Algorithms for Massive Data, Cloud and Distributed Computing

Module: Algorithms for Massive Data

Università degli Studi di Milano La Statale – MSc DSE

Project: Deep Learning for Comics and Faces Recognizer

Camillo Giuseppe Majerczyk (966956)
Milano, 20/08/2022

Abstract

The goal of this project is to develop a deep-learning-based system to discriminate between real faces and comics, using the "Comics faces" dataset available on Kaggle. For this task I explore the Convolutional Neural Networks (CNN), which are largely used for image classification. So, I use TensorFlow and Keras, two Python libraries to evaluate built-in neural networks architectures. Finally, the best model is chosen through hyper-parameters tuning.
The complete report highlights the following parts of the implementation: data organization and preprocessing, creation of Neural Network architecture, model selection and final conclusions.

Contents

1. Introduction

Artificial Neural Networks are inspired by the nervous system, which is an aggregation of neurons. Here, a neuron is a computational device that receives an input and produces an output. Different networks architectures have been studied, each is more suitable for specific uses. When it comes to image recognition, Convolutional Neural Networks better fit the task. Their architecture introduces the so called “convolutional layer”, which convolve the input and pass the result to the next layer. This work explains the process of creating a well-performing binary classifier.

2. Dataset

The “Comic faces” dataset is published on Kaggle and released under the CC-BY 4.0 license, with attribution required. It contains pictures organized in two folders. One containing 10’000 pictures of real images. The other one containing 10’000 corresponding drawings of the real images, that are the comics. For a total of 20’000 images in 1024x1024 format.

2.1 Data organization

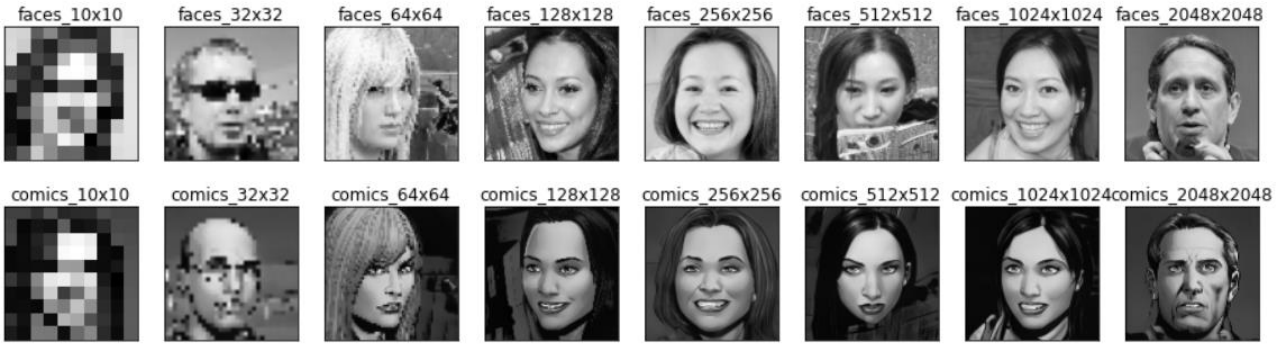
The complete dataset is downloaded directly from Kaggle to Google Colab using the Kaggle API. The Jupyter Notebook attached to this report is directly executable on Google Colab substituting the personal Kaggle API. That is, the username and key in the following cell:

```
!echo '{"username":"camillomajer","key":"ab79bba                    9e"}' > ~/.kaggle/kaggle.json
!kaggle datasets download -d defileroff/comic-faces-paired-synthetic-v2
```

Then, a label is applied to each image. That is, 0 for real faces and 1 for comics. The images and the respective labels are stored in separated variables. Thus, after the pre-processing phase, the overall dataset made up of image-label pairs is split in a training set and a test set. The training set is further split in each training section in a training set and validation set, that are automatically created by TensorFlow to compare the performances of the models and validate them.

2.2 Data pre-processing

To optimize the performance, in this work I focus on the size of the images and the colours. Experimental results in literature showed that classification with grayscale images usually results in higher accuracy classification than with RGB images. Hence, in this project the images have been processed in grey scale.
For what concern the size of the images, reducing the dimensions helps with the computational and storage capacity. Trying different visualisations, it is possible to notice that a 128x128 pixels resolution keeps a good quality of the images.



After this, the entire dataset has been randomly shuffled. This step is important to avoid biased performance of the CNN model due to a learning process on images sequentially ordered per category. At this point, encoded images and the labels can be stored in two arrays. Then, I apply one-hot-encoding to the labels, using the Keras function *to_categorical* which converts vectors of integers into binary class matrices. This is a typical technique used to encode multi-class features, but the price to pay is that the number of features grows up. In this case features are binary and already encoded as 0 and 1.

Finally, values for each pixel range from 0 to 255. The processing phase in this work concludes with the normalization of the pixel values of the images between 0 and 1. Each input to the neurons of a neural network must be encoded as a real number, but inputs with large integer values can disrupt or slow down the learning process. This normalization easy the work of the neurons.

3. Methodology

A Convolutional Neural Network is a Deep Learning algorithm which can take an image as input, detect its main characteristics and distinguish it from others. Hence, the algorithm reduces the images without losing critical features for a good prediction.
A layer in the CNN is composed by a “convolutional layer” and a “pooling layer”. Where the number of layers defines the level of details captured. The higher is this number, the higher is the complexity and the more computational power is required.
The convolutional operation takes place in the Convolutional Layer and involves the “filter” element. This shifts through the image until it is considered in its entirety. The number of times this filter shifts depends on the “stride length”. The objective is to extract high-level features from the image. Actually, the first convolutional layer captures the low-level features. The high-level features are captured adding more layers, which define the network architecture.
On the other hand, the Pooling Layer reduces the dimensionality of the convolved features, helping in decrease the computational power required. Also, it extracts the dominant features. It is a “max pooling” if it returns the maximum value of the filter’s window. Or it is an “average pooling” if it returns the average of the values of the filter’s window. Generally, max pooling performs better.

Once the model can understand the features, the image must be flattened into a column vector in order to be passed to a “Dense layer”, also called “fully-connected layer”. Each neuron of this layer receives input from all the neurons of previous layer. This is given as input to a neural network able to perform the classification using a Softmax function. This technique has two interesting results:

the obtained values sum up to 1 and they are all non-negative. This means that we can interpretate them as a probability distribution over the classes. In our case, the probability of being a real face image or a comics image.

3.1 Algorithms and implementations

There are various architectures of CNN. To better exploit the computational and time resources, in this work only some hyperparameters are tuned. I referred to efficient choices used in the literature to set many of the parts composing the models trained. Hence, for example, the binary cross entropy loss function and the accuracy metric are standard choices that fit good for this task. The choice of a 2X2 max pooling layer is generally more efficient than the average pooling, as per the previous paragraph. And the 3X3 size of the convolutional window is a default setting.

Here, every model trained has at least two convolutional layers associated with a ReLu activation function. This is a non-linear function which main advantage is that it does not activate all the neurons at the same time, and it is generally more efficient than the Sigmoid activation function. Other common features are mandatory for the correct functioning of the algorithm used, as described in the previous paragraph. For example, the presence of a flatten layer. Furthermore, every model uses the Adam Optimizer. Finally, trying to avoid overfitting I also add a “drop out layer”. This is a technique to limit how weights are changed during the training phase, and it is done fixing a probabilistic rate.

3.2 Hyperparameters tuning and model selection

A neural network has many hyperparameters to set. But there is not a rule of thumb to define the values that best fit any dataset. Hence, the tuning is performed to find sets of hyperparameters to build a good model over specific data. In this work, I divide the tuning into two parts. One aimed to find a good architecture. For this step perform the tuning with a Grid Search over the number of neurons in each layer and the number of convolutional layers. Once the appropriate initial architecture is set, I try to further optimize the best model evaluating more learning rates for the Adam optimizer and probabilistic rates for the newly add drop out layer. For both phases I use different subsets of the original training set, according to the computational resources and the time they require.

The choice of evaluating no less than two convolutional layers comes from the expected performances and the computational resources available. The evaluation of too many models would become too costly in terms of time, and it is unluckily to obtain a better performance with a single convolutional layer, as it is more difficult to detect high-level features. Moreover, we know from literature that to improve the performance is better to add layers instead of neurons to each of them. Based on this last consideration, I also decide to explore a range of layers sizes with just three values (32, 64 and 128). Similarly, I decide to evaluate models always built with a single dense layer. Indeed, the performance could be optimized to avoid overfitting with dropout layers, rather than multiple fully connected layers.
After an attempt with a training set containing 16000 elements, I notice that the time required to provide an output with these parameters is still too high. Therefore, I decide to reduce the training

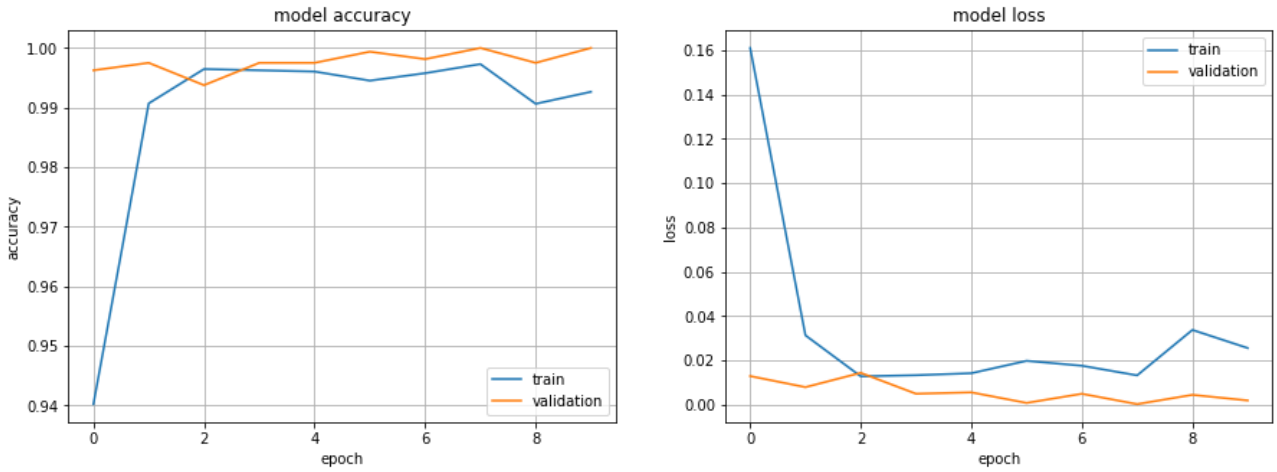
set size to sub-training-set of 4000 elements for the Grid Search process. From these results, I will consider the best performing hyperparameters.

```
Epoch 1/5
100/100 [==============================] - 61s 599ms/step - loss: 0.4275 - accuracy: 0.8850 - val_loss: 0.1099 - val_accuracy: 0.9787
Epoch 2/5
100/100 [==============================] - 60s 599ms/step - loss: 0.0287 - accuracy: 0.9953 - val_loss: 0.1696 - val_accuracy: 0.9750
Epoch 3/5
100/100 [==============================] - 60s 605ms/step - loss: 0.0185 - accuracy: 0.9962 - val_loss: 0.0257 - val_accuracy: 0.9962
Epoch 4/5
100/100 [==============================] - 63s 627ms/step - loss: 0.0078 - accuracy: 0.9978 - val_loss: 0.0254 - val_accuracy: 0.9962
Epoch 5/5
100/100 [==============================] - 60s 605ms/step - loss: 0.0078 - accuracy: 0.9984 - val_loss: 0.0144 - val_accuracy: 0.9987
```

As expected, increasing the layers' size only slightly improve the accuracy, which stays anyway really low (around 50%). But an increase in the number of convolutional layers definitely lead to a high accuracy. Finally, the best model is obtained with 3 convolutional layers. For what concerns the number of neurons, a higher number even worsens the results and doubles the time needed to obtain it. It is around 60 seconds for 32 neurons and over 120 seconds for 64 or 128 neurons.

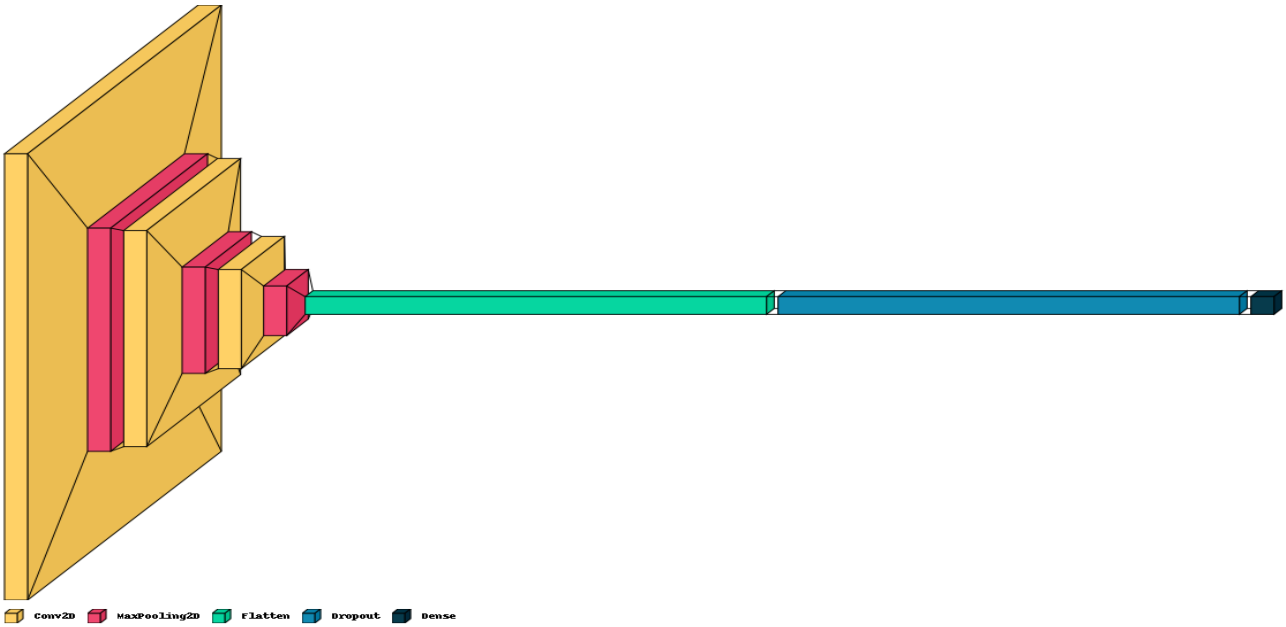
The best accuracy is obtained with 3 convolutional layers and layers' size 32. Once confirmed this architecture I want to further optimize the model. Hence, I add a dropout layer to lower the risk of overfitting. I set the drop out probabilistic rates to 0.5 and 0.7. Then I evaluate two different learning rates with the Adam optimizer [0.01 and 0.001], since in the previous Grid Search I only considered the default 0.01 learning rate.

It comes out that the model with the best performance has a 0.5 drop out probabilistic rate and a 0.01 learning rate within the optimizer. Next, I create the model with the best hyperparameters found so far. I randomly set the number of epochs to 10. Once the model is fit, it is possible to evaluate the performances as the number of epochs changes.



The best number of epochs is 7, where the test and validation accuracies are closer and higher in the graph. Finally, I just train the best model again using the best number of epochs as well. Despite the computational time is still quite high, it is anyway a bit lower then using the random number of epochs.

The final test accuracy is 99.80% and below is a visualisation of the final model's architecture.



3.3 Scale Up with data size

Scalability is achieved by normalizing the input values. Also, the initial reduction phase is important for an architecture able to scale larger datasets. To process larger amount of data we would need greater computational and storage resources. In Google Colab this means to buy a larger memory space and a faster GPU.

3.4 Further possible improvements of the models

To improve the model, we could work on a larger dataset. Obviously, in this case it is a limit placed by the availability of data or on the computational capabilities of the tool. In fact, during the implementation I had to consider a small portion of the entire training set available to speed up the computation. Otherwise, it would have been too expensive and above all too long to execute. Also, we could optimize performance by working more on hyperparameters tuning. This means considering more hyperparameters in the grid search, or a wider range of values for each of them. Any such improvement involves higher computational costs or longer times.
Furthermore, Grid Search is not the only possible alternative for performing the tuning. One major disadvantage is that we may miss good hyperparameter values, because we manually define the search space grid. To extend this range we could use a Randomized Grid Search, which considers a continuous range of uniformly distributed values selected at random. Although the search space is wider, this technique is much more time consuming. Especially if it is applied on many hyperparameters. A good attempt would be to perform Randomized Grid Search over a wider range of values just for the number of layers. Indeed, we know from the literature that enlarge the layer sizes do not improve the performance as does adding new layers. Even here, however, performance in terms of time should be considered.
To solve these issues, we could also distribute computation of the training phase in parallel jobs. TensorFlow has a built-in API that allows this distribution across more machines.

Conclusions

In conclusion, we can observe that several combinations of hyperparameters value could be tested. But the computational resources required increase a lot. To deal with this limitation, a good approach is to rely on the knowledge from the literature to set up the tuning phase. Hence, decide which hyperparameters to tune for a specific use-case and select a range of values compatible with the performance expectative. As demonstrated with the result, this targeted approach can lead to the construction of very accurate models. Moreover, in this project I divided the tuning into two parts. One aimed to set a good network architecture. One aimed at finding optimal probabilistic rate of the drop out layer and learning rate for the Adam optimizer. This subdivision has optimized the tuning work.

Obviously, other values like the number of epochs or batches can be tested. On the other hand, testing a greater number of convolutional layers is not that useful. In fact, a greater complexity of the network leads to a greater computational effort and above all to the risk of overfitting. Given the results obtained with the best model, which test accuracy is 99.80%, we can consider these final settings reasonably good.

Declaration

I/We declare that this material, which I/We now submit for assessment, is entirely my/our 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/our work. I/We 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/us or any other person for assessment on this or any other course of study.