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

Declaration

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

This project comes as an assignment for the "Algorithms for Massive Datasets" and "Statistical Methods for Machine Learning" courses. The dataset, taken from the "Glasses or No Glasses" project from Kaggle[1], 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, consisting in three Python Notebooks (version 3.7): in the first one the problem has been faced using the classic machine learning pipeline and tools; this notebook contains all the data analysis and preprocessing, together with the choices of network architectures and trials. Since the data comes in two different types, the second and the third focus on one type each and build the best model found in the first notebook for that data type, but using Pyspark to keep an eye on scalability. In the main page of the repository three badges that allow to directly execute the notebooks on Google Colaboratory are provided.

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

2 Data and Preprocessing

Two datasets are available, each one containing the same 5000 samples but provided 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 network architectures; 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 the associated key are all that is needed to get the data directly within the Python Notebook. After the download, a .zip archive is obtained: 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], a Python interface for Apache Spark: these two approaches led to two different implementations (the same stands 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, the *DataFrame* data structure by Pandas[4] seemed to be a proper choice to store the data from the two files; two tabular datasets were obtained, with 514 columns each: 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, in some images it is not clear if glasses are present or not, making them hard (or impossible) to classify. 37 images of this kind were found (two examples in 2); they will be referred to as images with an *undefined label*;
- the *glasses* column with the labels is missing in all the test samples.

Figure 2: Images with an undefined label.

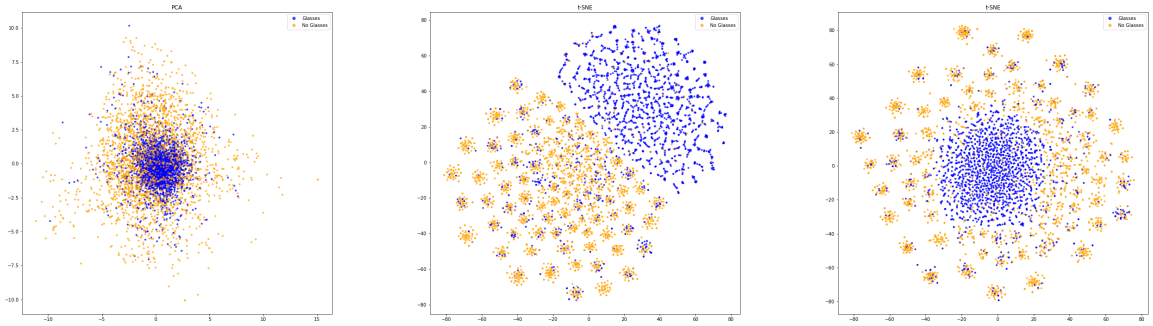
For this reason, a correction step followed: it was done by manually checking every label/image pair, eventually changing the wrong labels, adding the missing ones for the test samples and leaving the column empty for the undefined ones. The corrected label column was written in a .csv file and directly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

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

Many real-world problems that machine learning is able to solve exhibit the issue of class imbalance, that can be very hard to face: 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 are considered as 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 was not necessary since the ratio of people wearing glasses (ignoring those with undefined labels, since they had already been dropped) was 0.558 of the total.

In order to look at the data from another point of view, hoping to get some visual information, *dimensionality reduction* was applied: it is 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 also 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. Being non-linear, it is harder to perform but it often gives better results than PCA. 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

[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

people wearing glasses, are widespread without a particular pattern (to be precise, in the second case they are 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 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. All of this, was done after converting both the samples and the labels to Numpy[9] arrays, in the end having 3970 samples for the training set and 993 for the test set.

Pyspark-based The Pyspark-based pipeline was created in order to deal with an hypothetical case in which the dataset would have been such large that it could not fit in a single machine, making distributing the computation across a cluster of machines 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 a good 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; in the Pyspark context *join* is a very heavy operation and it had to be done only because the initial provided labels were incorrect/missing in some samples. In the end, the undefined samples were dropped from the dataset.

At this point, the main steps taken in 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

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,

[9] https://numpy.org

in which each pixel is a single value representing the luminous intensity. For this dataset, only one more preprocessing step was applied: *normalization*. Since pixels can have values from 0 to 255, 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[10], one of the most popular Python packages for managing images. Other attempts were done (even using a Pillow fork that uses SIMD[11] processing), but in all of them the required amount of time was about the same (about 4/5 minutes on Colaboratory to import all the dataset). 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 one needs to read from the filesystem, Pyspark provides various types of data format to choose from: for images, in particular, *image* and *binaryFile* formats are given. The second one was chosen, leading to a Pyspark DataFrame containing the path of each file and its bytearray, i.e. the image itself.

In this case the file paths were crucial because, as they contain the actual id of the image they are associated with, they had to be extracted to subsequently link images with labels. For this reason a UDF (user defined function) was created and executed to extract ids from paths. The ids were then used to apply the same inner join operation of the numerical dataset to associate corrected labels to images. Starting from the byte arrays, another UDF was in charge of decoding it to get the actual image, resizing and converting it to grayscale (all using the OpenCV[12] package), then normalizing it and, in the end, returning it as a *DenseVector*, a Pyspark data type to deal with array values. The aim of UDFs is taking a set of normal Python operations and making them "scalable"; of course, since it comes with a cost in terms of efficiency, they are advisable only when the operations are not reproducible with the Pyspark available tools.

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*[13] (MLP) for the numerical dataset, while *Convolutional Neural Network*[14] (CNN) was the choice for the image dataset. All the neural networks, in the classical pipeline, were implemented using Keras[15] on top of Tensorflow 2[16], 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[17]: 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 back 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 a *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 and a neuron can only point to a neuron of the subsequent layer.

Four different MLPs have been built for solving the task; The input layer of every network is made of 512 (the number of features) neurons, while the output layer has a single neuron with a *sigmoidal* activation

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

function. Thus, the output of each network is a value between 0 and 1 that can be interpreted as the likelihood of the sample to belong to the positive class. All the hidden neurons, instead, have a ReLU activation function, which is often used in practice since it combines good results with fast computation.

The choice for the loss function of every network fell on *binary crossentropy* (also called *log loss*), which is the negative average of the logarithm of the probability that each sample belongs to the right class. In formula:

$$Loss = -\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)$$

with m being the number of samples for which the loss is being computed, y_i the correct label and $\hat{y}_i$ the predicted one for the i-th sample.

The **first MLP** is very simple and was created as a first try to understand the right next step to take. It is made of two hidden neurons with respectively 16 and 8 neurons. The choice for the opimizer was *Adam* with the default parameters, in particular a *learning rate* of $1e-3$. Adam (Adaptive Moment Estimation) is an optimization algorithm that combines Stochastic Gradient Descent with Root Mean Square Propagation, since it uses squared gradients and momentum to adapt the value of the learning rate. It is often used because it converges fast while giving good scores.

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). This leads to predictors with good scores for the training set but worse ones for the test set (unseen data). Thus, reducing the number of parameters is usually 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.

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, trying to reach in slightly different ways the 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 while, hopefully, increasing the chances of reaching better points of local minima. Since it is the one with the best overall scores, even for a slight amount, as shown in the next section, this MLP was the one being implemented in the Pyspark implementation for the numerical dataset.

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.

The last model, named **bayesian MLP**, followed a different approach: the architecture and the choice of some hyperparameters was automatically done with the employment of *Bayesian Optimization*[18], 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.

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

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.

This technique is very powerful but at the same time quite expensive to perform and was employed to understand if all the analysis and choices were going in the right direction: if the performances of the best model according to Bayesian Optimization were not much better than that of the "manually tuned" ones, it would have meant that applying this heavy algorithm was not really necessary to solve the required task.

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 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, another 2d convolutional layer with a 5×5 kernel with 32 filters (with the same activation function) is followed by the same max pooling layer. The output of this layer was then flattened and given as input to two last hidden layers of 32 and 16 neurons (both with ReLU activation function) ending in the usual single output neuron with a sigmoidal activation function.

This was the architecture used for the Pyspark implementation on the image dataset; 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.

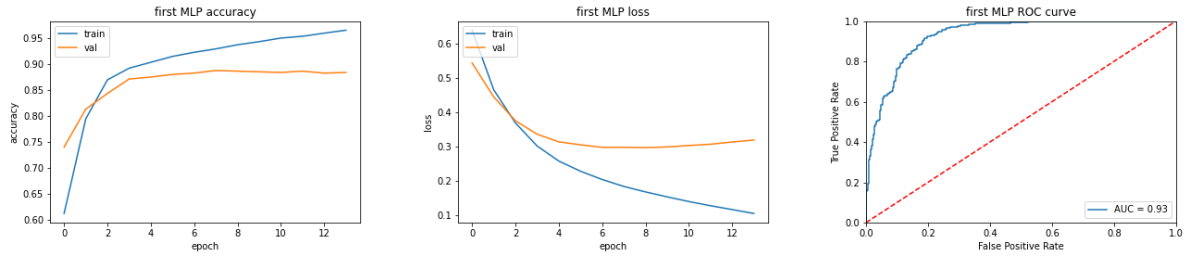
4 Experiments and Results

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 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. For this reason, it was required to let Keras automatically split the training set with a ratio of 0.8/0.2 to obtain a validation set. It was decided to lose some training examples in order to stop the computation in the best moment and, in particular, to choose the best combination of weights for this set. Being a surrogate of the test set, it is likely that those weights would also perform well on the test set.

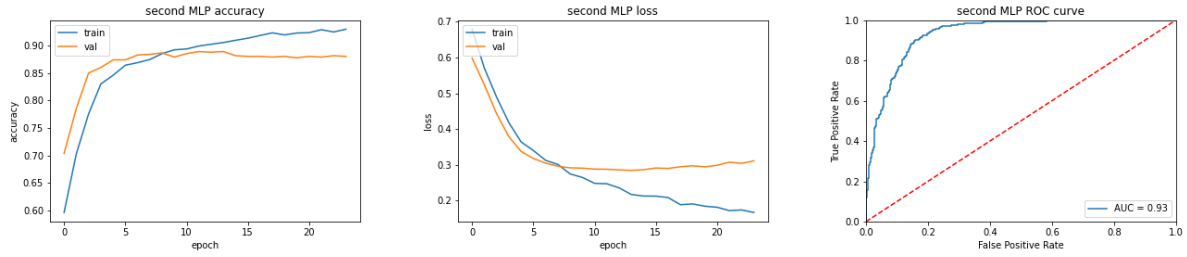
4.1 MLPs on the Numerical Dataset



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

Figure 4: Scores for the first MLP.

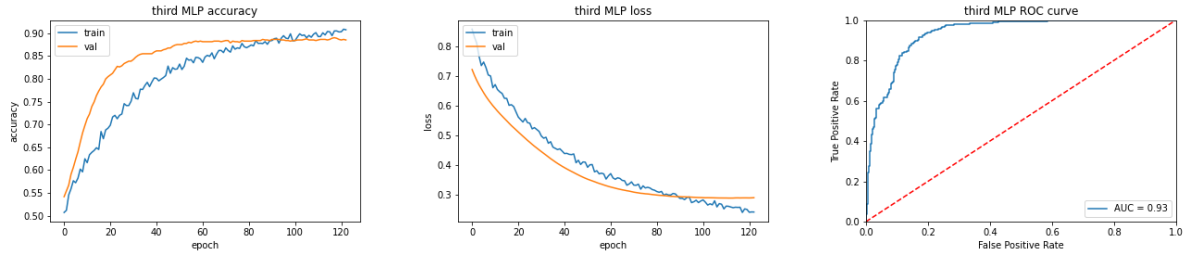
The first MLP was trained for 50 epochs and with a batch size of 128. The scores are not terrible but the model clearly suffers from overfitting: the training stopped very early (12 epochs) and just after the first epochs the accuracy and loss curves (4) on the training and validation sets bifurcate and then move away one another. Overfitting can also be seen in the final scores (3), with the training accuracy and loss being 0.937 and 0.169 while the test ones are 0.869 and 0.323 (in the 8-th epoch), respectively. 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 (because there is no clear imbalance), 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, anything below 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.



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

Figure 5: Scores for the second MLP.

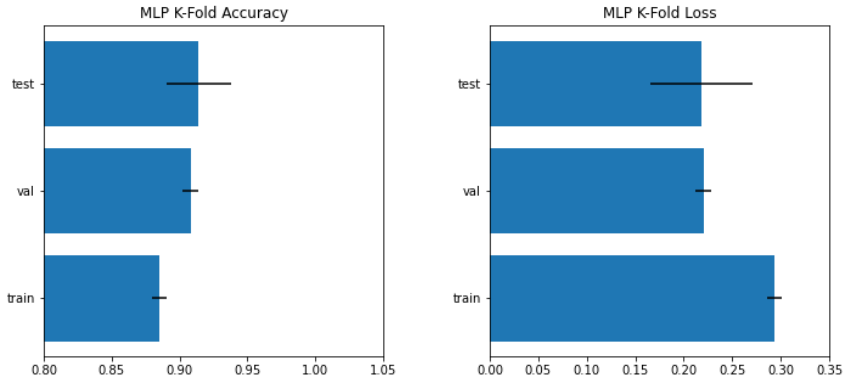
In the second model, that was trained with the same number of epochs and batch size, dropout was introduced, with the aim of reducing overfitting: it happened in some way, but overfitting still arised in about 6/7 epochs (5). Regularization caused a slight increase of the test accuracy to 0.878 but, at the same time, it also caused the decrease of training accuracy to 0.905 (3). The ROC curve is mostly similar to the one from the first MLP.



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

Figure 6: Scores for the third 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 show that overfitting is not a thing anymore. Unfortunately, the scores did not improve much: the only noticeable things are that the test loss slightly decreased, reaching 0.307 (3) and that the training stopped after a definitely greater number of epochs, since it was optimized with a lower learning rate.



(a) Accuracy. (b) Loss.

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

The results obtained with these three models were 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/architecture that had 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, having a test loss of 0.335 (3), confirmed that Bayesian Optimization did not bring great improvements.

The last step on the numerical dataset was the employment of *K-fold (external) 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 had a slightly lower loss on the test set. Given a dataset S, K-fold cross-validation is achieved by partitioning it in K subsets of about the same size $|S|/K$, obtaining $S = D_1 \cup \cdots \cup D_K$. Then, for $k = 1, \ldots, K$ training is done on $S^{(k)} = S \setminus D_k$ and testing on D_k. Each iteration generates a slightly different predictor and averaging the scores for all of

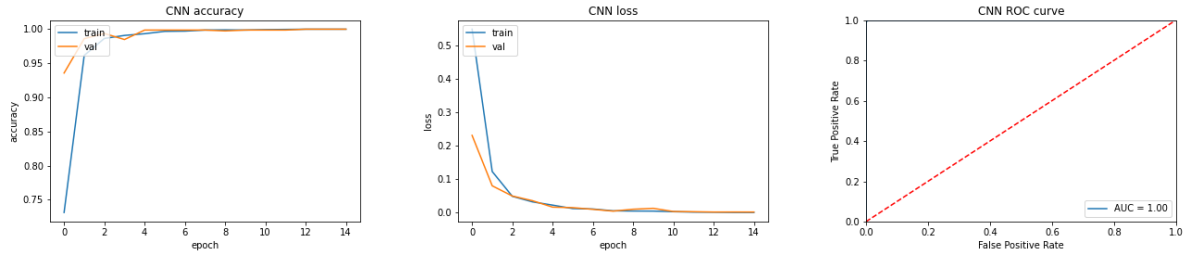
them gives an idea of the generalization capabilities of the chosen model. Even if there is a lot of overlapping since all the training sets are generated from the original dataset, this method is reliable and often used to validate a model, especially for considerable values of K. Choosing $K = 10$, in 7 it can be seen that, especially for the test set, there is indeed some deviation among the various folds but, overall, averaged scores obtained within the cross-validation showed that the model does not suffer much when facing "new" data, having an average test accuracy of 0.914 and test loss of 0.218 (as shown in the appendix 4).

	Acc			Loss			AUC
	Train	Val	Test	Train	Val	Test	Test
First MLP	**0.937**	0.887	0.869	**0.169**	0.298	0.323	0.93
Second MLP	0.905	**0.889**	0.878	0.217	**0.284**	0.316	0.93
Third MLP	0.901	0.887	0.878	0.257	0.287	**0.307**	0.93
Third MLP (Pyspark)	0.925		**0.888**	0.192		0.327	0.881
Bayesian MLP	0.822	0.884	0.864	0.403	0.311	0.3355	0.91
CNN	**1.0**	**1.0**	0.996	**0.0005**	**0.0018**	**0.0099**	1.0
CNN (Pyspark)	0.996		**0.997**	0.164		0.0108	0.997

Table 3: Scores for all the models.

As already said, only the third MLP was trained in the Pyspark implementation, with the exact same hyperparameters. In this case, however, there are two specific choices to make in order to let Elephas make the training distributed: the *frequency* to which nodes send their gradients to the master node can be done at the end of an epoch or batch (the latter was chosen); other than that, the *mode*, i.e. how this update is sent is another choice to make: it can be synchronous (all the nodes at the same time), asynchronous (with locks to not lose updates), which was the chosen one, or in hogwild mode (same as asynchronous but without locks). The split ratio among train, validation and test sets was also the same but, of course, the resulting sets were very likely to be 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 overall computation, the scores are not exactly the same, but they still seem reliable (3).

4.2 CNNs on the Image Dataset

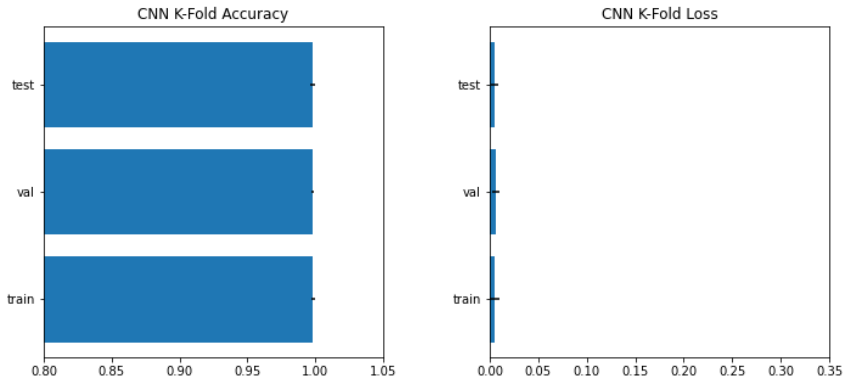


(a) Accuracy. (b) Loss. (c) Loss.

Figure 8: Scores for the CNN.

For this dataset the situation was totally different: one model was enough to obtain outstanding scores (3). 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. The curves in 8 show that this CNN converges very fast and does not overfit at all. 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, choosing $K = 5$: the choice should be independent from the algorithm and only depends on the



(a) Accuracy. (b) Loss.

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

amount of data, but CNNs are time consuming and a greater value would have taken too much with respect to the available resources. However, the averaged results are still great and the deviation among all folds is almost irrelevant (9 and 5).

For the Pyspark implementation everything followed in the same way as for the numerical dataset: the hyperparameters for the CNN were exactly the same and the updates were sent at the end of every batch in asynchronous mode. Also in this case the scores are not exactly the same but are still very similar (3).

5 Conclusion

This work has shown that trying to predict if (not real) people 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. Deeper networks on a greater amount of data could probably help to obtain better scores also on the numerical dataset.

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.

It was also shown that it is indeed possible to bring distribution within deep learning: since the employment of deep neural networks is often required when the task is hard and many data are needed, the possibility to distribute the learning among various nodes is of course a thing to keep an eye on when problems of this kind have to be solved.

Appendix

	Acc			Loss		
	Train	Val	Test	Train	Val	Test
1	0.885	**0.923**	0.861	0.298	**0.197**	0.310
2	0.887	0.906	0.899	0.291	0.219	0.298
3	0.875	0.907	**0.946**	0.303	0.221	**0.171**
4	0.888	0.903	0.933	0.302	0.226	**0.171**
5	0.878	0.908	0.917	0.295	0.220	0.192
6	0.886	0.904	0.933	0.300	0.230	0.174
7	0.883	0.913	0.907	0.292	0.221	0.197
8	0.883	0.904	0.937	0.294	0.225	0.178
9	0.890	0.910	0.913	0.281	0.225	0.211
10	**0.893**	0.904	0.893	**0.280**	0.218	0.279
AVG	**0.885**	**0.908**	**0.914**	**0.294**	**0.220**	**0.218**

Table 4: K-fold CV for the third MLP (K=10).

	Acc			Loss		
	Train	Val	Test	Train	Val	Test
1	**1.0**	**1.0**	0.996	0.0002	**0.0002**	0.011
2	0.996	0.997	0.995	0.0102	0.008	0.007
3	**1.0**	0.997	**1.0**	**0.0**	0.006	**0.0**
4	0.997	0.996	0.999	0.011	0.008	0.004
5	0.998	0.999	0.999	0.005	0.009	0.003
AVG	**0.998**	**0.998**	**0.998**	**0.0052**	**0.0063**	**0.0051**

Table 5: K-fold CV for the CNN (K=5).