ADM - Image Classification with Deep Learning

Sonia Petrini

June 10, 2022

1 Introduction

The aim of this project is to design a deep-learning architecture to perform binary classification on large amounts of data. In particular, the task is to distinguish comics from real faces. In the following sections we will present the data and its organization, the pre-processing applied to it, the methodology followed to choose the best architecture, and the results obtained by testing it on real unseen data, with a focus on the relevant steps for the scalability of our implementation.

2 Methods

2.1 Data and Data Structures

The dataset used to train and test the neural network is "comic faces", available on Kaggle. It contains 10000 pairs of faces, in the real and the comics version, for a total of 20000 examples. The whole dataset is randomly split in training (2/3) and test sets (1/3), and the former is further divided in actual training (2/3) and validation sets (1/3).

In order to obtain a catalogue relating each example with its label, we create a dictionary where the path is the key and the label is the value, containing the paths to all the examples. Then, we convert this dictionary into a `pandas` data frame, on which we are efficiently able to perform the splitting into training, validation, and test sets. The motivation is that when working with large amounts of data we want to make sure to reduce to the minimum the amount of operations performed on it, thus making use of the paths for labelling and splitting.

Once we have the three distinct set, we make a consideration on how to proceed in the design of the algorithm with respect to the pre-processing (explained in the next subsection) to perform on data: on one hand, from a space efficiency perspective we would avoid keeping large objects in memory, thus accessing the paths and transforming the data of a batch of examples at each training step. On the other hand, this would of course imply a much larger training time, as pre-processing requires some time. Thus, we proceed with only transforming the validation set, which is the smallest, to see the amount of memory occupied by it: approximately 0,000152 MB for 4200 examples. We consider this a reasonable amount of space, even when considering the larger sets; thus, we load and pre-process all the examples, creating for each set a multidimensional `numpy` array with the flattened images, and an array containing the labels.

2.2 Pre-processing

In order to be able to feed the images to the network we need to flatten the pixel matrices by reshaping the arrays. Moreover, a commonly used technique consist in reducing the dimensionality of the data by converting it from RGB to greyscale, and renormalizing the grey tones by dividing the inputs by 225. Since we are working with large amounts of data, and the original size of each image is 1024x1024, we also proceed with downsizing each of them to 64x64: this step provides a great advantage in terms of RAM usage and training time, with hopefully a little loss in terms of prediction accuracy. Later on, we will experiment with size to see if we can decrease it further with a reasonable loss in terms of performance.

2.3 Algorithms

In order to build the architecture of our NN we use a combination of Dense, 2DConvolutional, and Pooling layers. The base architecture contains two fixed Dense layer, one very close to the inputs and the second one very close to the outputs (output classification layer). Each model is compiled by setting *binary cross-entropy* as the loss function to minimize, *Adam* as the optimizer, and *precision* and *recall* as metrics , so that we are able to compute the f-score later on. Then, we make some experiments to find the ideal depth and layer composition of our model.

Parallelization A crucial aspect of our implementation is the parallelization of training via the `multiprocessing` library. In fact, since we will have to run several trainings to find the best architecture, we can save a significant amount of time and make use of all the computational power provided by *Google colab* by using multiple threads simultaneously. This is done by passing to `pool.map()` both the function that implements training and validation, and the list containing the arguments of each of the nets to train. Thus, `pool.map()` will simultaneously run the function fed with each of the different arguments, resulting in a total training time equal to the largest among them.

Random search Since in the scope of neural networks the search space for the tuning parameters can easily reach high dimensionality and we are dealing with massive datasets, performing a grid search would require a very large training time and CPU usage, not necessarily offset by the information gain. Thus, to orient our choice of the parameters with a good balance between time/CPU resources consumption and gained knowledge, we run a *random search* considering 6 (out of 12 possible) unique combinations of the following parameters and their domain subsets:

- *first Dense layer size*: [8, 16, 32]
- *which pooling*: ['average','max']
- *n convolutional layers*: [1, 2]

Moreover, since our aim now is to assess the relative performance between architectures rather than the absolute performance on the whole dataset, we only consider subsets the training and validation sets. To exploit parallelization in this randomized setting we first make sure to identify unique parameters combinations to avoid redundant training, then we pass them to the `pool.map()` function. In particular, we force the 6 nets to contain equal number of 'average' and 'max' pooling for a more balanced comparison. In this sense the search is not entirely random, but it allows us to span the whole domain of the search space without exploring it exhaustively.

Once the architecture is defined, we proceed to train each model fixing batch size to 50, and the number of epochs to 25.

Downsizing further As previously mentioned, keeping the images in their original dimension implies a large training time and RAM usage, while the full definition might not be necessary for obtaining a good classification performance. For this reason, in the pre-processing we reduced size from 1024x1024 to 64x64, and we used the rescaled images performed the random search. Then, once we made sure that we can obtain high accuracy and recall with 64x64 images, we wonder if we could further decrease the resolution while keeping the performance metrics high. Thus, we train the model built with the best architecture identified by the random search on two versions of the data, one where images have size 64x64, and one where they have size 32x32. Thus, we compare the time that we save with the loss in f-score to decide whether one offsets the other. This time, as we want to find the best architecture in absolute, we employ the whole dataset, and we train it for 30 epochs (on a batch size = 35).

2.4 Scalability

Given that it is well know that neural networks only work well with large amounts of data, we need to make sure that our program is able to scale up with it. Thus, we put in place some strategies to guarantee this. First, we perform as many operations as possible on the paths to the images rather than on the images themselves. Then, we first pre-process a portion of the data, to make sure that the transformed data will occupy a reasonable amount of space before transforming all the examples. Thus, instead of loading all data for later applying pre-processing we do this in one step, so that the normalized and downsized structures that we actually keep in memory have a much lower dimensionality. Another important steps concerns not using all the available data when not necessary, as in the case of the random search. Finally, one of the most important strategies is exploiting multiple threads simultaneously through parallelization, which allows to run more trainings at the time cost of the larger time.

3 Discussion

In this section we present the obtained results, and discuss our choices in the design of the final model architecture.

Random search In fig. 1 we display a plot of training time against f-score of both training and validation sets for each of the 6 explored nets. Each net is named according to the following convention: $firstlayersize\_whichpooling\_nconvolutional$. Given the structure of the plot, we aim at identifying the net in the left-most and lower portion of the space, where f-score is high but time is low. By looking at the validation scores, we observe that nets trained with 'max' pooling seem to overall perform better. As an interesting exception, 'max' pooling needs one more convolutional layer to reach the same score of 'average', given the same number of neurons in the first layer, but only when the latter is high enough (32). With a smaller number of neurons (16) in the first layer, 'average' performs much worse than 'max', even when 'max' has only 8 first layer neurons and they both have 2 convolutional layers.

If we now move our attention to the architectures with the highest f-scores on validation, we see that they are both based on 'max' pooling, but have a different balance of convolutional layers and neurons in the first layer. As expected, adding a convolutional layer increases training time, and we thus conclude that we can obtain the same performance as a 2-convolutional layers model in a smaller time by simply increasing the number of neurons in the first Dense layer. Our conclusions from the performed random search are the following:

- 'Average' pooling works better than 'max', but only with a larger number of neurons in the first Dense layer.

- When 'average' is used in combination with only 16 first layer neurons and 1 convolutional layer, its performance is much worse than the same model designed with 'max' pooling. Adding a convolutional layer just makes it worse.
- 'Max' pooling reaches the best performance in the search, both with 8_max_2, and 16_max_1. We thus select the latter as our preferred model, as it has a lower training time for virtually the same f-score.

Given what we have learnt, a possible further exploration could consider a net of the form 16_max_2. However, given that the f-score on validation is already very high, we estimate that the gain in performance that we could possibly obtain might not be justified by the increased training time. In fact, even with only 8 first layer neurons and working on a subset of the training set the elapsed time with 2 convolutional layers is 17 minutes more, meaning that when increasing the first parameter to 16 and considering larger amounts of data this increase in time will grow even further. Thus, we consider 16_max_1 to be the best net, also taking into account the parsimony principle and the need to be able to scale up.

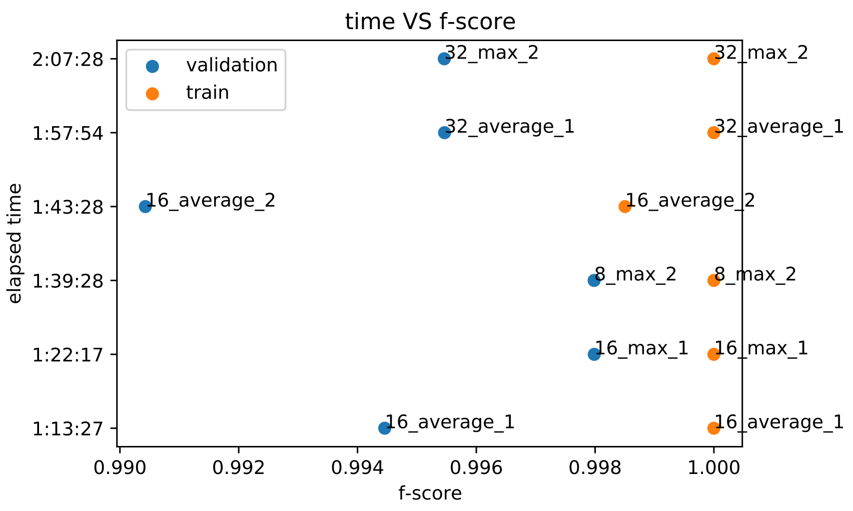

Figure 1: *Random search:* Elapsed time versus f-score. For each net both training and validation f-scores are shown.

Downsizing further As explained in the Methods section, the next step considers the possibility of further rescaling the images to 32x32, to save time in training and RAM usage in the images loading and transformation. Figure 2 shows the resulting time and f-score (again for both training and validation), for the best architecture identified with the random search, where images have size 64x64 and 32x32 (the size has been appended to the net name as defined before). Even if it might look like downsizing the figures has significantly worsened performance, when we compare the percentage decrease in elapsed time to the percentage decrease in validation f-score we find that by reducing size from (64,64) to (32,32), we can reduce training time by 0.549 %, while losing only 0.002 % in validation f-score. In the context of massive data, it's really important to take training time and RAM usage into account, even at the cost of few misclassified examples. Thus, we consider the benefits of rescaling the images of great value for the scope of our project, and we finally identify our best architecture as composed in the following way:

- images of size 32x32
- 16 neurons in the first layer
- one convolutional layer
- a 'max' pooling layer after the convolutional one

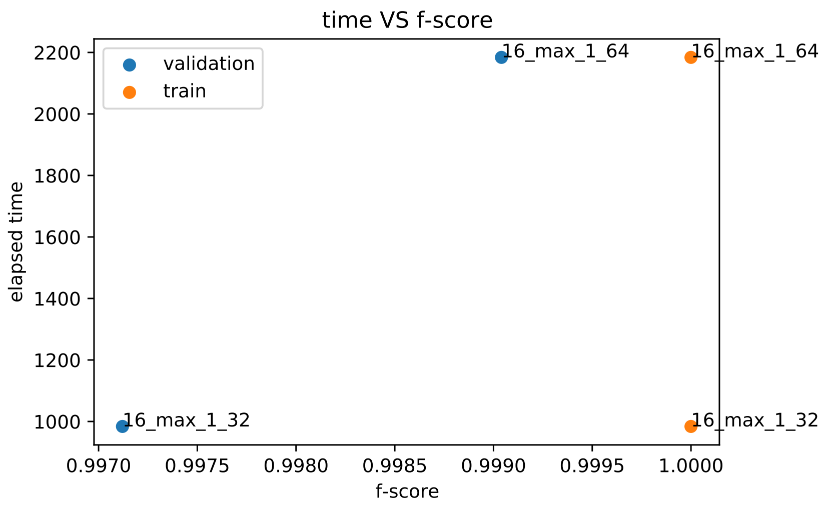

Figure 2: *Size search:* Elapsed time versus f-score. For each net both training and validation f-scores are shown.

Testing the best model The final crucial step of the design of a deep learning model is of course testing. Hence, to see how good our model would perform on unseen data, we evaluate it on the test set, adequately transformed. After 30 epochs, the resulting test scores are:

- Test precision : 0.999
- Test recall : 0.9967
- Test f-score : 0.9978

Despite the decrease in image resolution and the low number of convolutional layers, with the chosen model we are actually able to perform a very accurate classification of faces and comics, having both precision and recall above 99%. The obtained f-score is consistent with the score obtained on validation and it's actually higher, as for training the model we now used the whole training set (including the subset previously used for validation), thus allowing the net to learn from a larger number of examples.

4 Conclusions

We managed to build a model for image classification which scored 0.9978 in f-score on unseen data. First, the best architecture was selected through a random search over number of neurons in the first Dense layer, type of pooling, and number of convolutional layers. Thank to the random search we were able to gain information to orient our parameters choice without training and validating all possible models exhaustively, and to better understand the interactions between the parameters. Thus, we tried to further reduce the training time and RAM usage by reducing the images dimension further. Both experiments were parallelized to allow the program to scale up with data. In conclusion, the chosen best model is a fairly simple one, with only 16 neurons in the first layer and 1 convolutional layer, and 32x32 pixels are sufficient to obtain a very good test performance, with a significant decrease in training time.