Joint project for the Algorithms for massive datasets and Statistical methods for ML courses

Marco Ghezzi, Matteo Limoncini

Abstract

This project analyses and compares different solutions to tackle the problem of image classification for the Glasses or No Glasses dataset. We start from a simple neural network up to more complex Convolutional Neural Networks with multiple layers. The goal of this project is to implement an artificial neural network in order to identify if a person is wearing glasses or not. We used the TensorFlow API to build and train our models and the Keras Tuner framework for a better hyperparameters optimization. This project identified the best suited model to solve this problem: a convolutional neural network with 3 convolutional layers and 2 dense layers that reach an accuracy of 0.99 and a loss of 0.10

1. Introduction

In this project we used the Glasses or no Glasses dataset published on Kaggle. This dataset, created by a Generative Adversarial Neural Network, contains images of people wearing or not wearing glasses. We propose different solutions in order to classify if a person is wearing glasses, starting from a simple single layer Perceptron with one single neuron, up to more complex Convolutional Neural Network. We then face a typical problem of image classification analysis: overfitting. This issues was firstly addressed introducing dropout layers and then using the batch normalization technique.
We rely to TensorFlow APIs both to manage the data preprocessing and to create and train our models. We also used the Keras Tuner framework in our last experiments to improve hyperparameter optimization.

2. Data analysis

The whole dataset consists of 5000 images and its relative 512-latent vectors, divided into train and test set, that are used to generate this images; the train set, that contains 4500 entries, also includes a label for each entry that indicates if the person in the image is wearing glasses or not. In order to download the dataset, we used the Kaggle API [1] with our API credentials. After the dataset had been downloaded, we divided the images in 3 different directories, two for labeled images (one for people with glasses and the other for people without it) and the third one for images without labels. The dataset contains lots of dirty data; some images in the training set are misclassified (more than 10% of the entire dataset) and other images contains glasses

marco.ghezzi3@studenti.unimi.it (M. Ghezzi); matteo.limoncini@studenti.unimi.it (M. Limoncini)
https://github.com/marcoghezzi1 (M. Ghezzi); https://github.com/matteolimoncini (M. Limoncini)

[1] https://github.com/Kaggle/kaggle-api

Figure 1: Ambiguous images

without their temples or without their lenses, due to small errors during data generation (Figure 1). Thus, the majority of our experiments are based on these noisy dataset.

3. Data preprocessing

In order to read the images it's necessary to find the optimal method in term of space and time efficiency. In particular, it is important to find a solution to read the CSV that contains images and labels in a short time even when the dimensions of the CSV exceed the dimension of the memory of the machine we are using. After exploring some tools and frameworks the choice fell between two frameworks: *dask* [2] or *pandas* [3]. *Dask* is an open source framework that provides advanced parallelism for analytics, enabling performance at scale. *Pandas* is a tool that supports reading and writing data between in-memory data structures and different formats like CSV. We decided to use *dask* because it's a lightweight framework but at the same time it allows us to scale from a single node to thousand-node clusters.

Then, in order to split our labeled images, we generated three directories for train, validation and test sets: we used a split ratio of 0.8, 0.1 and 0.1 respectively. Starting from the images in these directories, we created a dataset using the TensorFlow function `tf.keras.utils.image_dataset_from_directory` that in our case creates three TF Datasets: one for training data, one for validation data and the last one for test data. Using this function we also specify the size of the images, to find the best results and to decrease the training time: in fact this resize operation is necessary because the original size of 1024x1024 is too large. We tried image sizes of 64x64, 128x128 and 256x256. In Figure 2 it can be seen that there are no particular differences between the image sizes: keeping this in mind, we decided to use an image size of 128x128 for the subsequent experiments since it is slightly better than 64x64 and the training time does not increase significantly.

After this process, each pixel of the image in the two datasets is normalized to values between 0 and 1, using the `tf.keras.layers.Rescaling` layer: this is useful since we are dealing with a binary classification task and the output of our neural network will be between 0 and 1.

[2] https://dask.org/
[3] https://pandas.pydata.org/

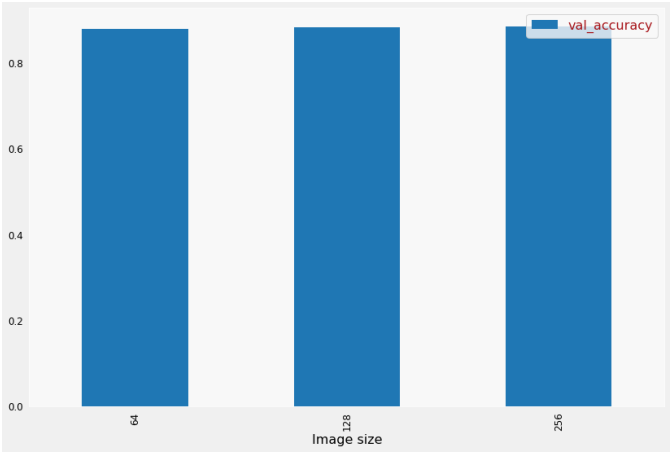


Figure 2: Validation accuracy on a CNN with different image size.

Then, the dataset is cached onto disk, since it can not be stored entirely in memory; this will save the operation of rescaling from being executed during each epoch. After this, we shuffle the training dataset: this process is done to avoid producing exactly the same elements during each iteration through the dataset.

Finally a prefetch operation is applied both to training and validation dataset: this process decouple the time in which data is produced from the time in which data is consumed. The `buffer_size` parameter has been fixed to AUTOTUNE: this assures that the buffer size is equal or greater than the number of batches consumed by a single training step.

The last step of the data preprocessing was to manually remove all the ambiguous and misclassified data, both from the training and the validation set. Firstly, it is necessary to remove incorrect images from the validation set because it is impossible to evaluate the performance of an artificial neural network with data with bad labels. Then, the incorrect images from the training set need to be removed; otherwise, the result will be a neural network trained on bad data. This manual process has been possible because of the small dimension of the dataset. Another possibility is to remove misclassified data, using an existing neural network already trained on correct data: in this way can be created a new training dataset that associates the new labels with those predicted by this neural network.

4. Development process

4.1. Simple Single layer Perceptron Artificial Neural Network

The first approach is to use a Single layer Perceptron with one fully connected Dense layer with one unit. The activation function used in this layer is the Sigmoid function, since we are dealing with a binary classification problem: this function takes any real value as input and outputs values in the range 0 to 1. The greater the input, the closer the output value will be to 1,

whereas the smaller the input, the closer the output will be to 0.

$$S(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

We decided to use the Adam optimization algorithm, an extension to stochastic gradient descent. The choice of this optimizer is due to the fact that it is straightforward to implement, computationally efficient and requires little memory.

Even the loss function needs to be properly chosen. For this type of problem the better loss function is the binary cross entropy. The main goal of this loss is to make our predictions as close to the true labels as possible, using the following formula:

$$-\left(y_{true} \cdot \log\left(y_{pred}\right) + \left(1 - y_{true}\right) \cdot \log\left(1 - y_{pred}\right)\right) \tag{2}$$

In order to evaluate the precision of the networks we decided to use two metrics: loss and accuracy. The loss is defined as the difference between the predicted value by the model and the true value, while the accuracy is defined as the number of correct predictions over the total number of predictions.

We trained the model on the preprocessed training dataset, using a batch size of 64. The batch size is a term used to refer to the number of training examples used in one iteration. The choice of the correct value of batch size is a trade off between accuracy and speed.

There are 3 different options:

1. batch mode: when the batch size is equal to the entire dataset. In this case an epoch and an iteration are equivalent;
2. mini batch mode, the method used in this project, where the batch size is greater than one but less than the total size of the dataset;
3. stochastic mode: this method uses a batch size equal to one. The gradient and artificial neural network are updated after each sample.

We train the network with a great number of epoch (60) but we introduce the early stopping callback to halt the training of neural network at the right time. Early stopping is a method that allows to specify an arbitrary large number of training epochs and stopping training once the validation accuracy stops improving on a hold out validation dataset. With this method we can avoid overfitting, typical of training with a lot of epochs.

Since we noticed that the network was learning too fast, we decided to add a learning rate decay. It starts with a large learning rate and then decays it multiple times during training after each epoch. An initially large learning rate accelerates training the network escaping spurious local minima: the decaying of the learning rate helps the network converge to a local minimum and avoid oscillation.

With this simple network we obtain some useful results (Figure 3). Since this is a binary classification so we have a 0.50 of accuracy with a random algorithm and with this network we achieve, in the best case, a value of 0.27 of loss and 0.87 of accuracy.

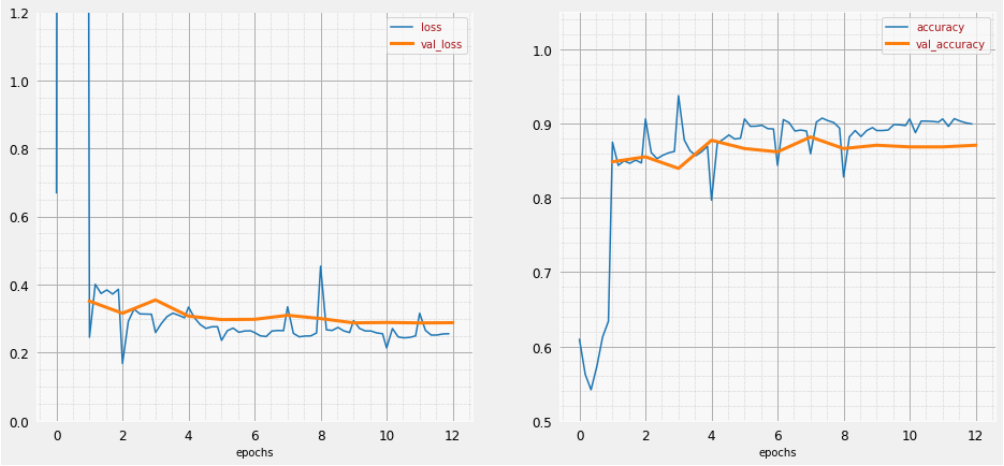


Figure 3: Training and validation loss, training and validation accuracy during the training of a single-layer neural network

4.2. Convolutional neural networks

Convolutional Neural Network (CNN) [1] are very similar to ordinary Neural Networks: they are made up of neurons that have learnable weights and biases. Each neuron receives some input, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. They still have a loss function on the last (fully-connected) layer.

Convolutional networks assume that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network. Unlike Regular Neural Networks, CNN scale well to full images. By using CNN it's possible to obtain better results with less training time.

Convolutional network is a sequence of layers, and every layer transforms one volume of activations to another through a differentiable function. We used three main types of layers to build our neural network: Convolutional Layer [4], Pooling Layer [5] and Fully-Connected Layer:

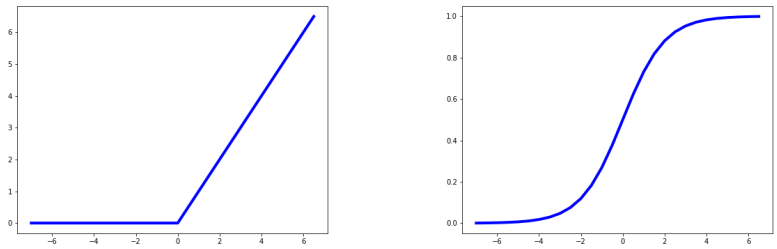
- the convolutional layer is the core of a convolutional network that uses most of the computation resources: neurons in these layers receive input from a subarea of the previous layer. The parameters to be specified consist of a set of learnable filters. Every filter is small along width and height, but extends through the full depth of the input volume;
- the pooling layers, inserted between successive convolutional layers, has the function to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting;

[4] https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D
[5] https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D

- neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

The first model consists of a neural network with 2 convolutional layers and the usual output Dense layer with the sigmoid activation function connected with a fully-connected layer. Both the convolutional layers have a kernel size of 3x3, while they have a different number of filters: 16 for the first and 32 for the second. Between the convolutional layers we apply a Max Pooling layer, a sliding window that applies the 'max' operation. We use a flatten layer that allows us to change the shape of the data from a vector of 2d matrices (or nd matrices really) into the correct format for a dense layer to interpret. We use a dense layer with 128 neurons and an output layer with 1 neuron, due to binary classification. The last layer uses the sigmoid activation function, the others use ReLU (Rectified Linear Activation) activation function.

The choice of activation function in a neural network is a critical point because it defines how the weighted sum of the input is transformed into an output from a node or nodes in a layer of the network. All hidden layers typically use the same activation function. The output layer will typically use a different activation function from the hidden layers and it is dependent upon the type of prediction required by the model.



(a) ReLU activation function (b) Sigmoid activation function

Figure 4: Activation functions

With this first convolutional neural network we obtain good results, as can be seen in Figure 5. The graphic shows a problem of overfitting: at a certain point (more or less from 10 epochs) the results for training became better and better but the results in the validation set do not improve. The phenomenon of overfitting can be defined as the occasion in which the network learns too much from the training set and it becomes difficult to generalize its knowledge, so it cannot perform accurately against unseen data and it has bad results in validation set. Overfit occurs when the model trains for too long on sample data or when the model is too complex, because it can start to learn the "noise", or irrelevant information and become unable to generalize well to new data.

4.3. Avoiding overfitting

There are different methods to avoid overfitting.

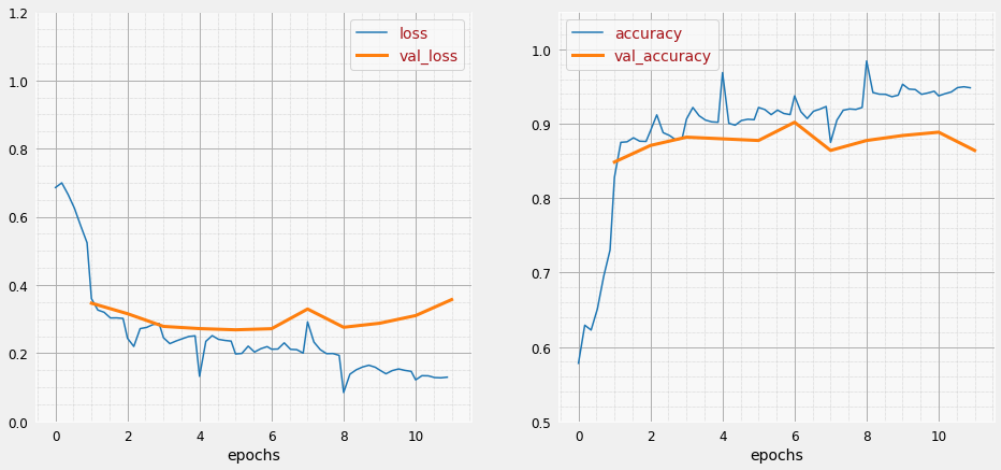


Figure 5: Overfitting

One possibility is to use more data to train the network, for example using a different split of the dataset or using data augmentation. We use 80% of the dataset to train the network so we decided to not use data augmentation to avoid increasing too much training time.

Another possibility to avoid overfit is to simplify the model. To try to reduce overfitting we introduce **dropout**. Dropout is a technique that randomly decides to switch off some percentage of neurons of the network. When the neurons are switched off the incoming and outgoing connection to those neurons is also switched off. With this technique the model reduces its training capability. Dropouts are usually not used after the convolution layers, or used with a percentage very low like 0.1 or 0.2. Dropouts are mostly used after the dense layers of the network with a percentage that is not higher than 0.5. In the network we tried to insert dropouts after the dense layers with a percentage of 0.4 but we did not see a great improvement, as it is represented in Figure 6.

Another technique used in this project to try to reduce overfitting is the **batch normalization**. Batch normalization is a layer that allows others layers to learn in a more independent way. This layer is added to standardize input or output between the layers of the model. Usually it is used after the convolutional and pooling layers. In this project a batch normalization layer has been used to try to reduce overfitting after input and hidden layers. In order to add batch normalization we rely on the BatchNormalization layer provided by Keras.

Also with batch normalization we had some improvements but there is still overfitting as it is pictured in Figure 7.

The last possibility is to use early stopping to reduce the number of epochs. An artificial neural network could learn too much and produce overfitting if training used a great number of epochs.

Early stopping has been used in this project with some good results in terms of overfitting but not in terms of loss and accuracy, so it has been decided not to focus on overfitting but to change point of view. We decided to focus on the structure of the network and on how we can choose in a proper manner the best hyperparameters.

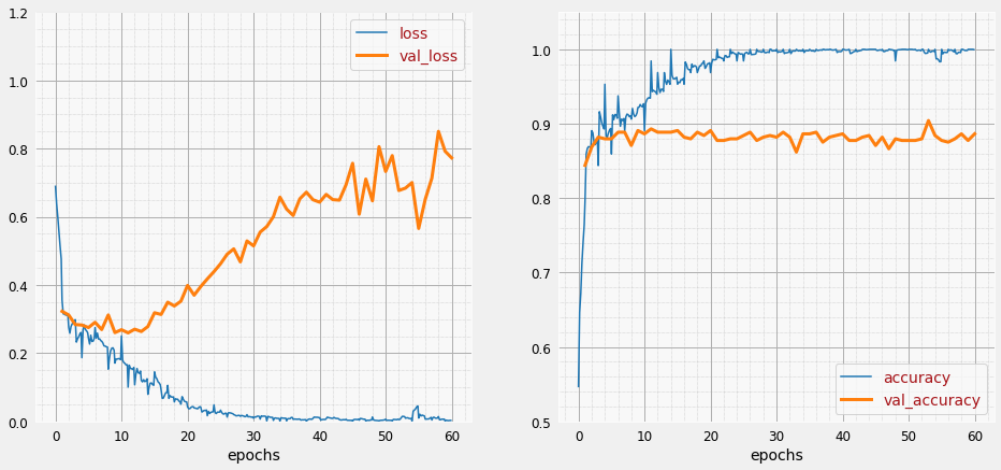


Figure 6: Training and validation loss, training and validation accuracy with dropout

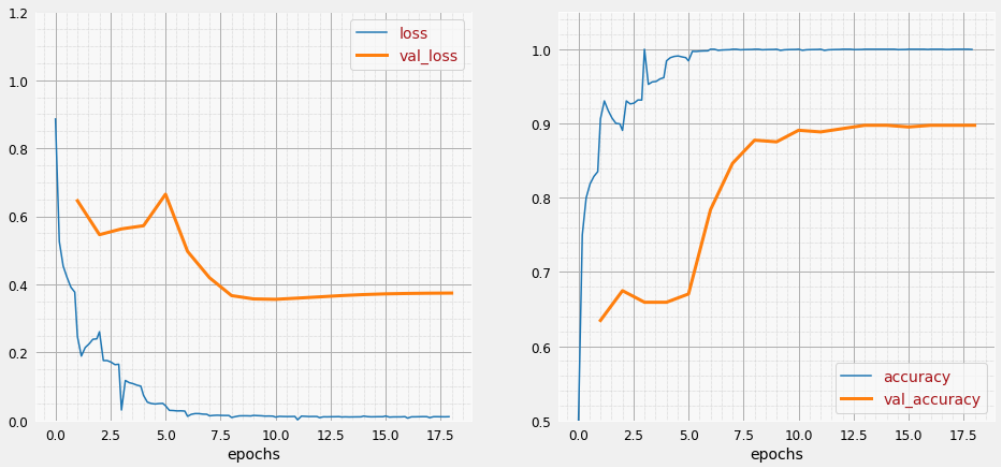


Figure 7: Training and validation loss, training and validation accuracy with batch normalization

4.4. Hyperparameter tuning

Hyperparameters are variables that modify the training process and the topology of a machine learning model; they remain constant while the training process is ongoing and they heavily impact the performance of our models. [2] Hyperparameters are of two types:

- **model hyperparameters**: they are used to change model selection such as the number of hidden layers or the architecture of a neural network;
- **algorithm hyperparameters**: they influence the speed and the quality of the learning algorithm.

In our work the hyperparameters tuned are the number of convolutional layers in the CNN, the number of filters and the kernel size for each convolutional layer, concerning the model

hyperparameters. The algorithm hyperparameters tuned are the learning rate, the number of epochs and the rate parameter in the Dropout layers.

Finding the optimal hyperparameters for a neural network is probably the most difficult task: in our first experiments we randomly choose those hyperparameters and we evaluated the results of the model. In our last experiment we referred to the **Keras Tuner** [6] framework that solves the problem of hyperparameter tuning: in particular, it provides search algorithms to find the best hyperparameter. In this work we used the **Hyperband** algorithm [3].

Hyperband is an algorithm that tries to speed up the RandomSearch algorithm up through adaptive resource allocation and early-stopping: it is based on the idea that when a certain choice of the hyperparameters gives us bad results, it trains that model with a small number of epochs, in particular less than the '`max_epochs`' parameter. The algorithm then selects the best candidates based on the results of these few epochs and it applies full training and evaluation on the final chosen candidates.

Even though the algorithm gives us the best hyperparameters, we need to fix the search space for our hypermodel. In particular we set the search of the number of convolutional layers between 2 and 4, the number of filters between 12 and 64, the kernel size between 3x3 or 5x5, the number of units in the last Dense layer between 32 and 200, the pooling layer between AveragePooling and MaxPooling and the learning rate chosen between 0.01, 0.001 and 0.0001. Finally, it is necessary to setup the arguments inside the Tuner Class; especially we set the `max_epochs` (maximum number of epochs to train one model) to 20 and the `seed` in order to replicate the same starting point over multiple executions. After the algorithm terminated its execution, we saved the best model and we evaluated it with the test set, consisting of images never seen by the model. The validation accuracy reached by this model is near 0.91, while we achieve an accuracy of 0.86 on the test set after 30 trials of the Hyperband algorithm.
Finally, we applied the data cleaning process explained in Section 2 in order to further improve the results.

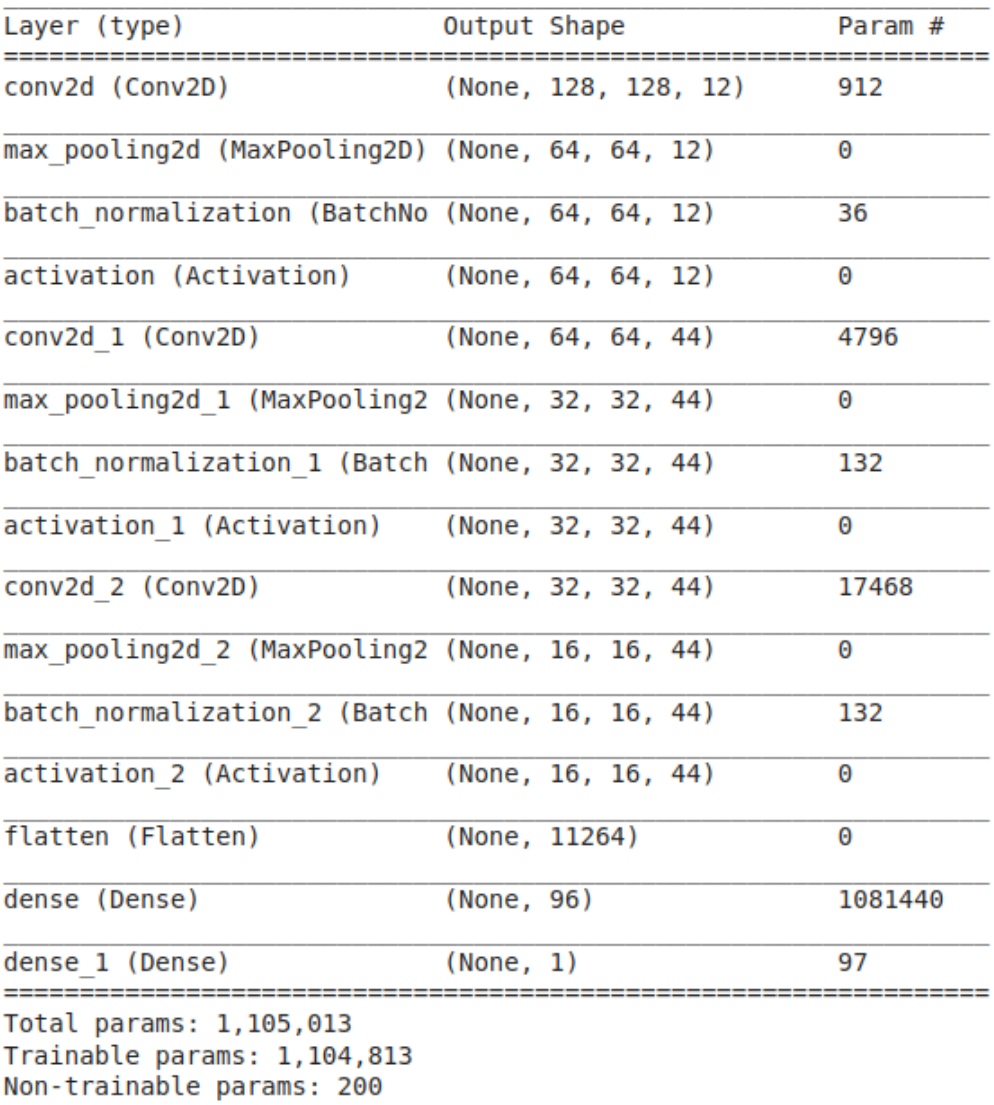
5. Best model and results

After the new datasets had been built we decided to directly use the Keras Tuner framework that was defined above. In Figure 8 it's represented the summary of one of the best models produced by the algorithm. Since this algorithm randomly chooses the best hyperparameters after each iteration, we could've different models at the end of its execution. In our work, after approximately 12 minutes of the Hyperband algorithm execution, we have an accuracy on the test set of 0.99 and a loss of 0.10.

We obtained good results in images that the network does not use for training with a training time that was not so high.

In our work, we managed to reach these results by using a dataset of homogeneous images. In future development, it's possible to try to improve the accuracy of the network and the ability to detect different images using a bigger and heterogeneous dataset resorting to the data augmentation technique [4]. The idea is to apply several transformations to data in the dataset

[6] https://keras.io/keras_tuner/



Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 128, 128, 12)	912
max_pooling2d (MaxPooling2D)	(None, 64, 64, 12)	0
batch_normalization (BatchNo	(None, 64, 64, 12)	36
activation (Activation)	(None, 64, 64, 12)	0
conv2d_1 (Conv2D)	(None, 64, 64, 44)	4796
max_pooling2d_1 (MaxPooling2	(None, 32, 32, 44)	0
batch_normalization_1 (Batch	(None, 32, 32, 44)	132
activation_1 (Activation)	(None, 32, 32, 44)	0
conv2d_2 (Conv2D)	(None, 32, 32, 44)	17468
max_pooling2d_2 (MaxPooling2	(None, 16, 16, 44)	0
batch_normalization_2 (Batch	(None, 16, 16, 44)	132
activation_2 (Activation)	(None, 16, 16, 44)	0
flatten (Flatten)	(None, 11264)	0
dense (Dense)	(None, 96)	1081440
dense_1 (Dense)	(None, 1)	97

Total params: 1,105,013
Trainable params: 1,104,813
Non-trainable params: 200

Figure 8: Model with best results trained on cleaned data

(like horizontal or vertical flips, zoomed in/out, color distortions or translations) to augment the dimension of the datasets for training data.

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

References

[1] J. Wu, Introduction to Convolutional Neural Networks, 2017. https://cs.nju.edu.cn/wujx/paper/CNN.pdf.

[2] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016. http://www.deeplearningbook.org.

[3] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, A. Talwalkar, Hyperband: A novel bandit-based approach to hyperparameter optimization, Journal of Machine Learning Research 18 (2018) 1–52. URL: http://jmlr.org/papers/v18/16-558.html.

[4] L. Perez, J. Wang, The effectiveness of data augmentation in image classification using deep learning, 2017. arXiv:1712.04621.

A. Online Resources

The sources for this project are available via GitHub