Glasses or No Glasses

Github: https://github.com/flaviofuria/GlassesOrNoGlasses
Kaggle: https://www.kaggle.com/jeffheaton/glasses-or-no-glasses

Algorithms for Massive Datasets
Statistical Methods for Machine Learning

Flavio Furia

Academic Year 2020/2021

1 Introduction

This project has been carried out to fulfil an assignment for the "Algorithms for Massive Datasets" and "Statistical Methods for Machine Learning" courses that is based on the "Glasses or No Glasses" challenge from Kaggle[1]. The dataset is made of color images of people that are not real: they have been created with a Generative Adversarial Neural Network[2]. The results are impressive and, in fact, apart from some samples (which will be shown later) it is not so easy to tell that images from this dataset are fake without knowing it in advance. The images are, essentially, portraits of people faces and the task is to determine if a person is wearing glasses or not, in a supervised way, using neural networks.

The Github repository linked in the title page contains all the code produced to satisfy the task: three Python Notebooks (using Python 3.7) with badges on the Github pages that allow to directly execute them on Google Colaboratory have been realized: the first one contains the classic pipeline to face a machine learning problem, with all the trials, analysis and experiments carried out in order to achieve the best possible results; the second and the third one implement the best two models obtained in the first one, focusing on scalability.

An in-depth description of all the reasoning and work done will now follow.

2 Data and Preprocessing

As already said, the dataset is totally available on Kaggle, where it was originally published for a competition; it is made of 5000 samples available in two different types:

- *numerical*: one sample is a latent vector of 512 numbers, each one used by the GAN to produce an image.
- *images*: one sample is a 1024×1024 color image.

Both of them were used separately to train two different neural networks; thus, from now on they will be referred as *numerical dataset* and *image dataset*, respectively, in order to avoid confusion. The data are easily downloadable via the Kaggle API: a username and an associated key are all that is needed to get the data directly within the Python Notebook. The result of downloading is a .zip archive that is easy to unzip: it contains a folder with all the .png images and two .csv files that compose the training and test set of the numerical dataset.

(a) Glasses. (b) No glasses.

Figure 1: Two samples drawn from the image dataset.

[1] https://www.kaggle.com/jeffheaton/glasses-or-no-glasses [2] https://en.wikipedia.org/wiki/Generative_adversarial_network

2.1 Numerical Dataset

As already said, the task has been faced in a "classical" way, using the usual Python libraries for data analysis and machine learning and in a "distributed" way, using Pyspark[3]: these two approaches led to two different implementations (even for the image dataset). It is important to point out that, since there is not a "right" or "best" way to deal with these kind of tasks per se, referring to the following approach as "classical" is being done just to highlight the difference with the Pyspark-based approach.

Classical Pipeline Being saved in .csv format and organized in a tabular way, an ideal choice to store the data from the two files is, of course, the *DataFrame* data structure by Pandas[4]; in this way two tabular datasets, with 514 columns each, were obtained: one *id* column, one column for each of the 512 features, named $v1, v2, \ldots, v511, v512$ and one column named *glasses* with values 1.0 or 0.0 that obviously indicates if that person is wearing glasses or not. It was decided to concatenate these two datasets (and perform a random train/test split in a subsequent moment) into a single one.

The first sample in the numerical dataset is also the first image in the folder and the same holds for the remaining samples/images; thus, it is possible to look an image and check the associated label. While having a look at them, some issues arised:

- it was clear that some associations are wrong, i.e. some samples are labeled with a value of 1 while the person has no glasses and others with a value of 0 while the person is wearing a pair of glasses;
- being created by a neural network, images can present some "noise", making them hard (or impossible) to be classified. 37 of this kind of images were found; they will be referred to as images with an *undefined label*;
- the *glasses* column with the labels was missing in all the test samples.

Figure 2: Images with an undefined label.

For this reason, a correction step was required; it was done by manually checking every label/image pair, eventually changing the wrong labels and adding the missing ones for the test samples. The label column for images with undefined labels was left empty. The corrected label column was written in a .csv file and directly stored in the Github repository(here). It was then used to replace the *glasses* column of the dataset.

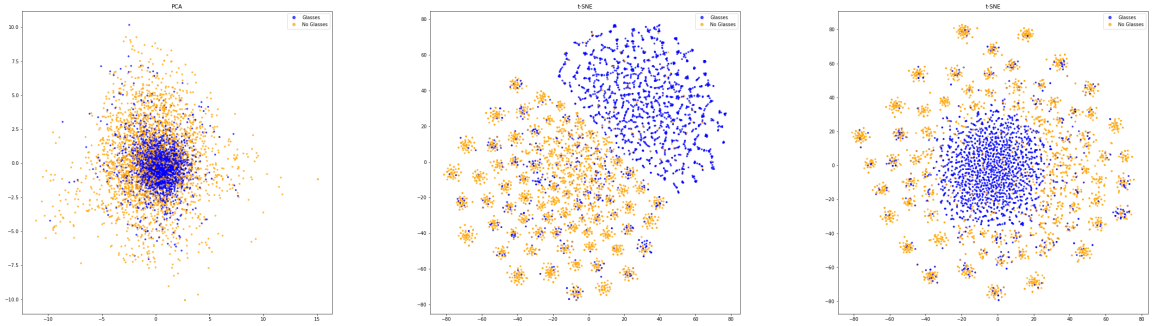
Many real-world applications that are dealt by machine learning exhibit an issue, that can be very hard to face, i.e. class imbalance: this happens when in a classification task, often a binary one, the number of positive and negative samples is dissimilar (as one could expect, for this work, people wearing glasses will

[3] https://spark.apache.org/docs/latest/api/python/ [4] https://pandas.pydata.org

be considered to be the positive samples). Predicting if an individual could have a particular rare disease is a very challenging task mostly because the low number of examples of people with that disease makes it almost impossible to learn the features of that particular class. Imbalance can be faced in different ways but, fortunately, it wasn't necessary since the ratio of people wearing glasses (ignoring those with undefined labels) was 0.558 with respect to the total number of samples.

At this point it was reasonable to look at the data from another point of view: it was done by using *dimensionality reduction*, a technique that allows to map a high-dimensional space into a low-dimensional space, with the idea (and the hope) of extracting the crucial information embedded within the data leaving out the less important features. This process is known as *feature extraction* and it is very useful when dealing with high-dimensional datasets. Another interesting task this technique can help with is *visualization*; the idea is to "shrink" down the dimensions of the datasets usually to 2 or 3 and visualize them with some plotting technique, without touching the real dataset. In this way meaningful properties of the data, like the degree of correlation between samples and labels, could be observed.

After some few early experiments, it was chosen to use dimensionality reduction only for visualization; two different techniques were used, the first one being *Principal Component Analysis*[5] (PCA), a linear transformation that maps high-dimensional points into a lower-dimensional representation, trying to preserve the maximal amount of variance, with the idea that greater variance is followed by a greater amount of information. In particular, the Sklearn[6] implementation was used, together with the Matplotlib[7] library to give a graphical representation of the results, which are shown in 3a: blue (orange) points are samples of people wearing (no) glasses, thus it is easy to see that there is a cluster of positive samples around the center of the plot, suggesting that there actually is some kind of relation between samples and labels. However, the two classes are far from being well separated, as there are blue points out of the cluster and orange points within the cluster.



(a) PCA. (b) t-SNE. (c) t-SNE after PCA.

Figure 3: Dimensionality Reduction.

The other dimensionality reduction tool applied is *t-Distributed Stochastic Neighbor Embedding*[8] (t-SNE); while PCA tries to keep distant points in the high-dimensional space also distant in the reduced space, this technique aims at keeping close points in the original space also close in the low-dimensional space. The 3b and 3c figures show the results of this algorithm, the first applied on the standardized numerical dataset and the second on the same dataset already reduced to a dimension of 50 with PCA (it is good practice when the dimensionality is high): with this two representations the relation between labels and samples is even more clear, as positive and negative points are almost all well separated. It is interesting to see that the blue points, representing people wearing glasses, are widespread without a particular pattern (to be precise, in the second case they're clustered around the center), while orange points are subdivided in dozens of little clusters; As it happens with PCA, there are points of one class inside the clusters of points of the opposite class, thus maybe these are the actual samples that will be hard to classify in the right way. However, the

[5] https://en.wikipedia.org/wiki/Principal_component_analysis
[6] https://scikit-learn.org
[7] https://matplotlib.org
[8] https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding

non-linearity of t-SNE has shown to be very helpful for visualizing the data.

A crucial part of the data preprocessing/preparation step is that of feature scaling, i.e. making all the features respect the same scale of values so that none of them can dominate over the others during the training process, thing that could lead to poor results. In particular, *standardization* was the chosen approach: it consists on scaling all the values in such a way that every feature, after being scaled, has a mean of 0 and a standard deviation of 1. However, *StandardScaler* by Sklearn was used to learn the mean μ and the standard deviation σ parameters of every training set feature and then the transformation was applied also to the test set using these values: not including the test set when computing means and standard deviations of the features makes this set as unseen as possible, as it should be. This choice required the application of a random train/test split (with a ratio of 0.8/0.2) on the dataset before standardizing; the validation sets were not manually created, as Keras[9] allows to automatically split the training set during the training. All of this, was done after converting both the samples and the labels to Numpy[10] arrays, in the end having 3970 samples for the training set and 993 for the test set.

Pyspark-based Pyspark is, as one should expect, a Python interface for Apache Spark. The main idea behind the creation of a pipeline of this kind to deal with the numerical dataset (and the image one, too) was to think of a situation in which the dataset would be much larger: in this case, since mantaining all the data in main memory is hardly impossible, distributing the computation across a cluster of machines becomes crucial.

The main data structure provided by Spark to store a collection of data is the RDD (Resilient Distributed Datasets), an immutable distributed collection of objects; each dataset of this kind is divided into logical partitions that can be processed in parallel. Nevertheless, in Spark 1.3 version, Dataframes were introduced: born in order to overcome the limitations of RDDs, a Spark Dataframe is a distributed collection of data organized in named columns, thus being more adapt when working with structured data. That being said, DataFrame seemed to be the right choice to store the data.

When creating a dataframe, Pyspark provides the possibility to specify the *schema*, i.e. the structure of the data that are being stored: the two .csv files were stored in two separate dataframes, in which the id values were considered as integers, while the labels and all the 512 features were stored as double, and then they were concatenated. After storing the corrected labels too in a dataframe, an inner join operation on the id column between this and the final dataframe followed; join is a very heavy operation and it had to be done only because the initial provided labels were incorrect/missing in some samples.

At this point, the main concepts of the classic implementation were reproduced relying on Pyspark functionalities. In particular, dimensionality reduction and standardization were implemented using Pyspark *pipelines*, i.e. sequences of stages that, starting from a set of features, allow to assemble them in a single column and apply a set of *transformations/actions* to the whole column (thus to the entire dataset at once). A transformation is a function taking an RDD/dataframe and producing one or more RDD/dataframe as output; they are lazy, so they are executed only when an action is called. Unfortunately, t-SNE and visualization tools are not available on Pyspark and required to pre-convert the data to Numpy arrays and opt for their Sklearn/Matplotlib implementations. For these two, in a situation in which one is dealing with a larger dataset, it could be good practice to just take a random sample of the dataset and still apply t-SNE on that sample (assuming that it is representative of the entire population under analysis).

In the end, after a random shuffle, train and test dataframes were created randomly splitting the original one with the same 0.8/0.2 ratio. Then, as it was done before, train and test data were standardized, fitting the scaler only on the training set to leave the test set as untouched as possible.

2.2 Image Dataset

For this dataset the 5000 samples, contained in a separate folder of the unzipped archive, are provided as 1024×1024 color images in .png format, with each of the $1024 \times 1024 = 1.048.576$ pixels having 3 values, one for each RGB channel. Since processing an image of this size would be extremely both time and space consuming, it was decide to resize them to 128×128 and the RGB model was abandoned in favour of its grayscale counterpart, in which each pixel is a single value representing the luminous intensity. For this dataset, only one preprocessing step was applied: normalization. Since pixels can have values from 0 to 255,

[9] https://keras.io/ [10] https://numpy.org

it was achieved by simply dividing each pixel value by 255, in order to have new values ranging from 0 to 1. Of course, as for the numerical one, also this dataset was filtered, removing the undefined samples.

Classical Images were imported using Pillow[11], one of the most popular Python packages for managing images. Other attempts were done (even using a Pillow fork that uses SIMD[12] processing), but in all of them the required amount of time was about the same (about 4/5 minutes using on Colaboratory). This choice also allowed to easily resize and convert to grayscale and, in the end, turn them to Numpy arrays.

Pyspark-based In this case, dealing with images required slightly more effort. When reading from the filesystem, Pyspark provides various types of data format to choose from: for images, the obvious choices are *image* or *binaryFile* formatsthe second one was chosen. At first, the *image* format sounded as the obvious choice but, after some experiments, the *binaryFile* format also proved to be effective, and it was preferred.

Both the format, however, provides the path of the files along with the byte arrays (i.e. the content of the file): paths are important because, as already said, they contain the actual id of the image they are associated with. For this reason a UDF (user defined function) was used to extract ids from paths and the same inner join operation for the numerical dataset was applied to associate corrected labels to images. Starting from the byte arrays, one last UDF was in charge of decoding it to get the actual image, resizing and converting it to grayscale (all using the OpenCV[13] package), then normalizing it and, in the end, returning it as a *DenseVector*, a Pyspark data type to deal with array values.

3 Models

All the process of data preparation and preprocessing was followed by an investigation of the neural network classes that were best suited for the kind of data available. It was decided to opt for *Multi-Layer Perceptron*[14] (MLP) for the numerical dataset, while *Convolutional Neural Network*[15] (CNN) was the choice for the image dataset. All the neural networks, in the classical pipeline, were implemented using Keras[16] on top of Tensorflow 2[17], as the combination of the two provides both ease of use and customization.

In the Pyspark implementations, in order to bring distribution into the world of deep learning, one additional Python library was required, i.e. Elephas[18]: given a Keras model, distribution is obtained by serializing the model, sending it to the *workers*(or *executors*, processes in charge of running individual tasks in a given Spark job), let them train their chunk and send gradients to the *driver* (the process running the *main()* method). A master model on the driver uses these gradients together with an optimizer in order to update its weights.

3.1 Multi-Layer Perceptron

Just to remind the main concepts, a neural network is a directed graph with nodes being neurons and edges being links between them. Neurons are divided in three sets: *input* neurons, which receive information from the external world, *output* neurons which returns back information to the external world and *hidden* neurons, which can only communicate with other neurons. If the graph underlying a neural network is acyclic, it is referred to as *feedforward* neural network. Multi-Layer Perceptrons are feedforward neural networks organized in a layered structure with an input layer, an output layer and 0, 1 or more hidden layers. There can be a link between two neurons if and only if they belong to consecutive layers.

Four different MLPs have been built for solving the task; every network has, of course, an input layer with 512 (the number of features) neurons and an output layer made of a single neuron with a *sigmoid* activation function (binary classification can be faced with a single neuron that outputs 1 for positive samples and 0 for negative ones), while all the neurons of every non-output layer had *ReLU* as activation function. The loss function used for all the networks was *binary crossentropy*. The first MLP is very simple, having two hidden layers with 16 and 8 neurons, respectively and *Adam* as optimizer with the default parameters provided by Keras, thus with a learning rate of 1e-3. It was created as a first try to understand the next step to take.

[11] https://python-pillow.org [12] https://it.wikipedia.org/wiki/SIMD [13] https://opencv.org
[14] https://en.wikipedia.org/wiki/Multilayer_perceptron [15] https://en.wikipedia.org/wiki/Convolutional_neural_network
[16] https://keras.io [17] https://www.tensorflow.org [18] https://github.com/maxpumperla/elephas

Overfitting is the condition arising when a neural network performs very well on samples drawn from the training set but fails to generalize, behaving poorly on samples that has not seen yet; one of the main cause of overfitting is a great disparity between the size of the training set and the number of parameters of the network (with the latter being larger than the first). Thus, reducing the number of parameters often is a good idea when a network is struggling with overfitting, and *dropout* is one of the regularization techniques going in this direction: it consists in randomly "switching off" some of the neurons during training, in order to reduce the number of parameters. The employment of dropout to the first hidden layer is the only difference between the first and the second MLP; every neuron of that layer has a probability 0.4 of being omitted during training. As it will be shown in the next section, it was done in order to reduce overfitting. Adam was the optimizer, with default parameters.

In the third MLP more parameters were added, together with more regularization: all the 16 and 16 neurons of the two hidden layers have a probability of 0.4 and 0.3, respectively, to be switched off. Moreover, in the second hidden layer, *l1* and *l2* regularization parameters have been introduced, both with a value of 1e-5 : these terms penalize the weights of the networks in order to fight overfitting, reaching in slightly different ways the same same objective. In this case, the learning rate was reduced down to 1e-4: using a lower value means taking smaller steps while computing the gradients, reducing the speed of convergence and, hopefully, increasing the chances of reaching points of global minimum. This MLP was the one being also implemented in the Pyspark implementation.

The last model followed a different approach: the architecture and the choice of some hyperparameters was automatically done with the employment of *Bayesian Optimization*[19], an optimization technique that, given a search space, finds the best possible combination of hyperparameters in that space that optimizes an unknown objective function that is treated as a random function.

The algorithm was in charge of searching for the best number of layers and, for each one of them, the optimal number of neurons (ranging from 8 to 256), eventually applying regularization (and how much of it to apply) and choosing the activation function. The loss and the optimizer were fixed to be binary crossentropy and Adam but, for the latter, the choice of learning rate and its other parameters were left to the algorithm. Bayesian Optimization is a quite expensive task and was employed with the hope of not obtaining a model with much better performances with respect to the "manually tuned" ones: this would mean that applying this heavy algorithm was not really necessary to solve the required task and just served as a double check of all the reasoning made during the analysis.

Layers	Units	Activation	Notes
Input			shape=512
Dense	16	ReLU	
Dropout			0.4
Dense	16	ReLU	$l1 = 1e-5$ $l2 = 1e-5$
Dropout			0.3
Dense	1	sigmoid	

Table 1: Architecture of the third MLP.

3.2 Convolutional Neural Network

Convolutional Neural Neworks are feedforward networks with the capabilities of adaptively learning the spatial hierarchies of features, thing that makes them one of the best choice to deal with images. One of the building blocks of CNNs are, of course, *convolutional* layers: these are layers in which, as it happens in the animal visual cortex, every neuron takes input only from its *receptive field*, i.e. only from a small portion of the previous layer's neurons and this is achieved thanks to convolution, a linear operation that computes the element-wise product of each element of a *kernel* and a given input and, thus extracting features from small regions of the input. *Pooling* layers, instead, are employed in CNNs in order to downsample the input, combining together outputs of regions of neurons: in *MaxPooling* layers, for example, given a small portion

[19] https://en.wikipedia.org/wiki/Bayesian_optimization

of neurons as input, the output of a neuron of this layer would be the maximum of the input values. After the last convolution/pooling layer, input is usually flattened and fed into a sequence of fully connected layers.

Only one CNN was built, characterized by one bidimensional convolutional layer with a 3×3 kernel, 16 output filters in the convolution with ReLU as the activation function followed by a 2×2 bidimensional max pooling layer; after that, there is another 2d convolutional layer with a 5×5 kernel with 32 filters with the same activation function) followed by the same max pooling layer. The output of this layer was flattened and given as input to two last hidden layers of 32 and 16 neurons (boh with ReLU activation function) ending in the usual single output neuron with a sigmoid activation function.

For the Pyspark implementation, everything was the same, with only one slight difference: since samples were all stored as DenseVector (which are unidimensional) and CNNs require input with a dimension of $width \times height \times channels\_num$, a simple *reshape* layer was put first.

Layers	Units	Kernel	Activation	Notes
Input				shape=128x128x1
Conv2D	16	3	ReLU	
MaxPooling2D				strides=2
Conv2D	32	5	ReLU	
MaxPooling2D				strides=2
Flatten				
Dense	32		ReLU	
Dense	16		ReLU	
Dense	1		sigmoid	

Table 2: Architecture of the CNN.

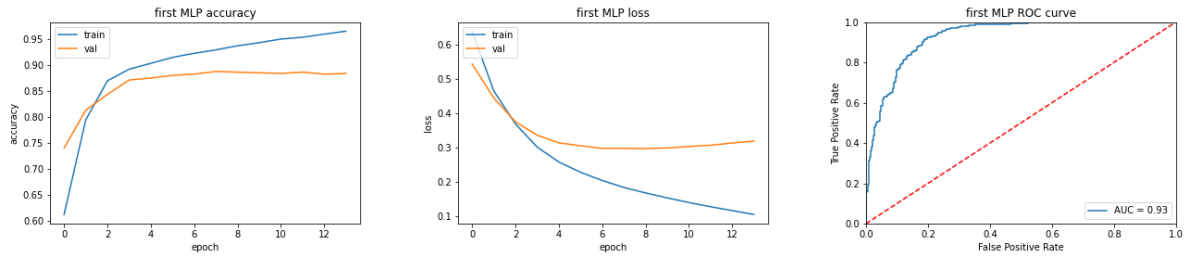
4 Experiments and Results

All the experiments were carried out within the Python Notebooks using the computational resources provided for free on Google Colaboratory, thus resulting in slower computation (especially for the CNNs) but at the same time giving a clear representation of the experiments and results for each single trained model.

All the models were trained using two specific Keras *callbacks*; the first one, *early stopping*, lets the training automatically stop 0, 1 or more epochs after a chosen metric start decreasing: *validation loss* was the choice, since the validation set is the closest thing to unseen data inside the training context. The creation of the validation sets was automatically done inside the *fit* function, splitting the training set with, again, a ratio of 0.8/0.2. The second callback, *model checkpoint*, was used in order to save the network weights produced in the best epoch according to validation loss; the scores and evaluations were then carried out with that specific combination of weights.

4.1 Numerical Dataset

The first MLP was trained for 50 epochs and with a batch size of 128. The scores are not terrible but the model clearly suffers from overfitting: the training stopped very early (12 epochs) and just after the first epochs the accuracy and loss curves (4) on the training and validation sets bifurcate and then move away one another. Overfitting can also be seen in the final scores, with the training accuracy being 0.937 while the test one is 0.869 (in the 8-th epoch). The loss is always one of the most important metrics to look at in order to check the performance of a classifier; accuracy, instead, can be misleading when the dataset is imbalanced: a classifier with the 95% of negative samples could easily achieve an accuracy of 0.95 by just classifying each sample as negative. Even though this is not the case, it was still decided to compute on the test set the *ROC curve* which compares the *true* and the *false positive rate* and measures, together with the *AUC* value (the area under the curve), the capability of the classifier to distinguish positive from negative samples at various thresholds. The curve for the first MLP in 4c shows that the classifier does quite good at discriminating: the dotted red line shows the performance of a predictor that does random guessing,

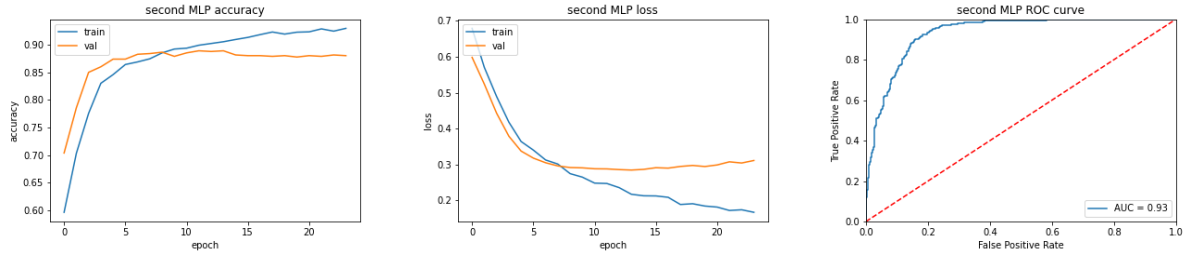


(a) Accuracy. (b) Loss. (c) ROC curve.

Figure 4: Scores for the first MLP.

anything above this line is bad and can be turned into good just by reverting the predictions, while the best is achieved by trying to reach the left corner of the plot, with the point $(0, 1)$ representing perfection. For all the classifiers, the ROC curve was computed only for the test set, in order to show the performance of the classifier in discriminating positive from negative only on unseen data.

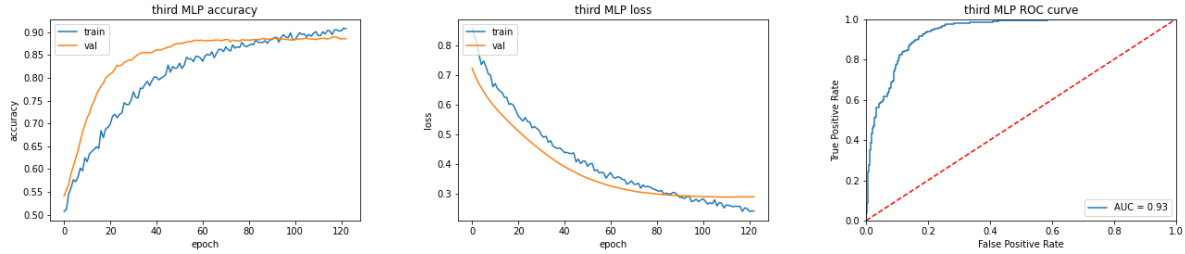
In the second model, that was trained with the same number of epochs and batch size, dropout was introduced, with the aim of reducing overfitting: it happened in some way, but overfitting still arised in about 6/7 epochs (5). Regularization caused slight increase of the test accuracy to 0.878 but, at the same time, it also caused the decrease of training accuracy to 0.905. The ROC curve is mostly similar to the one from the first MLP.



(a) Accuracy. (b) Loss. (c) ROC curve.

Figure 5: Scores for the second MLP.

The third model was built not to reduce overfitting, but to demolish it and dropout on the second hidden layer with the employment of l1 and l2 regularizers managed to achieve this task: the curves in 6 clearly shows that overfitting is not a thing anymore. Unfortunately, the scores didn't improve much: the only noticeable thing is that the test loss slightly decreased, reaching 0.307.

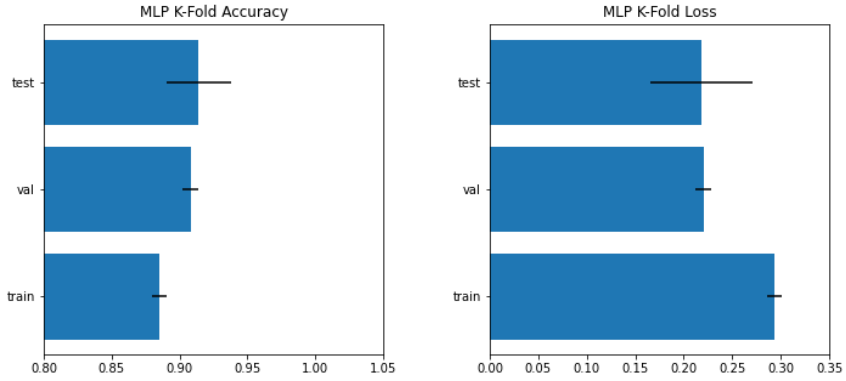


(a) Accuracy. (b) Loss. (c) ROC curve.

Figure 6: Scores for the third MLP.

The sequence of results obtained with these three models was suggesting that maybe there was no much

room for improvements and Bayesian Optimization was employed to check if there was a better combination of hyperparameters that has not yet been considered: 10 trials with 5 executions for each trial searching for the best combination in the search space to minimize the validation loss ended up with a model with a validation loss of 0.299. Training the MLP obtained with this combination of hyperparameters confirmed that this model was not the best, having a test loss of "just" 0.33.



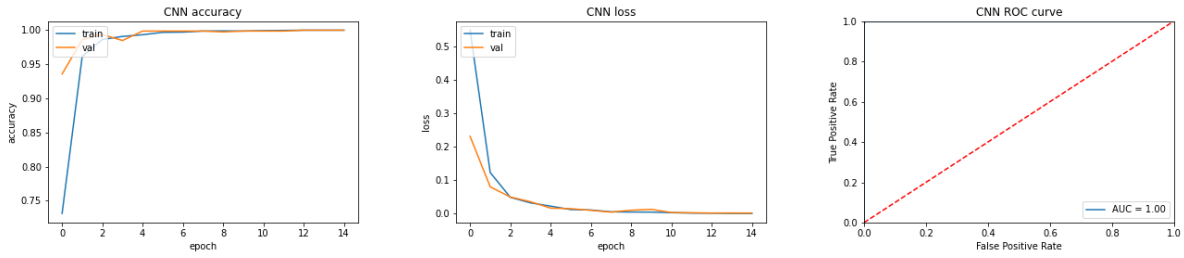
(a) Accuracy. (b) Loss.

Figure 7: K-fold Cross-Validation on the third MLP.

The last step of the experiments on the numerical dataset was to apply *K-fold cross-validation*, with the aim of evaluating the generalization capabilities of the best model: even though the second and the third one had very similar scores, the latter was chosen to be the "best" one, since it was a slightly better loss on the test set. Given a dataset, K-fold cross-validation is achieved by partitioning it in K subsets and then for K times training the model on $K-1$ subsets and testing it on the remaining one, obviously each time choosing a different one. Then, all the scores should be averaged, in order to have a double check of how the model can generalize when dealing with unseen data. For $K = 10$, even if in 7 it can be seen that, especially for the test set, there is indeed some deviation among the various folds, the averaged scores obtained with the cross-validation showed that the model does not suffer much when facing "new" data.

As already said, only the third MLP was trained in the Pyspark implementation, with the exact same hyperparameters. The split ratio among train, validation and test sets was also the same but with high probability the resulting sets were different. For this reason and also because each worker computes its gradients and then sends them back to the driver node, which in turn uses them to make an overal computation, the scores are not exactly the same, but still seems reliable.

4.2 Image Dataset

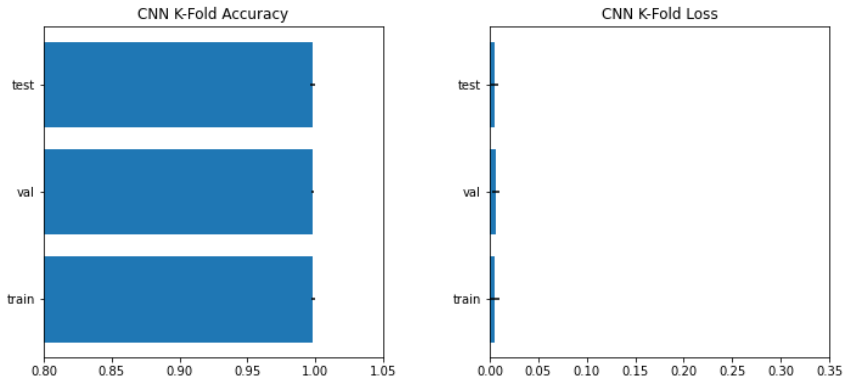


(a) Accuracy. (b) Loss. (c) Loss.

Figure 8: Scores for the CNN.

For this dataset the situation was totally different: one model was enough to obtain outstanding scores. The CNN was trained for 15 epochs, even though less than 5 were enough to reach great results, with a batch size of 64 and the usual automatic 0.8/0.2 training/validation split. This classifier hardly makes any mistake, reaching 1.0 accuracy on the training and validation sets and 0.996 on the test set; the loss is close to 0.0 and the area under the ROC curve is 1.0, which means that the predictor is close to perfection.

In the same way as it happened for the third MLP in the numerical dataset, K-fold Cross-Validation was applied (but K was lowered to 5, because CNNs are much longer to train): the results are still great and the deviation among all folds is almost irrelevant.



(a) Accuracy. (b) Loss.

Figure 9: K-fold Cross-Validation on the CNN with $K = 5$.

5 Conclusion

This work has shown that trying to predict if people generated by a GAN are wearing glasses or not is a quite challenging task. MLPs fed with the latent vectors used to generate the images gave decent results but clearly suffered from overfitting; having more data, deeper networks could be employed and, in this way, maybe the scores could improve, without risking the arise of overfitting.

Training on the actual images with CNNs, instead, even without putting great effort into preprocessing and hyperparameter tuning, proved to be very succesful, reaching great scores even on unseen data and without requiring a very deep nor complex network.