Faces and Comics recognizer with Convolutional Neural Networks

Margherita Maroni - 965589

March 30, 2022

Abstract

In this paper it is presented how a Convolutional Neural Network can be built for comics or faces recognition of images from a dataset taken from Kaggle. I use the CNNs because they are really useful for computer vision: thanks to their filters, they are able to detect patterns inside the images. It is performed a model selection tuning the hyperparameters in such a way to get the model with the highest cross-validated accuracy. The model has been trained on 7,000 images and then tested on 3,000 images obtaining a test accuracy of 99.86%

1 Introduction

Comics and faces detection is an interesting area and in recent years many techniques have been proposed to distinguish comics from faces. In this paper, I am going to use Deep Neural Networks for comics or faces recognition.
I will firstly use a simple Neural Network to classify the images and then I will try to improve my model with Convolutional Neural Networks that turn out to be very useful for computer vision. Indeed, Convolutional neural networks are interesting because they are artificial neural networks that have some type of specialization able to pick out or detect patterns and make sense of them. This ability in pattern detection is what makes Convolutional Neural Networks so important for image analysis.
For this purpose, I use a dataset from Kaggle with images both of comics and of real faces. In the next sections I am going to explain how the dataset is composed and how I make the pre-processing, thereafter I will show the neural networks that I created.
I start doing some experiments with simple neural networks, inserting the parameters by feel (i.e., without the model selection). Thereafter, I do a model with Convolutional Neural Network following the suggestions given by the model selection. I use the GridSearchCV that is a hyperparameter optimization technique of the class model_selection provided in *scikit-learn* library. Hyperparameter optimization is a fundamental part of deep learning that assumes a big role in neural networks because they are difficult to configure and there are a lot of hyperparameters that need to be tuned.
Eventually, using 10,000 images (i.e., 7,000 for the train and 3,000 for the test), I obtain a test accuracy of 99.80% for the CNN where the hyperparameters are selected through the GridSearchCV.

[1]I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

[2]Link to the GitHub repository: https://github.com/Margherita98/face_comic

2 Materials and methods

In this section, I am going to illustrate the dataset that I use for this project and the pre-processing phase with train and test split that I run on the data for creating the Convolutional Neural Networks.

2.1 Dataset

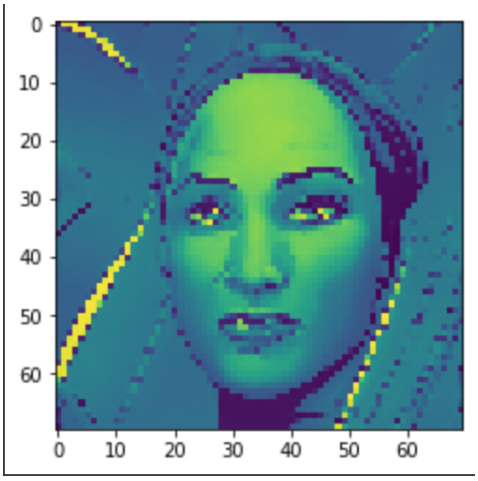
The dataset is taken by Kaggle[1] and it contains 20,000 images divided in two folders:

- in the first folder there are 10,000 pictures of real faces;
- in the second folder there are 10,000 images that are comics.

For the Convolutional Neural Network I decide to use 10,000 images as input: 5,000 taken from the folder of comics and 5,000 taken from the folder of faces.
Reducing the amount of images taken from the original dataset permits me to use a high resolution of the image (size 100x100) and three channels of colors.
Indeed, we have a lot of examples to teach the model what is a face and what is a comic.

2.2 Preprocessing

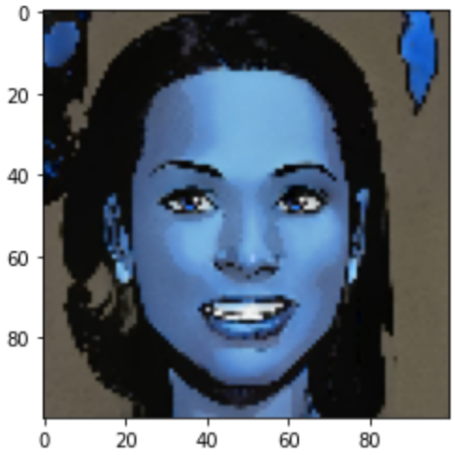
After setting the directory of the *subfolder* containing the two *subsubfolders* "comics" and "faces", I convert the images into an array with the function *iamread* from OpenCV. The function *iamread* takes as input a jpg file and it gives back an array that is the numeric encoding of the image.
Initially, for the first simple neural network, I convert the images into one channel of color just to reduce the dimension because at the beginning the colors of the images are not fundamental. With the size of the image 70x70 and the third dimension put equal to 1, I visualize an image as the following one:



However, when I make the Convolutional Neural Network with model selection, I retain the three channels of colors and I resize my images to a dimension 100x100x3 in order to have a more accurate result from the model.

[1]Kaggle data avaialable at the following link: https://www.kaggle.com/datasets/defileroff/comic-faces-paired-synthetic-v2

Then, an example of image that I visualize from the new resize (i.e., 100x100x3) of the pictures is as the one it follows:



Originally, images' shape is 1024x1024x3, but giving as input to the model images of size 1024x1024 with three channels of colors will be very time-consuming for the neural networks. This is the reason why I decide to reduce the size of the images to 100x100 with three channels of colors.

2.2.1 Train and Test split

The dataset I use for my analysis contains 10,000 images: the first 5,000 are faces and the last 5,000 are comics. For the purpose of doing the train and test split of the dataset and with the aim of obtaining mixed sets, I reshuffle the images.
At this stage, I need to map my images to labels of numerical values: 1 is the label assigned to faces and 0 is the label assigned to comics. Then, I create a list containing 10,000 items: each item is a list containing an array of dimension (100, 100, 3) and the associated label (either 0 or 1).
Thereafter, I do the train and test split, assigning 70% of the files to the training part of the model and the remaining 30% to test it with new images never seen before. With the aim of having balanced train and test sets, I specify the parameter *stratify y* in the function *train and test split*.
In the training phase, I applied a three-fold cross validation such that the algorithm learns the weights on 2/3 of the training data and it retains the remaining 1/3 of training data for the validation. Therefore, the process is clearly repeated three times, one for each split of the cross-validation.
Then, I encode the categorical features as a one-hot numeric array with *OneHotEncoding* of the labels taken from *sklearn.preprocessing*.

2.3 Convolutional Neural Networks for visual recognition

In this section, I am going to explain the Convolutional Neural Networks I developed with purpose of faces / comics recognition.
As we know, Convolutional Neural Networks are neural networks that learn some type of specialization for being able to detect patterns and make sense of them.
Basically, Convolutional Neural Networks have hidden layers called convolutional layers and also

non-convolutional layers. The former layers are what is able to detect patterns. With each convolutional layer it is needed to specify the number of filters the layer will have and these filters are actually what detects edges, circles, squares, corners, etc., in the images. The deeper the network goes, the more sophisticated the filters become.
Before implementing the model of the CNN, I run the *GridSearchCV* with cross-validation. I will feed in input different training and test splits, namely x_train, x_test, y_train, y_test and I will also input my model from the library *sklearn* using the function GridSearchCV from the class *model selection*. Then, in a dictionary called *param_grid*, I specify in the keys the names of the hyperparameters, and in the corresponding values the list with the possible choices.
Below it is shown the function *algorithm_pipeline* that implements the dictionary param_grid:

```
def algorithm_pipeline(x_train, x_test, y_train, y_test,
                       model, param_grid, cv=3, scoring_fit='accuracy',
                       ):
    gs = GridSearchCV(
        estimator=model,
        param_grid=param_grid,
        cv=cv,
        n_jobs=-1,
        scoring=scoring_fit
    )
    fitted_model = gs.fit(x_train, y_train)

    return fitted_model
```

Thereafter, I define the neural network architecture and since I have a dataset that consists in pictures, I define it with CNN. For the purpose of doing the CNN, I previously need to expand the dimensions of my x_train and x_test and add another dimension for obtaining the right input data.

So, firstly I expand the dimensions:

```
x_train = np.expand_dims(x_train, -1)
x_test = np.expand_dims(x_test, -1)
```

And then I define the CNN architecture as it is shown below:

```
def build_cnn(activation = 'relu',
              dropout_rate = 0.2,
              optimizer = 'Adam'):
    model = Sequential()

    model.add(Conv2D(32, kernel_size=(3, 3),
                     activation=activation,
                    input_shape = (100,100,3)))
    model.add(Conv2D(64, (3, 3), activation=activation))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(2, activation='softmax'))

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

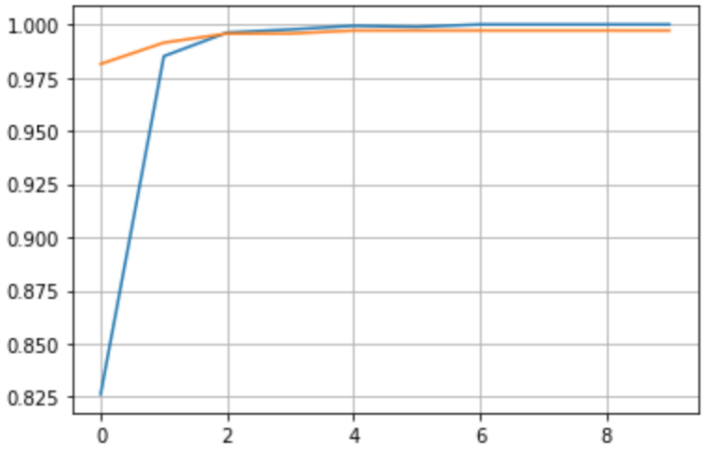
At this stage, I just set the parameters and model to input into the *algorithm_pipeline* and with the *param_grid* I select three parameters I would like to optimize and for each of them I set two possible values:
- optimizer: ['Adam', 'Nadam'];
- dropout rate: [0.3, 0.7];
- activation layer: ['relu', 'sigmoid'].

```
param_grid = {
              'optimizer' :           ['Adam', 'Nadam'],
              'dropout_rate' :        [0.3, 0.7],
              'activation' :          ['relu', 'sigmoid']

             }

model = KerasClassifier(build_fn = build_cnn, verbose=0)

model = algorithm_pipeline(x_train, x_test, y_train, y_test, model,
                           param_grid, cv=3, scoring_fit='accuracy')
```

From this GridSearchCV it turns out that best parameters are:
- *relu* for the activation layer;
- 0.3 is the best dropout rate;
- *Adam* is the best optimizer.

3 Results of the experiment

After selecting the best parameters, I run a model of CNN for 10 epochs, using *relu* as the activation layer, 0.3 as the dropout rate and *Adam* as the best optimizer in the *compile*.
I fit the CNN with a batch size of 128 and 10 epochs and at the tenth epoch, the model reaches an accuracy of the validation equal to 100% .
The accuracy on the train and validation set are shown in the following plot:



Then, I evaluate the model on the test to see how it behaves on images never seen before. It turns out to output a test accuracy of 99.86% and it is a great result!
Eventually, on a test of 3,000 images, the model does an error in the predictions of the label for four images.
Below, I print the four misclassifed images:

4 Discussion

The CNN model developed for the comics / faces recognition in this paper could be surely further improved and it has the potential to obtain even an higher accuracy on the test set.
Of course, I could have used the whole set of images from the Kaggle dataset and probably having a training set of 14,000 images would have boost the accuracy even on the test set. Although four misclassified images are not a bad result for a CNN with only three hyperparameters tuned.
Moreover, I could have used *Keras* preprocessing layer for data augmentation and this choice would have led me to train a better model on the images and to obtain more accurate neural networks that perform even better and that are more reliable.
Eventually, I may want to scale my training onto multiple GPUs on one machine, or multiple machines in a network or on Cloud TPUs. In order to support these use cases, *Tensorflow* has many available strategy such as MirroredStrategy, *TPUStrategy*, *MultiWorkerMirroredStrategy*, *ParameterServerStrategy*, *CentralStorageStrategy*.
For example, the TPUStrategy is useful for run Tensorflow training on Tensor Processing Units. Basically, TPUs are practical because they permit to accelerate machine learning workloads and they are also avaialble on Google Colab. Even though this strategy could be very interesting to scale up data size, it is needless to say that it is necessary to have a PC with more GPUs or to use them on Google Colab. Practically, when I try to implement TPUs strategy on my code, it does not output any more interesting result than what I already obtained before.