AMD-SM2L-2020-21-PROJECT

Davide D'Ascenzo
University of Milan
davide.dascenzo@studenti.unimi.it

ABSTRACT

Neural networks are an increasingly popular solution for many machine learning problems. One of the tasks where neural networks shine is object recognition. In this work we focus on such task developing a convolutional neural network capable of identifying people with and without glasses. The whole pipeline is meant to scale with large datasets in a distributed fashion. Tensorflow APIs are used to prepare the dataset and ensure data parallelism, while Keras APIs are used to train the models and provide distributed capabilities. Hyperparameter optimization is tackled through random searches and descriptive statistics.

1 Introduction

Object recognition (or object detection) is a computer vision technique for identifying objects in digital images and videos. Here, we will focus on images of people with and without glasses. To deal with this task, convolutional neural networks (CNNs) are built.

CNNs are a particular class of neural networks, commonly used to analyze visual inputs. They are characterized by the extensive application of the convolution operation (and hence the name) to selectively extract features from small portions of the input.

In this work, all CNNs are built using Tensorflow and Keras. Tensorflow is a machine learning framework particularly focused on deep neural networks. Keras, on the other hand, is a high-level API built on top of Tensorflow. It is commonly used for the intuitive workflow that it provides to build and train neural networks.

2 Dataset

The dataset used is the "Glasses or No Glasses"[1] dataset provided on Kaggle by Prof. Jeff Heaton under the CC BY-SA 4.0[2] license. It consists of 1024x1024 pixels sized photos of people with and without glasses (as the name suggests), generated by a generative adversarial network (GAN)[1]. The GAN uses a 512-dimensional latent vector to generate each image, therefore no real people are present in the dataset.

The images are organized in train and test sets by means of two CSV files. Both files contain IDs of the images belonging to one or the other set, and the 512-dimensional latent vector associated with each image. However, only the train CSV file contains labels for the images.

Since we want to assess the quality of our classifiers, we need labels to test our predictions, therefore only the train set is used.

2.1 Data organization

The dataset is organized in one big folder containing all the images, and the two CSV files with information relative to each image. We reorganize this configuration to make it more suitable to our needs. In particular:

[1]`https://www.kaggle.com/jeffheaton/glasses-or-no-glasses`
[2]`https://creativecommons.org/licenses/by-sa/4.0/`

- First, we load the train CSV using Dask[3].
- Once the CSV is loaded, we use the IDs and the labels to subdivide train images in ‘glasses’ and ‘no_glasses’ folders. Test images are discarded.
- We proceed to check for data unbalancedness: there are 1.74 ’glasses’ labels every ’no glasses’ label. Since the dataset doesn’t seem particularly unbalanced, we keep it as it is.
- We further split the data in ‘train’, ‘val’ and ‘test’ folders, each containing a ‘glasses’ and ‘no_glasses’ folder. A shuffle operation is performed before splitting, in order to get rid of possible patterns or biases in the data gathering process[4].

  This final split simplifies the subsequent Tensorflow Dataset (TF Dataset) generation.

2.2 Data preprocessing

At this point, we need a way to load the data into the memory. We have to use batches because the dataset is too big to fit entirely in memory, and even if it could, we should always work towards an input pipeline for large datasets. For this reason, TF Dataset API is the best option available, since it provides different methods to deal with loading in batches and parallel and distributed computing. In particular:

- We resort to the ‘image_dataset_from_directory’ function to generate our TF Datasets, which loads images from a folder tree structure like the one we created earlier. From this operation, three datasets are created: one for training, one for validation and one for testing. We also perform the resizing of the images in this step, in order to reduce the amount of data transmitted to the GPUs in the training steps. The resizing is necessary because 1024x1024 size images are too large to be used in our environment, in fact 64x64, 128x128 and 256x256 size images are used in subsequent analyzes.
- Using the TF Dataset API and a Rescaling Keras layer, each image in the datasets is normalized to pixel values between 0 and 1. Normalization is useful to have comparable values in every layer of a neural network. Since the output of our CNNs will be between 0 and 1 (because we are dealing with a classification task), normalizing the input will make learning easier.
- The datasets are then cached to file. We can’t cache the datasets in memory, even if we are using 64x64 size images, because larger datasets couldn’t fit entirely in memory. Nevertheless we can cache the preprocessed datasets to disk, avoiding to repeat the resizing and the rescaling operations each time. This step reduces the computational load on the CPU, since all the preprocessing takes place on it.
- A shuffle operation is pipelined in the training dataset. This ensures the creation of different batches in each training epoch, which prevents the possibility of getting stuck with a particularly bad batch configuration.
- Lastly, a prefetch operation is pipelined to every dataset. Prefetching decouple the data loading done by the CPU from the data processing done by the GPU: while the GPU is processing the current batch, the CPU starts to load in memory the subsequent batch, in order to speed up the whole computation. See Figure 1 for an example of workflow with and without prefetching. This ends the data preprocessing.

2.3 Data augmentation

Data augmentation is a technique that enhance the size and the diversity of the data available, commonly used when dealing with images. It acts as a regularizer and helps reduce overfitting when training a machine learning model [2]. In this work we implement data augmentation via Keras Data Augmentation Layers, in particular the following layers are used:

- RandomFlip[5] layer: this layer randomly flip images horizontally and vertically. In our case, only horizontal flip is implemented.

[3]Dask is an open-source library for parallel computing and large datasets. We use Dask keeping in mind that our dataset could be much bigger than the one presented, and the CSV file could possibly not fit entirely in memory. Dask addresses this issue using lazy evaluation and performing computation only when needed, without having to keep all the data in memory. This is somewhat similar to the Spark approach with Resilient Distributed Datasets (RDDs).

[4]Clearly, if the data are not representative of the whole population, shuffling the dataset will not fix this problem. However, for instance, if the data are collected using some pattern (e.g. taking pictures of people from town to town), then this shuffle operation will get rid of the intrinsic pattern of how the data have been collected or generated.

[5]`https://www.tensorflow.org/api_docs/python/tf/keras/layers/experimental/preprocessing/RandomFlip`

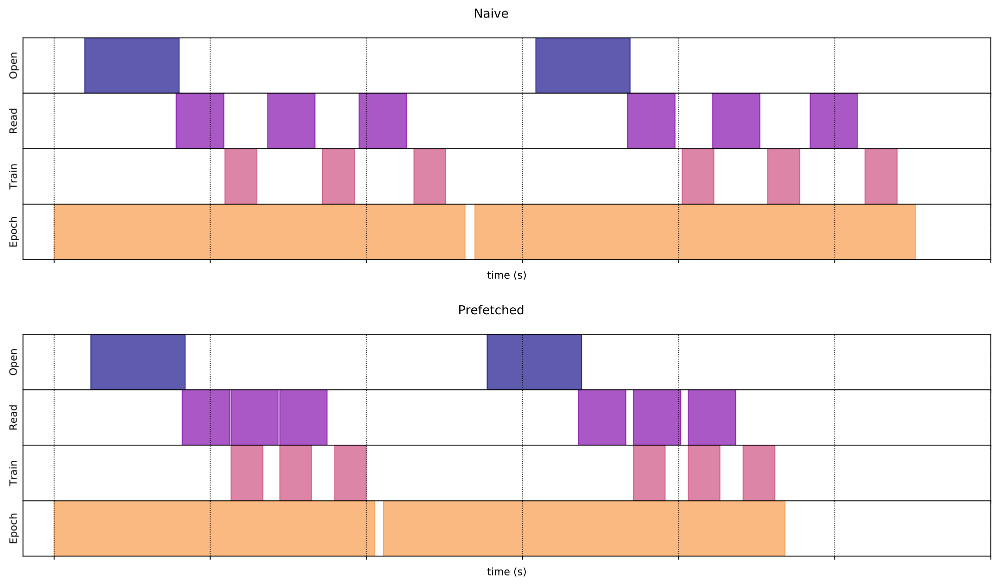


Figure 1: Data execution time plots showing workflow with (bottom) and without (top) prefetching. Images taken from `https://www.tensorflow.org/guide/data_performance#prefetching`

- RandomTranslation[6] layer: this layer randomly translate the image in both width and height directions.
- RandomRotation[7] layer: this layer randomly rotate the image. In our case, the rotation is capped to $\pm 45°$.
- RandomZoom[8] layer: this layer randomly zoom the image by a certain factor.

Figure 2 shows the whole data augmentation pipeline, while Figure 3 shows some images generated after augmentation. Since data augmentation takes place after data preprocessing, the images shown have already been resized.

The described pipeline is part of the models fed to GPUs for training. Hence, all augmentation steps take advantage of GPU speed and don't overload the CPU.

3 Convolutional Neural Network

CNNs, as previously stated, are a particular subclass of neural networks characterized by the convolution operation. This operation is performed through a kernel, called convolutional kernel, that is repetitively applied and shifted along all the input dimensions, producing the so-called feature map. Convolutional kernels are squared matrices, generally of odd dimension, that are convoluted with a portion of the input, commonly called receptive field (term borrowed from biology, more precisely from the human visual system, from which CNNs are inspired). Since these kernels are shifted along the entire input, the presence of a certain feature will be captured no matter where it occurs in the input. For this reason, CNNs are also known as shift invariant or space invariant artificial neural networks (SIANN) [3][4].

Apart from the convolution operation, CNNs are characterized by another operation called pooling. Pooling is the aggregation of features in a feature map using a certain window size juxtaposed across the entire feature map. All the data in each window are aggregated into one value, and this is tipically done to reduce the dimension of the feature maps while trying to preserve as much useful information as possible. Lowering feature maps dimension, CNNs are free to grows in depth, without much burden on the amount of computation performed in the network.

In this work, CNNs are dinamically built by means of hyperparameters (explained in the subsequent section), but all of them are characterized by the following Keras layers:

[6]`https://www.tensorflow.org/api_docs/python/tf/keras/layers/experimental/preprocessing/RandomTranslation`

[7]`https://www.tensorflow.org/api_docs/python/tf/keras/layers/experimental/preprocessing/RandomRotation`

[8]`https://www.tensorflow.org/api_docs/python/tf/keras/layers/experimental/preprocessing/RandomZoom`

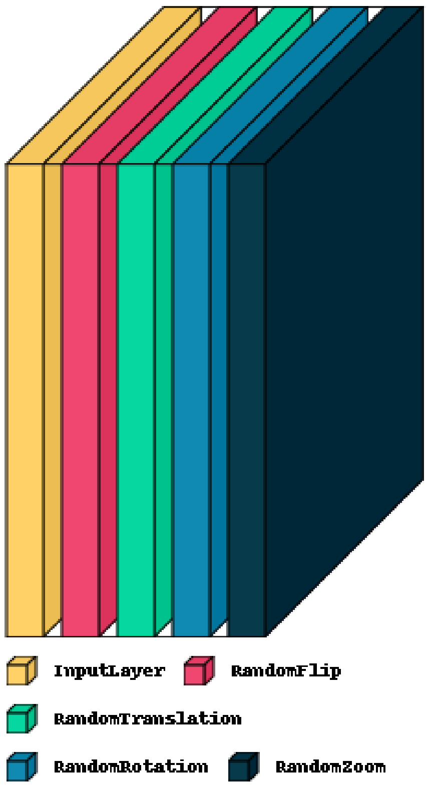


Figure 2: The data augmentation pipeline.
Image generated with `https://github.com/paulgavrikov/visualkeras`

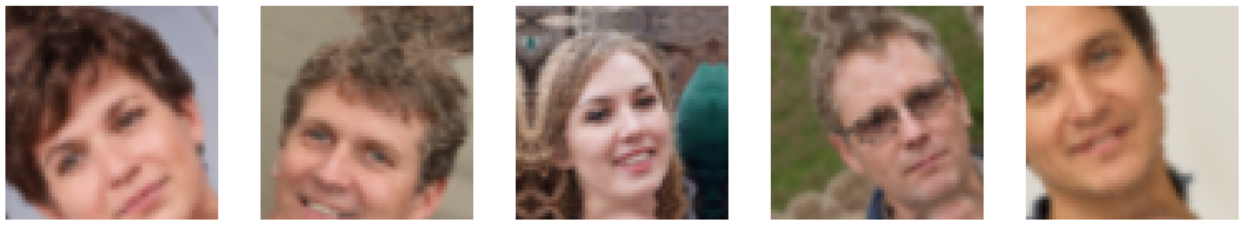


Figure 3: Images generated after augmentation. Since data augmentation takes place after data preprocessing, the images shown have already been resized.

- Input[9] layer: this layer represent the entry point of a neural network. Its shape should be equal to the shape of the inputs. In our case the shape is determined by the image size and the image channels.
- Conv2D[10] layer: this layer applies the convolution operation, with a certain convolutional kernel, to the input of the layer. The significant arguments to provide to this layer are:
  - Filters: this argument represent the number of feature maps generated by the Conv2D layer.
  - Kernel size: in our case, the kernel size is fixed to 3 for all Conv2D layers. This choice is due to the fact that chaining subsequent 3x3 kernels can mimic any other bigger kernel, using lesser parameters. For example, consider a 5x5 kernel: it consists of 25 parameters, but we can obtain the same receptive field using two 3x3 kernels chained one after the other. Each 3x3 kernel consists of 9 parameters, using two of such kernels we have a total of 18 parameters. Thus, we bring down the number of parameters from 25 to 18.

[9] `https://www.tensorflow.org/api_docs/python/tf/keras/layers/InputLayer`
[10] `https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D`

  - Strides: this represent the shift to make in each dimension of the input, after applying the kernel. We left the default value of (1,1), which represent a shift of one unit in each direction (which means that the kernel covers every contiguous 3x3 block present in the input).
  - Padding: since convolution aggregate the receptive field into one value, the resulting feature map will be slightly smaller than the input. This parameter handles this problem giving the possibility to return a feature map of the same size of the input padding the borders of the input during convolution.
  - Activation: this is discussed in the next bullet point, as an indipendent layer.
- Activation[11] layer: this layer applies a certain activation function. It can be instantiated as a layer or passed as an argument to other layers (e.g. to Conv2D layer). Activation functions used in this work are 'ReLU' (for each hidden layer) and 'Sigmoid' (for the output layer).
- MaxPooling2D[12] layer: this layer applies the pooling operation to a 2D input, using the 'max' function as aggregator.
- AveragePooling2D[13] layer: this layer applies the pooling operation to a 2D input, using the 'average' function as aggregator.
- BatchNormalization[14] layer: this layer applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1 using last batch output as reference. This operation should reduce the training time and stabilize the network [5].
- GlobalMaxPooling2D[15] layer: same as MaxPooling2D, but it aggregates an entire feature map in one value.
- GlobalAveragePooling2D[16] layer: same as AveragePooling2D, but it aggregates an entire feature map in one value.
- Dense[17] layer: this is a standard fully connected layer.

The following architectural pattern is applied to all CNNs later discussed:

- Input layer;
- Data augmentation layer;
- a variable number of Convolution blocks;
- GlobalMaxPooling2D/GlobalAveragePooling2D;
- Dense layer;
- Output layer (equals to a Dense layer with one unit);

where the data augmentation layer corresponds to the data augmentation pipeline previously discussed, and each convolution block is a block of layers composed of:

- Conv2D layer;
- MaxPooling2D/AveragePooling2D layer;
- BatchNormalization layer;
- Activation (ReLU) layer.

An example of such architecture can be seen in Figure 4.

4 Hyperparameters

An hyperparameter is a parameter which is not learned during training. This kind of parameters are tuned by the user to control the learning process in some way. We can identify two types of hyperparameters:

[11] `https://www.tensorflow.org/api_docs/python/tf/keras/layers/Activation`
[12] `https://www.tensorflow.org/api_docs/python/tf/keras/layers/MaxPool2D`
[13] `https://www.tensorflow.org/api_docs/python/tf/keras/layers/AveragePooling2D`
[14] `https://www.tensorflow.org/api_docs/python/tf/keras/layers/BatchNormalization`
[15] `https://www.tensorflow.org/api_docs/python/tf/keras/layers/GlobalMaxPool2D`
[16] `https://www.tensorflow.org/api_docs/python/tf/keras/layers/GlobalAveragePooling2D`
[17] `https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense`

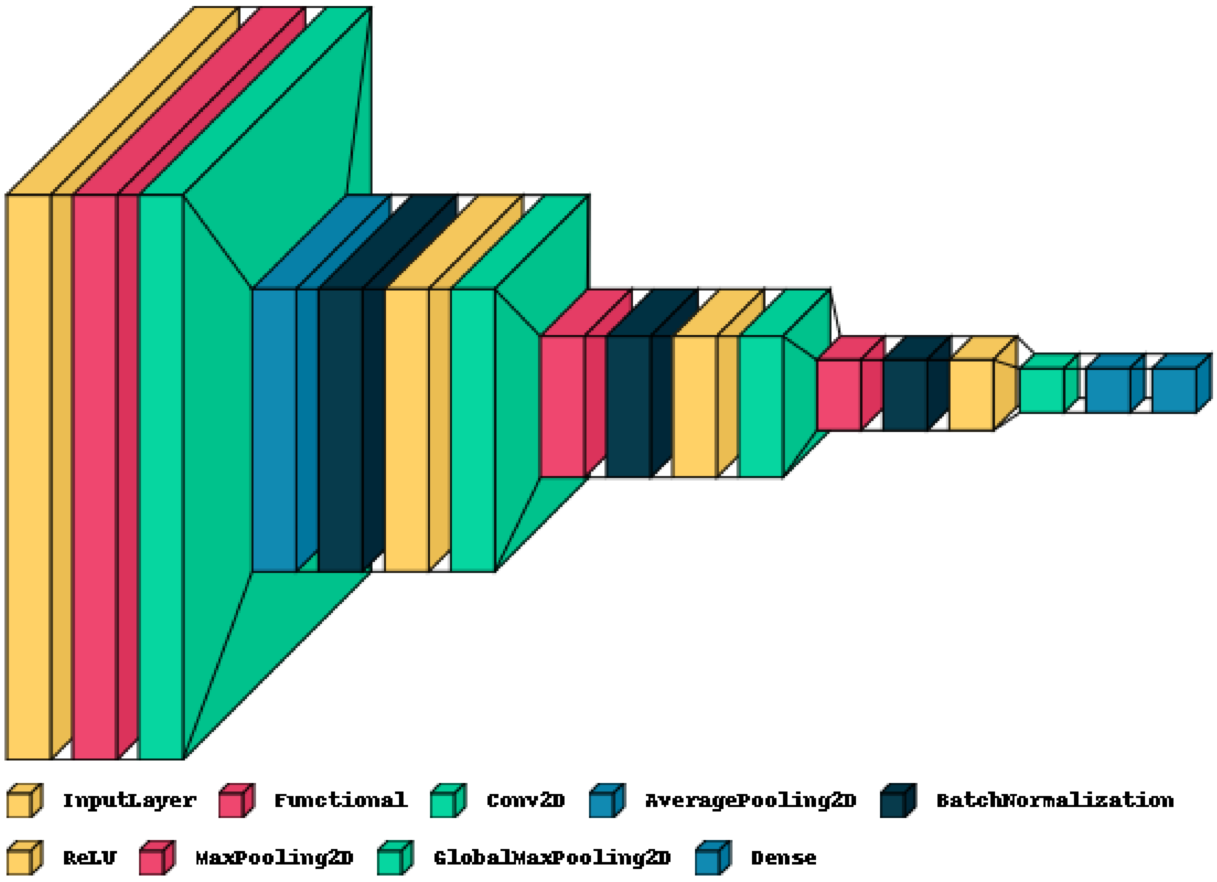


Figure 4: An example of CNN architecture.

- Model hyperparameters: they are used to control the model hypothesis space. In our case, model hyperparameters are all those parameters that control the CNN architecture.
- Algorithm hyperparameters: they are used to control the speed and the quality of the learning process.

Model hyperparameters used in this work are:

- Number of convolution blocks (‘conv_layers’): this hyperparameter is used to decide the number of convolution blocks in the CNN architecture.
- Number of feature maps (‘filters’): this hyperparameter controls the number of feature maps output from the Conv2D layer. A different hyperparameter value is associated with each Conv2D layer.
- Pooling (‘pooling’): this hyperparameter defines the pooling operation (MaxPooling2D or AveragePooling2D) after each Conv2D layer. Each convolution block has a different hyperparameter value associated with it.
- Global pooling (‘global_pooling’): this hyperparameter defines the global pooling operation (GlobalMaxPooling2D or GlobalAveragePooling2D) applied after the convolution blocks.

Algorithm hyperparameters used in this work are:

- Optimizer (‘optimizer’): this hyperparameter controls the optimizer used by the CNN during training, options are ‘adam’[6] or ‘sgd’ (stochastic gradient descent).
- Training epochs (‘epochs’): this hyperparameter sets the maximum number of epochs to train each CNN model.
- Early stopping patience (‘earlystopping_patience’): this hyperparameter defines the number of epochs after which training is prematurely interrupted since the model has stopped improving.

Lastly, we have the image size hyperparameter. This is somewhat between a model and an algorithm hyperparameter, and it is used to select the dimension of resized images.

4.1 Hyperparameter optimization

Estimating good hyperparameter values is not an easy task. In fact, increasingly sophisticated algorithms have been developed during the years. In this work, we use a fairly simple algorithm: random search.

Random search is an optimization technique based on random sampling of the parameter space. In hyperparameter optimization, the random search algorithm simply select random values in the domain of each hyperparameter and perform the training of the generated model. This algorithm can be really powerful when only a small subset of the hyperparameters significantly affect the performance of the model. Since the algorithm perform a random sampling in the parameter space, aggregating and analyzing the results for each hyperparameter can lead to good insights of which hyperparameter we should focus on. Plus, random search is an embarrassingly parallel algorithm. This means that with little or no effort we are able to parallelize the whole execution of the algorithm. This property of the random search should not surprise: since the algorithm sample at random from the parameter space, we can train each generated model independently and aggregate the results at the end. Since we are trying to develop a scalable and distributed pipeline, the choice of this optimization technique is crucial.

Nevertheless, also random search algorithm has its own (meta)hyperparameters to be tuned. However, these parameters are much more user-friendly and they can be tuned manually as needed. For example, the random trials ('random_trials') hyperparameter controls the number of samples (and therefore the number of models to train) generated by the algorithm, and it could be easily adjusted based on the computational capabilities of your own system. Another parameter is the random seed ('random_seed'), which controls the sampling process and can be fixed in order to reproduce the same results. In this work, a random seed equals to 1337 is set.

5 Distributed training architecture

Before delving into the core algorithm, we want to clarify the distributed architecture used to train the models. Since the computation is mostly done during random search optimization, we aim to distribute and parallelize this technique as much as possible. For this reason, we resort to Keras Tuner.

Keras Tuner Keras Tuner is a subsection of the Keras API that provide a scalable framework for hyperparameter optimization. It offers different tuning strategies with an easy-to-use API. As previously stated we will use random search as a hyperparameter optimization technique, therefore the RandomSearch Tuner of Keras Tuner is used.

The Tuner[18] class handles the hyperparameter search process, including model creation, training, and evaluation. For each trial, a Tuner receives new hyperparameter values from an Oracle instance. After calling 'model.fit(...)', it sends the evaluation results back to the Oracle instance and it retrieves the next set of hyperparameters to try.

The Oracle[19] class is the base class for all the search algorithms in KerasTuner. An Oracle object receives evaluation results for a model (from a Tuner class) and generates new hyperparameter values.

In order to distribute and parallelize the whole optmization process, we resort to two techniques: the first one is distributed training of one model on multiple GPUs, while the second one is parallel training of multiple models independently one from the other.

MirroredStrategy 'tf.distribute.MirroredStrategy'[20] strategy is used for synchronous distributed training on multiple GPUs on one machine. The MirroredStrategy creates one replica per GPU device. Each variable in a model is mirrored across all the replicas. Together, these variables form a single conceptual variable called MirroredVariable. These variables are kept in sync with each other by applying identical updates. Efficient all-reduce algorithms are used to communicate the variable updates across the devices. All-reduce aggregates tensors across all the devices by adding them up, and makes them available on each device.

Chief-worker model The chief-worker model is the infrastructure used to scale the learning process on multiple machines.

[18]`https://keras.io/api/keras_tuner/tuners/`

[19]`https://keras.io/api/keras_tuner/oracles/`

[20]`https://www.tensorflow.org/guide/distributed_training#mirroredstrategy`

The chief process is executed on a single-threaded CPU instance and runs a service to which the workers report results and query for the hyperparameters to try next. Substantially it acts like the Oracle of the search process. Since the chief is in charge of collect and aggregate results, only one chief process is allowed at a time.

The worker process is usually executed on a GPU environment composed of one or more GPUs. It is in charge of the building, training and evaluation of a model, requesting hyperparameters to the chief. Substantially it acts like the Tuner of the search process. If more than one GPU is assigned to a worker, the MirroredStrategy distributes the training procedure on all GPUs. For example, if we have 16 workers with 4 GPUs on each worker, you can run 16 parallel trials with each trial training on 4 GPUs.

To allow comunication between the chief and the workers, some environmental variables need to be set for each process:

- KERASTUNER_TUNER_ID: this variable distinguishes a chief process from a worker process. In case of a chief process, we set it to 'chief'. In case of a worker process, any unique name is allowed, but usually 'tuner0', 'tuner1', etc are used.
- KERASTUNER_ORACLE_IP: this variable identifies the IP of the chief process. All workers should be able to resolve and access this address.
- KERASTUNER_ORACLE_IP: this variable identifies the IP address or hostname that the chief service should run on. All workers should be able to resolve and access this address.
- KERASTUNER_ORACLE_PORT: this variable identifies the port that the chief service should run on. This can be freely chosen, but must be a port that is accessible to the other workers.

Lastly, the same python script can be run on all processes. This means that we can write and launch the same code for all the processes, and Keras Tuner will take care of executing the right part on each of them.

6 Algorithm

The algorithm proposed can be subdivided in three parts:

- Image size hyperparameter optimization
- 'conv_layers' and 'optimizer' hyperparameters optimization
- Average ensemble model

6.1 Image size hyperparameter optimization

The first part of the algorithm addresses the problem of selecting a proper image size. This choice is crucial for the training speed of the models.

Training large images is expensive: the computational cost grows quadratic in the image size, since images have two dimensions. For this reason, we should resort to a bigger image size only if it truly improves the model performance.

We set up a Keras Tuner RandomSearch optimization algorithm to analyze the relationship between image size and validation accuracy[21]. The image sizes considered are: 64x64, 128x128 and 256x256.

It is important to specify that we intentionally kept the number of random trials pretty small (equals to 5 per image size). The reason is that, for each image size, the CPU needs to perform the data preprocessing and cache the resulting data to disk. Then, it needs to retrieve the data for every training epoch. With 256x256 images, the CPU is a huge bottleneck in the learning process, and random search takes long time. After all, the reason for doing this optimization is to check if a large image size is worth using. Also, we set the 'epochs' hyperparameter equals to 50 and the early stopping patience equals to 5 to further speed up the computation.

Figure 5 shows the results of this random search. The image sizes considered appears to be uncorrelated with the validation accuracy, since the models perform the same on average.

Equipped with the new information acquired about image size, we are ready to perform the next step. From now on, all models will be trained on 64x64 images.

[21]Validation accuracy is the proportion of correctly labelled images in the validation set by a certain model.

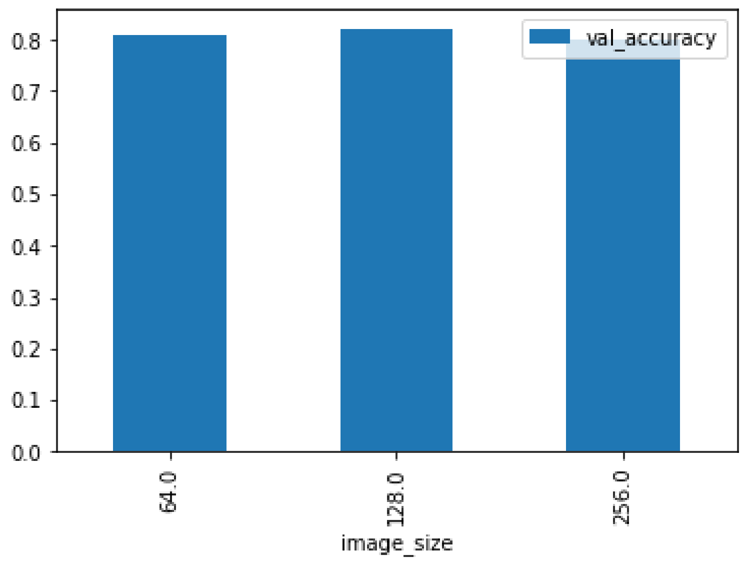


Figure 5: Plot of average validation accuracy grouped by image size.

6.2 ‘conv_layers’ and ’optimizer’ hyperparameters optimization

At this point, we want to analyze two important hyperparameters of our models: ‘conv_layers’ and ‘optimizer’. It’s hard to draw conclusions about the other model hyperparameters since they all rely on the ‘conv_layers’ hyperparameter.

We proceed to run another random search, this time increasing its parameters quite a bit:

- the number of random trials is set to 100;
- the number of epochs is still set to 50;
- the early stopping patience is set to 10.

Turning the number of random trials from 5 to 100 may seem a huge change, however training 64x64 size images is much faster than the other two sizes. The reason is that the prefetch operation on the dataset allows the GPU to run almost continuously, while using bigger image sizes prevent such speed due to the CPU data loading bottleneck.

Figure 6 shows the results of this second random search. The optimal ‘conv_layers’ value appears to be 3, since no significant improvement can be seen after this value. And the best ‘optimizer’ is clearly ‘adam’, which performs much better than the ‘sgd’ counterpart.

With these two hyperparameters set, we proceed to the third and final part of the algorithm.

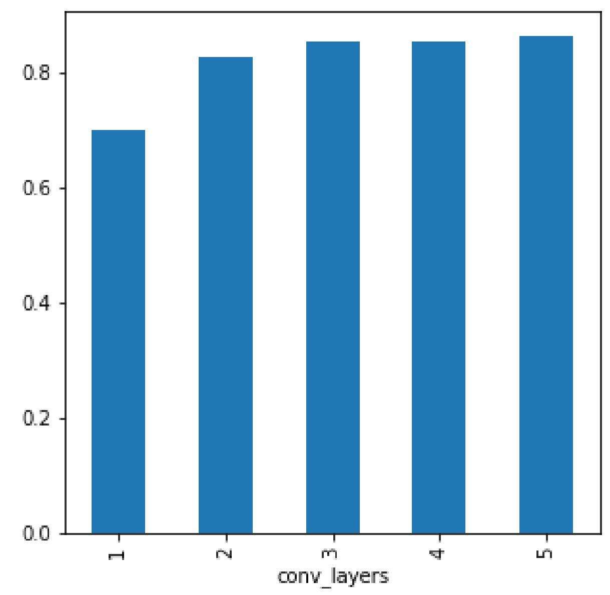
6.3 Average ensemble model

In this last part, our objective is to build a good classifier. The naive approach could be to run the random search algorithm again and choose the model which achieves the best validation accuracy. However, to make our predictions more accurate and robust, an ensemble model is built.

Ensemble model An ensemble model, or simply ensemble, is a predictor that uses the outputs of multiple models to generate its predictions. There are various ways to aggregate the outputs of the models fed to an ensemble, which leads to different types of ensemble. Two of the most common ensemble models are the ‘majority voting ensemble’ and the ‘average ensemble model’.

The ‘majority voting ensemble’ in a model that combines the predictions of the input models choosing the predicted label that occurs the most. In our case, since the predictions are values in the interval (0,1) (where 0 means ‘no_glasses’ and 1 ‘glasses’), we need to round them to the nearest integer. Then, we can count the number of times ‘0’ and ‘1’ occur, and calculate our final prediction.

The ‘average ensemble model’ is a model that aggregates the predictions of the input models averaging them. In this way, if an input models predict a value around 0.5 (which means that it is highly unsure of the corresponding label), this



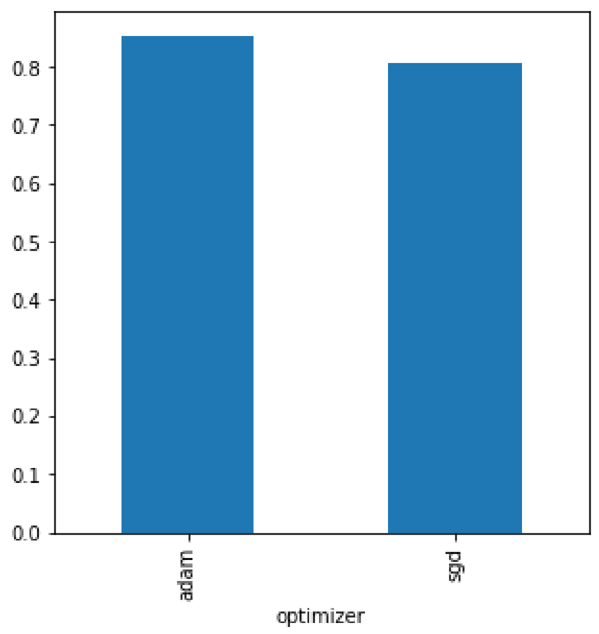


Figure 6: Plot of average validation accuracy grouped by the number of convolution blocks (left) and the optimizer (right).

value will be less influential in the final prediction than a highly confident prediction near 0 or 1. After averaging all input models predictions, we round the obtained value to get our output label.

In this work, both ensembles are built, but we want to focus our attention on the 'average ensemble model', since it is easier to implement with Keras API and should lead to better performance.

First, we run another random search. The parameters of this last run are:

- number of random trials: 150;
- number of epochs: 100;
- early stopping patience: 10.

We also set another parameter that determine the number of models saved: since the whole random search optmization is run on multiple processes (possibly on multiple machines), we need to save the information we want to keep on disk. In this case, we kept the 100 best models (based on validation accuracy).

After the random search run, we retrieve such models and analyze their performance with respect to validation accuracy. Then, we set a threshold to select only the best models to build our 'average ensemble model'. In this case, a threshold of 0.89 validation accuracy is set. Clearly, this threshold can be lifted if more and more random trials are executed, since the probability of finding better models increases. We don't want to select all saved models, since the worst performing ones can worsen the performance of the best models. Definitely, we want to obtain an ensemble model whose predictions are better than any single naive model.

Therefore, we built the 'average ensemble model' using all models with validation accuracy greater than 0.89 and combining their predictions with the Average[22] Keras Layer. A validation accuracy of 0.9065 is obtained, compared with the validation accuracy of 0.8931 of the best naive model.

Finally, we tested this ensemble model on the test set, never seen by any model until now, achieving a test accuracy of 0.8783. We expect to have a lower accuracy on test set (compared to the validation set), since the models are chosen based on their validation accuracy, and therefore the ensemble validation accuracy is surely an upper bound of the true accuracy on new data.

We also built a 'majority voting ensemble' to make a comparison with our ensemble: we expect to get a test accuracy that is lesser or equal than the 'average ensemble model' accuracy. The reason is that the 'average ensemble model' uses

[22]`https://www.tensorflow.org/api_docs/python/tf/keras/layers/Average`

the uncertainty of the input models predictions when it calculate its output label, while the ‘majority voting ensemble’ doesn’t, since he rounds all predictions to 0 or 1 and counts the most occurring label. After evaluating the ‘majority voting ensemble’ on the test set, we achieve the same test accuracy of 0.8783.

In order to get a better estimate of the test accuracy, we could run a cross-validation technique that should lower the variance on the test accuracy estimate. However, in this work we are thinking in terms of large datasets, and if a dataset is large enough and the data are shuffled correctly, a single test accuracy estimate should be fine. In our environment (Colab notebook) we couldn’t use cross-validation, since the whole algorithm already takes 5-6 hours to complete.

6.4 Wrap-up

At this point we may wonder if we can do better: a test accuracy of 0.8783 on a binary classification task is not bad, but it’s not that good either.

One way to answer this question is to look to the wrongly labelled images. In fact, looking to our model errors allows us to get an idea of what it is doing wrong. Figure 7 shows a sample of five images wrongly labelled by the ensemble model. We can see that, even if there are some true errors in the label predictions, some images (e.g. the first one) seem to be correctly classified. Therefore, some errors in the true labelling procedure were made. Indeed, even the author of the dataset draws attention to the presence of missclassified data, since the labelling procedure was carried out by the GAN. These label errors can be interpreted, in a statistical learning framework, as stochasticity in the label values. And all the errors made by the GAN contributes to the statistical Bayer error. Therefore, we will never be able to retrieve a perfect classifier.

Nevertheless, we can train more models and hope to achieve greater accuracy, as there are still some true ‘wrong predicted images’.

This concludes our image analysis.

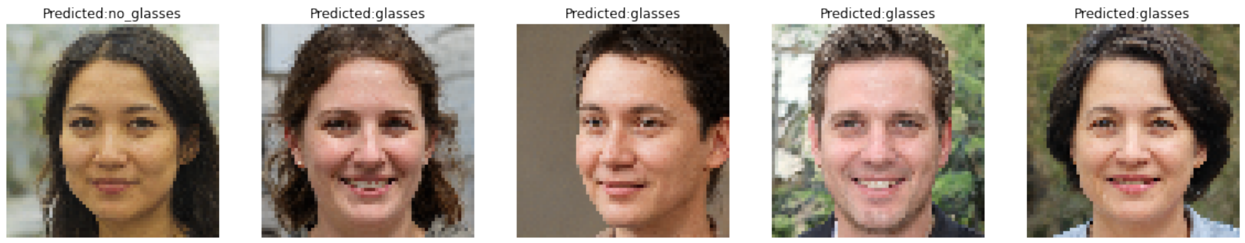


Figure 7: A sample of five images wrongly labelled by the ‘average ensemble model’.

7 Extra: 512-dimensional latent vectors analysis

In this extra section we want to analyze the 512-dimensional latent vectors which are part of the original dataset but were not considered in the previous analyzes. This section only wants to provide a comparative result with respect to the image analysis, therefore no particular techniques are used to deal with large CSV files. Since our CSV file is small, we assume that it can fit entirely in memory and proceed to use it as it is.

The latent vectors can be seen as a compact representation of the images from the GAN point-of-view. Since the GAN uses only these vectors to generate the images, we expect the presence or absence of glasses to be somehow encoded within these vectors.

Therefore, we proceed to build a toy neural network, to test its performance: after just five epochs, the validation accuracy is almost 100%. We test the toy model on the test set, achieving the same almost 100% accuracy.

This result should not be surprising. The images are generated and labelled by the GAN, and since the latent vectors are the GAN compact representation of the images, we should expect to obtain the same labels by analyzing the images from the GAN point-of-view. Indeed, we are clearly overfitting the data, and the built neural network is not useful at all. In fact, remember that when we analyzed the images in the previous section, some of them were missclassified. But this network, looking from the GAN perspective, is unable to pinpoint where the mistakes were made by the GAN itself.

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, page 2672–2680, Cambridge, MA, USA, 2014. MIT Press.

[2] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):60, Jul 2019.

[3] Wei Zhang, Kazuyoshi Itoh, Jun Tanida, and Yoshiki Ichioka. Shift-invariant pattern recognition neural network and its optical architecture. In *Proceedings of Annual Conference of the Japan Society of Applied Physics.*, 1988.

[4] Wei Zhang, Kazuyoshi Itoh, Jun Tanida, and Yoshiki Ichioka. Parallel distributed processing model with local space-invariant interconnections and its optical architecture. *Appl. Opt.*, 29(32):4790–4797, Nov 1990.

[5] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, page 448–456. JMLR.org, 2015.

[6] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.