy: 0.4103
Epoch 70/200
633/633 [==============================] - 48s 76ms/step - ...
    loss: 1.0125 - accuracy: 0.6275 - val_loss: 1.5315 - ...
    val_accuracy: 0.4119
Epoch 71/200
633/633 [==============================] - 48s 76ms/step - ...
    loss: 1.0056 - accuracy: 0.6267 - val_loss: 1.5296 - ...
    val_accuracy: 0.4177
Epoch 72/200
633/633 [==============================] - 48s 76ms/step - ...
    loss: 1.0129 - accuracy: 0.6252 - val_loss: 1.5314 - ...
    val_accuracy: 0.4152
Epoch 73/200
633/633 [==============================] - 47s 75ms/step - ...
    loss: 1.0109 - accuracy: 0.6254 - val_loss: 1.5342 - ...
    val_accuracy: 0.4150
Epoch 74/200
633/633 [==============================] - 50s 79ms/step - ...
    loss: 1.0085 - accuracy: 0.6285 - val_loss: 1.5293 - ...
    val_accuracy: 0.4103
```

4.2.2 Fine Tuning

Again, in this phase, the same experiments as in the previous analysis were conducted, and the same layers were unfrozen. Here are the results obtained using the new dataset:

Table 5: Fine tuning Results

unfreeze layers	Train Accuracy	Val Accuracy	Test Accuracy
3	0.8339	0.5644	0.56
6	0.8375	0.5954	0.59
9	0.8574	0.6229	0.62

As we can see, differently from the first analysis, we obtained better results with the last fine tuning test where 9 convolutional layers were unfreeze. Below we report the code and the results obtained in detail.

```
pretrained_model = keras.applications.vgg16.VGG16(include_top=False,
                                        input_shape=(IMG_WIDTH, ...
                                            IMG_HEIGHT, 3),classes=7,
                                        weights='imagenet')

pretrained_model.trainable = True

set_trainable = False
for layer in pretrained_model.layers:
    if layer.name == 'block3_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False


model = keras.Sequential()

model.add(pretrained_model)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation = 'relu'))
model.add(Dropout(0.5))
# Output layer
model.add(layers.Dense(7, activation='softmax'))
model.summary()

Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 1, 1, 512)         14714688

 flatten_5 (Flatten)         (None, 512)               0

 dense_10 (Dense)            (None, 256)               131328

 dropout_5 (Dropout)         (None, 256)               0

 dense_11 (Dense)            (None, 7)                 1799

=================================================================
Total params: 14847815 (56.64 MB)
Trainable params: 14587655 (55.65 MB)
Non-trainable params: 260160 (1016.25 KB)
_________________________________________________________________

opt = keras.optimizers.Adam(learning_rate=0.00001)

model.compile(loss="categorical_crossentropy",
              optimizer=opt,
              metrics=["accuracy"])

# Create ImageDataGenerator object with data augmentation
datagen = ImageDataGenerator(
```

```
    rotation_range=10,
    zoom_range=0.15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    shear_range=0.15,
    horizontal_flip=False,
    vertical_flip=False,
    fill_mode="nearest")

#earlystopping
early_stopping = EarlyStopping(
    monitor = 'val_loss',
    min_Δ = 0.001,
    patience = 5,
    restore_best_weights = True,
    verbose = 1
)

checkpoint = ...
   tf.keras.callbacks.ModelCheckpoint('/content/drive/Shareddrives
                /Cocchella_Cantini/models/BalancedFineTuningVGG16.h5',
                                           monitor='val_accuracy',
                                           save_best_only=True)

# Train the model with data augmentation and class weights
history = model.fit(
    datagen.flow(image_train_rgb, categorical_label_train, ...
       batch_size=64),
    epochs=200,
    callbacks=[early_stopping, checkpoint],
    validation_data=(image_val_rgb, categorical_label_val),
    shuffle=True)
```

Epochs during training:

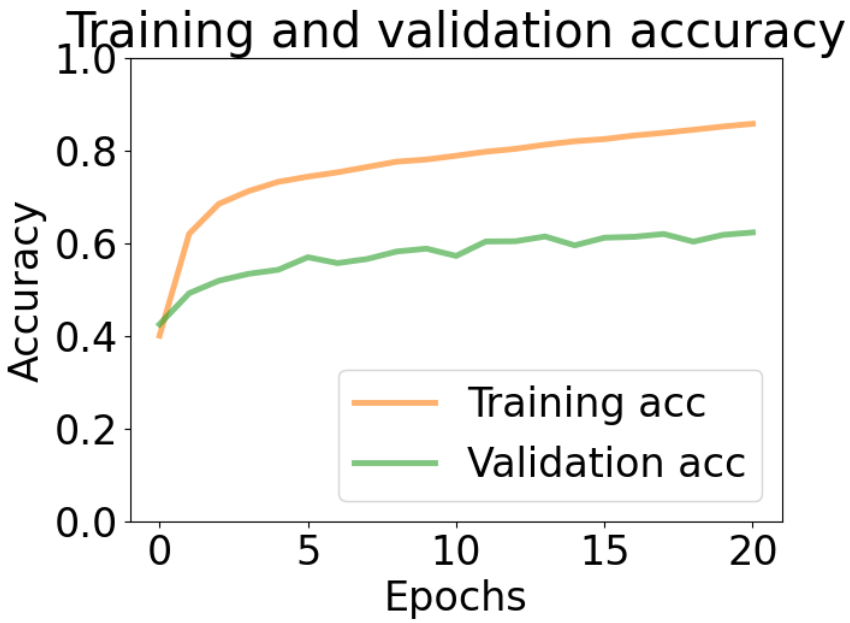
```
Epoch 14/200
633/633 [==============================] - 50s 79ms/step - loss: ...
   0.5284 - accuracy: 0.8123 - val_loss: 1.0597 - val_accuracy: 0.6142
Epoch 15/200
633/633 [==============================] - 50s 79ms/step - loss: ...
   0.5060 - accuracy: 0.8197 - val_loss: 1.1008 - val_accuracy: 0.5951
Epoch 16/200
633/633 [==============================] - 49s 77ms/step - loss: ...
   0.4885 - accuracy: 0.8244 - val_loss: 1.0476 - val_accuracy: 0.6116
Epoch 17/200
633/633 [==============================] - 51s 80ms/step - loss: ...
   0.4694 - accuracy: 0.8321 - val_loss: 1.0769 - val_accuracy: 0.6135
Epoch 18/200
633/633 [==============================] - 52s 82ms/step - loss: ...
   0.4561 - accuracy: 0.8379 - val_loss: 1.0619 - val_accuracy: 0.6196
Epoch 19/200
633/633 [==============================] - 53s 83ms/step - loss: ...
   0.4357 - accuracy: 0.8444 - val_loss: 1.1391 - val_accuracy: 0.6031
Epoch 20/200
633/633 [==============================] - 53s 83ms/step - loss: ...
```

```
    0.4181 - accuracy: 0.8515 - val_loss: 1.0563 - val_accuracy: 0.6177
Epoch 21/200
633/633 [==============================] - ETA: 0s - loss: 0.4037 ...
    - accuracy: 0.8574Restoring model weights from the end of the ...
    best epoch: 16.
633/633 [==============================] - 51s 81ms/step - loss: ...
    0.4037 - accuracy: 0.8574 - val_loss: 1.0598 - val_accuracy: 0.6229
Epoch 21: early stopping
```

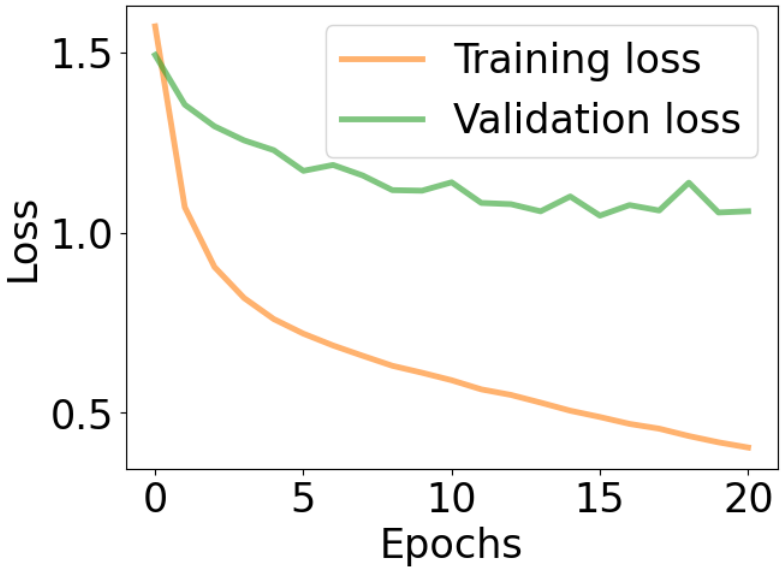
- The value obtained for each metric are:

	precision	recall	f1-score	support
0	0.55	0.50	0.53	958
1	0.82	0.45	0.58	111
2	0.48	0.42	0.45	1024
3	0.77	0.87	0.82	1774
4	0.56	0.64	0.59	1233
5	0.51	0.49	0.50	1247
6	0.78	0.69	0.73	831
accuracy			0.62	7178
macro avg	0.64	0.58	0.60	7178
weighted avg	0.62	0.62	0.62	7178

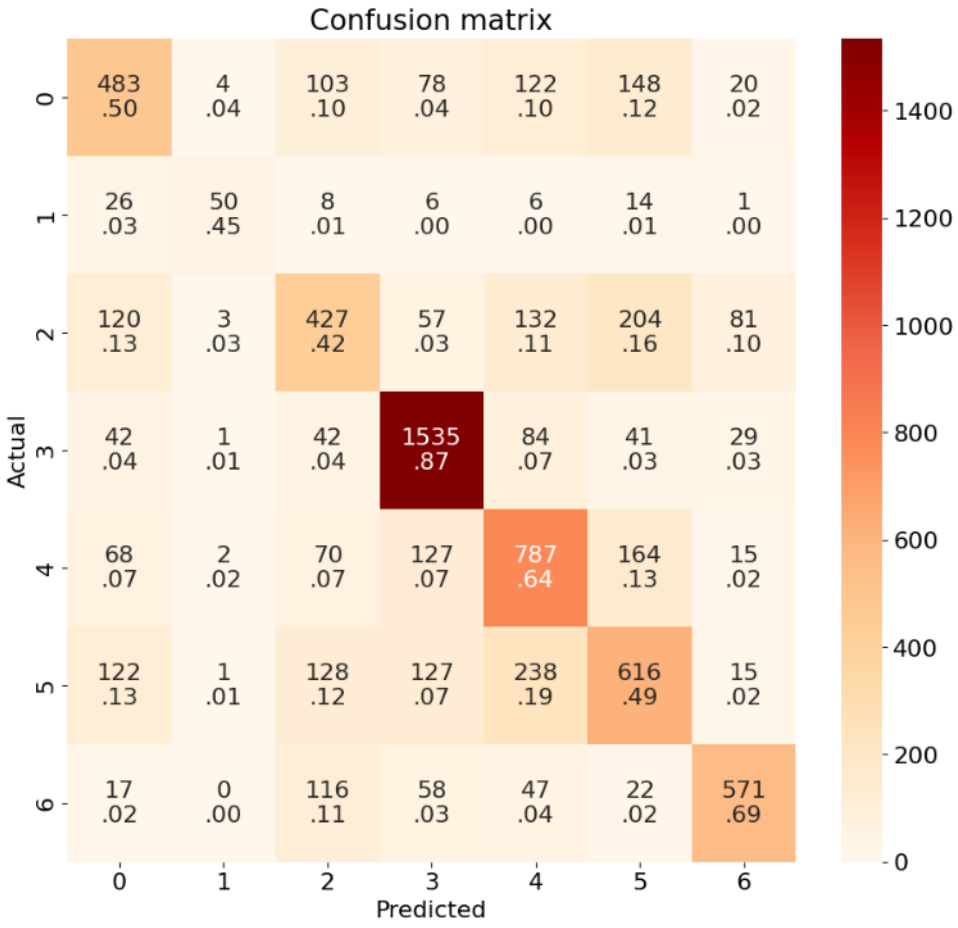
- Plot of training and validation accuracy:



- Plot of training and validation loss:



- Confusion matrix:



4.2.3 Consideration

The graphs exhibit a completely different trend compared to the initial analysis. As we can see, our network is beginning to display signs of overfitting, a behavior to be avoided during CNN training. This situation indicates that our network is struggling to generalize effectively and appears to be relying too heavily on training set features. This conclusion is drawn from the notable difference between the two loss curves. Specifically, both curves show a decrease, but the training loss decreases more rapidly than the validation loss. Additionally, the latter curve starts to oscillate and, after a certain point, ceases to decrease. We attempted to address this situation by experimenting with various combinations of hyperparameters, but unfortunately, we did not observe any improvements in terms of overfitting.

5 Conclusion

The results indicate that the most favorable performance was achieved in the first analysis, where the dataset was not balanced in advance. In both cases, the superior performance was observed when constructing the CNN from scratch. This can be attributed to the pre-trained network's training on a dataset that significantly differed from the one used in our problem.

Comparing the results obtained from the two analyses, we can observe that in the second case, we obtain a significantly worse situation compared to the first case, even though the initial dataset was the same. This suggests that the problem that eventually resulted in overfitting during the second analysis may be attributed to the initial dataset balancing through data augmentation. This procedure effectively balances the dataset by increasing the number of images in the minority classes through slight variations of existing images within that class. This has resulted in a high number of similar images, which inhibit the network's ability to generalize adequately.

To try to improve the accuracy of the models, potential experiments could involve: balancing the dataset by introducing more pronounced variations to the images or integrating new images into the minority classes, expanding the dataset's size, and, finally, employing deeper networks.