Algorithms for Massive Data Cloud and Distributed Computing
Module: **Algorithms for Massive Data**
Project 4: Comics and Faces Recognizer

Andrea Pio Cutrera - 965591

March 29, 2022

Abstract

This project wants to develop a **deep-learning-based system** able to discriminate between real faces and comics images, using the ≪Comics faces≫ data available in Kaggle. Some models, implemented from the standard libraries of Python, like Tensorflow and and Keras, are used to train and evaluate built-in neural networks architectures. The **best model**, selected through hyper-parameter tuning, is a convolutional neural network which attains a pretty good external validity: **99.93% accuracy** on the test set.

Keywords— Deep Learning, Computer Vision, Image Content Recognition

Contents

[1] **Link to the GitHub repo**: https://github.com/andreacutrera/comics_faces_recognizer

[2] 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

1 Introduction

Computer vision is the field of Artificial Intelligence which makes machines able to mimic one of the senses of human intelligence: the *human sight*. Machines have become pretty much capable of recognizing objects in images, videos and sounds, and in this little experiment I tried to find out how to build a **supervised classifier** able to distinguish in pictures *real human faces* from *drawn ones*.

Below we can see some examples of the first 10 images of real pictures and subsequently the first 10 images of drawn faces to have an idea on what kind of pictures we are now going to deal with.

2 Data and Methodology

2.1 Data Description

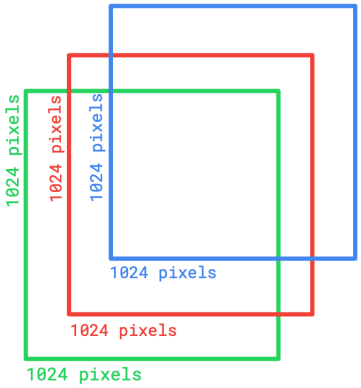
Comics faces data available in Kaggle[1] are just pictures organized in **2 folders**:

- One folder containing **10,000 pictures** of **real images**;
- Another folder containing **10,000 corresponding drawings** of real images, which we call **comics**.

[1]The ≪Comic faces≫ dataset is published on Kaggle and released under the CC-BY 4.0 license, with attribution required: https://www.kaggle.com/datasets/defileroff/comic-faces-paired-synthetic-v2

2.2 Data Pre-processing

The images in the 2 folders have been pre-processed through *OpenCV* Python library which includes several hundreds of computer vision algorithms for image operations[2].
Repeating for both the folders (comics and faces) an entire cycle over all the images inside, we are able to load all the pictures in an array form. It has been used the function of *OpenCV* (i.e. ***imread***) which allows to get the numerical encoding of the image in an array of 3 dimensions: *1024x1024x3*.
Essentially **squared images** with 1024 pixels' values for the horizontal and vertical dimensions. The additional third dimension gives us the **3 channels** of colors, namely **RGB**: Red, Green and Blue.
So our original images are encoded like 3-layered squared images in which the first layer represents the picture in scales of greens, the second in scales of reds and the third in scale of blues, regardless the ordering. Values inside each pixels ranges from 0 to 255, which are all the possible color tone for each channel.



For the purpose of our experiment I decided to use just one channel of color reducing the last dimension to just one.
What is left to choose is the dimension of images we want to retain. Instead of using the maximal definition of original pictures, as we can see from below, the resolution with 128x128 pixels return us images of very good quality.

Since our label set is binary, we just attach to each single image a label encoded as 1 for the faces and 0 for comics, as answering to the question: *Is it a real face? Yes=1, No=0*

Initializing an empty list and appending lists containing each the couple *array-label* we get the data from the folders in an encoded form.

[2]Open Source Computer Vision Library: http://opencv.org

```
working_directory = !pwd                # get present working directory
working_directory = working_directory[0] # get the full string
working_directory
```

```
DIR = working_directory + "/face2comics_v2.0.0_by_Sxela/face2comics_v2.0.0_by_Sxela" # go inside the 2 folders
CATEGORIES = ["comics", "faces"] # set the names of the subsequent folders, which would correspond to the
                                 # labels of the class of our images - Comics and Faces.
```

```
# Define the image size below before the starting - for a good quality use 100
data = []
image_size = 128
```

```
def get_data():
    for category in CATEGORIES:
        path = os.path.join(DIR, category)                # 2 iterations for both categories create
                                                          # respectively 2 paths (2 folders)
        classification_label = CATEGORIES.index(category) # get the classification label as an integer
                                                          # (is it a real face ? Yes = 1, No = 0)
        for image in os.listdir(path):
            try:
                image_array = cv2.imread(os.path.join(path, image), cv2.IMREAD_GRAYSCALE) # cv2 read an image for all the images
                image_resized = cv2.resize(image_array, (image_size, image_size)) # resize the image
                data.append([image_resized, classification_label]) # append each image encoded along with its label
            except Exception as e:
                pass
```

The list named data would have been composed by the 20K couples *image-label*, but for the sake of this experiment they have been retained 4K comics and 4K real faces pictures, giving us a balanced dataset, which has the following logical representation:

- binary **label set**: $\mathcal{Y} = \{0, 1\}$
- d = 128x128 features for our **data domain**: $\mathcal{X} \equiv \mathcal{X}_1 \times ... \times \mathcal{X}_d$, whose values $\mathcal{X}_i \subset \mathbb{Z}^+ \; \forall i = 1, ..., d$; in particular each $\mathcal{X}_i \in \{0, ..., 2^8 - 1\} \equiv \{0, ..., 255\}$

Values inside the arrays are then *normalized* in the range 0-1 by dividing them by the maximum value of 255. Once the list of lists (*image-label's*) has been created, we can make a *random shuffle* in order to have images mixed. Subsequently the encoded images and labels have been saved in 2 separate *numpy arrays*. Labels have been *one-hot-encoded* and a dimension has been added to the arrays of images in order to get the correct shape for the input into the *Convolutional layers*. This third dimension has value one, because I'm going to use just one out of the 3 channels of colors.

The last step we require to set up our experiment is the **random split** between **train and test set**. Part of training data have been retained as **validation set**, to get the best hyper-parameters. Test set would be used instead to measure the external validity of the trained and validated model (on never seen images).
Split has the following form:

- 5760 **training** *images* with corresponding 5760 training *labels*;
- 640 **validation** *images* with corresponding 640 validation *labels*;
- 1600 **testing** *images* with corresponding 1600 testing *labels*.

2.3 Model

The most common used models for *image classification* are the **Convolutional Neural Networks**. Instead of using the simple *Multi-layer Perceptron* which receives an input signal and reaches the output layer through n fully-connected hidden layers, I have chosen to build the Neural Network Architecture with 3 main features:

- **Conv2D**: 2 *convolutional layers* to get some transormations of the images able to capture patterns inside the pictures;
- **MaxPooling2D**: 2 *maxpooling layers* to get a reduced form of the images which come from the convolutional transformation;
- **Dropout**: layer to regularize the learning phase.

Going a little bit more in details, the convolutional layer aims to find patterns through filters with dimension specified in the kernel size as 3x3. What remains to be chosen is the number of filters.
For what concerns the maxpooling layer, I specified a pool size of 2x2, which reduces to one pixel all the pools of 2x2 squares, reducing the overall image by half the pixels in columns and rows.
Then with a flatten layer we get the data flattened. The last but one layer is the dropout one, in which I need to choose the share of random weights to which I can stop the update.
The final layer is the fully connected output layer with 2 neurons: what we want to get from that are the probability distribution of the binary predictors for the labels.

3 Hyper-parameter tuning for Model Selection

Once the model architecture has been chosen, what we need is to understand which are the best hyper-parameters to be tuned. Performing a grid search over all possible combinations of values specified in each hyper-parameter we compare the validation accuracies across all the models. For simplicity I have set just 2 values for each of the 4 following hyper-parameters:

- filters_1: **numbers of filters** in the first *convolutional layer* [100, 200]
- filters_2: **numbers of filters** in the second *convolutional layer* [25, 50]
- **dropout rate**: [0.5, 0.7]
- lr: **learning rate** [0.01, 0.001]

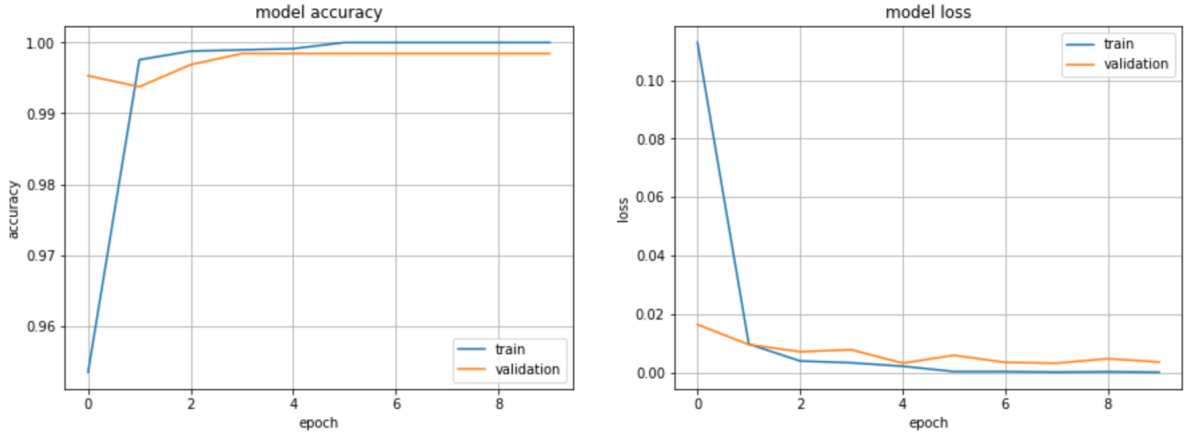
```
def build_model(hp):
    model = tf.keras.Sequential(
    [
        keras.Input(shape=input_shape),
        #convolutional layer
        layers.Conv2D(hp.Choice('filters_1', values=[100, 200]), kernel_size=(3, 3), activation="relu"),
        #pooling layer
        layers.MaxPooling2D(pool_size=(2, 2)),
        #another convolutional layer
        layers.Conv2D(hp.Choice('filters_2', values=[25, 50]), kernel_size=(3, 3), activation="relu"),
        #another pooling layer
        layers.MaxPooling2D(pool_size=(2, 2)),
        #flatten layer
        layers.Flatten(),
        #dropout layer
        layers.Dropout(hp.Choice('dropout_rate', values=[0.5, 0.7])),
        #last 2 neurons for the outcome
        layers.Dense(num_classes, activation="softmax")
    ])

    #compile model
    model.compile(loss='categorical_crossentropy',
                  optimizer=tf.keras.optimizers.Adam(learning_rate=hp.Choice('lr', values=[1e-2, 1e-3])),
                  metrics=['accuracy'])
    return model
```

3.1 Best Model

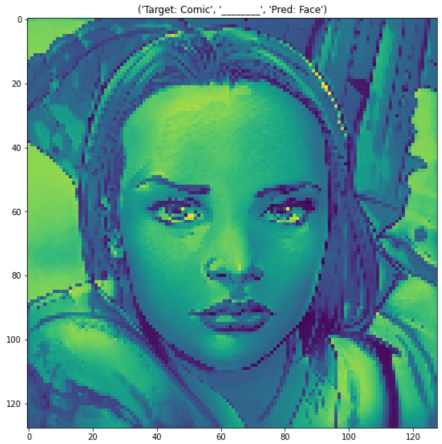
In the experiment I made, the model which have come out to be the best in the validation set is the one with the following combination of hyper-parameters:

- **filters_1**: 100
- **filters_2**: 50
- **dropout rate**: 0.5
- **lr**: 0.001

Retraining the model with the configuration above specified, as we can see in the following graph the **best number of epochs is 4**, when validation accuracy reaches a stabilized value around **99.84%**.

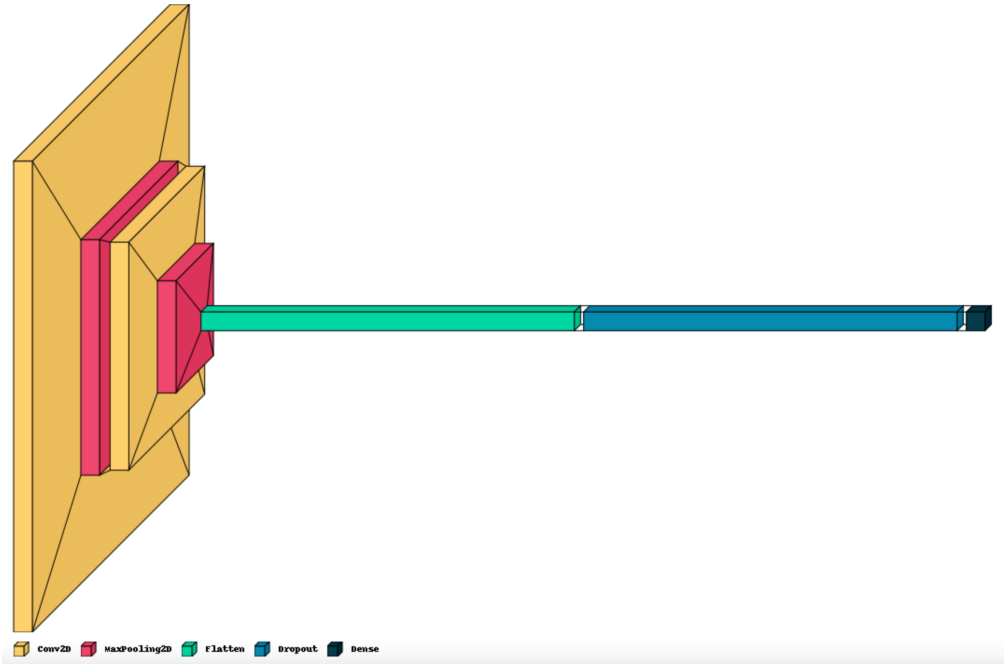


In the last phase we just retrain the best model with the best number of epochs and evaluate it on the test set. The result is an **impressive accuracy** of **99.94%**.
Just 1 out of the 1600 never seen testing examples is misclassified. Below I report the picture just to understand which is the image in which the prediction has not been made correctly.



4 Discussion

The model architecture built so far is the following described in the picture below:



Accuracy with these pretty simplified experiments reached very good values, but it is still open to many improvements to the performance and the computational time.

First of all what we can improve is an **increase** in the **number of examples** used to train, validate and test the models: all the available images in the Kaggle repository are 20K, and we just used 40% of them (i.e. 8K).

Another thing which could help finding models with a greater performance is to **increase the cardinality of the sets of the hyper-parameters** to be tuned. In the experiments done so far, we have compared 4 hyper-parameters with 2 values each, for a total number of 16 (i.e. $2x2x2x2$) possible models. We could have increased the new set for the numbers of filters in the first convolutional layer to: [50, 100, 150, 200, 250, 300], or the dropout rate among the values: [0.1, 0.3, 0.5, 0.7] or the number of layers to include in the neural network architecture, and so on.

The problem with these 2 approaches is that the **required computational time increases** a lot. To solve this issue we need to increase computational power and distribute computation of the training phase in parallel jobs.

With the buil-in API of *tensorflow* (i.e. **distribute.Strategy**) we can **distribute the computation** over more GPUs, more machines and TPUs (i.e. *Tensor Processing Units*) with minimal modifications to the code implementation[3]. Once set-up the resources on which you want to distribute the computation, GPUs or TPUs, you just need to create the instance of the strategy which could be one of the following: *MirroredStrategy, TPUStrategy or MultiWorkerMirroredStrategy.*

[3]For the complete guide follow the documentation: https://www.tensorflow.org/guide/distributed_training