Deep learning: Binary classification system for face property detection

Alba Ordoñez

April 2019

Contents

Abstract

This report describes the strategy used for developing a binary classification algorithm in a data challenge where the main goal was to detect whether an image contained some property or not. Given that convolution neural networks are the state of the art methods used for this kind of task, I built a classification algorithm based on these architectures.

1. Introduction

The goal of this challenge was to develop a binary classification algorithm to detect whether an image contained some property or not. The performance criterion was calculated from the average accuracy for each class and ranged between 0 and 1. The higher the score the better the performance.

A special procedure was followed in this challenge: it was divided into 2 phases. In the first one, I was provided a training set which contained raw images and their corresponding labels and another validation dataset which contained only raw images. In the second phase (which was launched only few hours before the deadline) I was provided a test set. The goal of the challenge was to run the algorithm developed in the first phase on this test set. This meant that I had limited time to fine tune the developed algorithm on this last dataset.

Another interesting feature of this challenge was that the validation set provided in the first phase (without the corresponding labels) was already included in the training set. Consequently, I could not completely rely on only this dataset for building adequate models to solve the problem.

The adopted strategy to solve this challenge was as follows: First, I investigated the training set and given its properties I built a proper dataset that could be used for building a suitable classification algorithm. For computer vision tasks, state-of-the art approaches are based on using convolution neural networks (CNNs). For this reason, I decided to experiment with several CNN models, starting from very simple ones and going to more complicated ones. On more complex architectures, the idea was to use transfer learning, as described later on in this report. After comparing the different tested CNN architectures, the final step consisted of choosing one model and train it more thoroughly with the aim of accurately predicting the labels on the test set provided during the second phase.

2. Investigation of the training set

The provided training set was composed of RGB images of size 56 x 56 x 3 and there were 116157 images in this dataset. Another file containing the labels associated to each image was also provided.

2.1. Identifying duplicate images

It turns out that the provided training set contained few duplicate images. In order to detect them, the solution I chose was to calculate a hash for each individual image in the training set, store the hashes in a list to find out the repeated ones. The are several perceptual hashing algorithms but the one I picked here was the dHash algorithm described in more detail in [2]: The idea behind it was to compute the difference in brightness between adjacent pixels, identifying the relative gradient direction. The algorithm can be actually decomposed into 4 parts: grayscale the image, shrink the image to a common size, compare adjacent pixels, convert the difference into bits.

Figure 1 shows a couple of duplicate examples that were found by using the dHash algorithm.

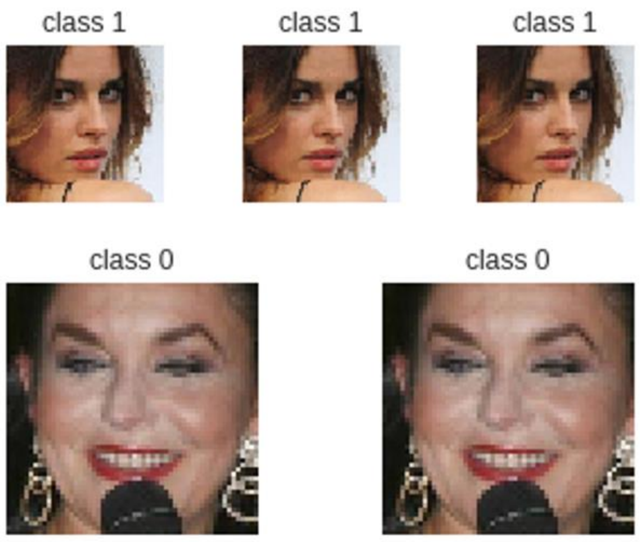


Figure 1: Duplicate examples found by the dHash algorithm

The total number of duplicate images that were found with the dHash algorithm was 254. However, by using another hashing technique one might find a different value.

It also turns out that there were few detected duplicate images that were associated to labels 0 and 1. For example, I found one exact duplicate having this characteristic (Figure 2). The 8 other images that were found are displayed in Figure 3. Note again that by using another hashing methodology, other examples might be found. I also displayed the difference between the images because I was wondering if this would enable me to find out the hidden property behind the faces labelled as 1. However, I did not manage to conclude regarding that. In the end, these special duplicates made me conclude that the labelling was performed automatically and there was some errors in it. For this reason, I decided to build a new training set that excluded all the duplicated images. In addition, I excluded all the images that were labelled as 0 and 1. Finally, I ended up with 115894 image in the training set (i.e. I removed 263 images from the original dataset composed of 116157 samples).

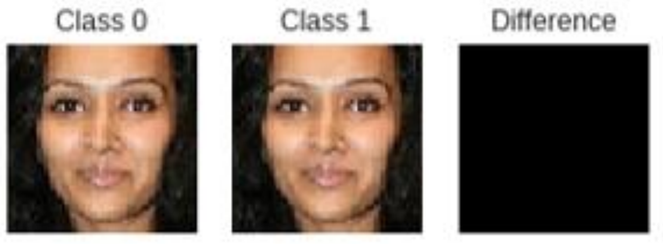


Figure 2: Exact duplicate labelled as 0 and 1

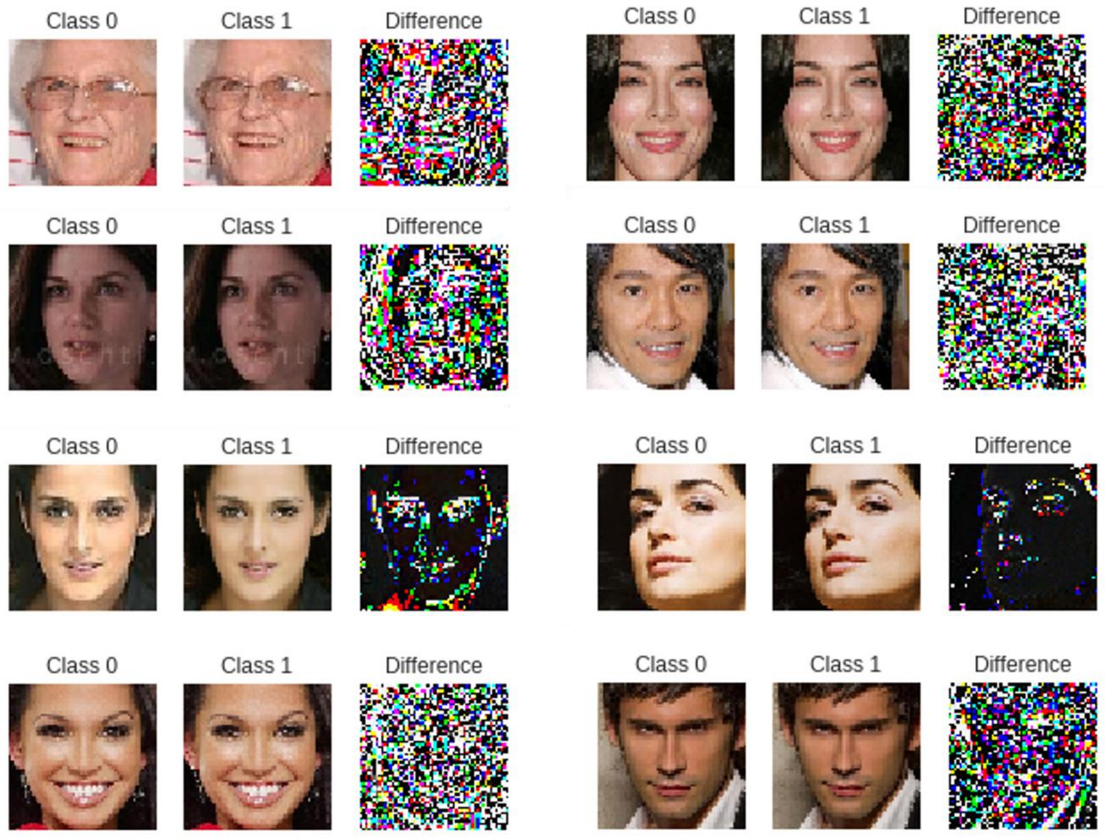


Figure 3: Duplicates labelled as 0 and 1

2.2. Building train, validation and test sets for modelling

To be able to experiment with different CNN models, I split the new generated training samples according to the below proportions:

- **Training set**: 59.5 %
- **Validation**: 10.5 %
- **Test set**: 30 %.

Note that I have used Keras to set up CNN models and this framework has a very useful utility called **ImageDataGenerator** that can generate batches of tensor image data with real-time data augmentation and processing. For example, if I need to scale all the images by a rescaling factor (e.g. 1/255), this procedure does it without any need to store the whole sets of images in memory provided you chose an adequate batch size. This a very important concept particularly when training deep learning models on GPUs (known for having less memory than CPUs). I have used generators computed from **ImageDataGenerator** to build all the models that are presented in the following section.

3. Experimenting with several CNN models

To keep track of the different tested models, I saved them into .h5 format together with an associated csv table containing the information of the learning curves associated to each model for each epoch. Note that for each CNN architecture, I decided to save the best model together with the one obtained at the last epoch. The loss function and the weighted accuracy of the validation set were controlled such that the best model would have the minimal loss function and the maximum averaged accuracy. Every five epochs, I also did a prediction on my test set to evaluate the associated averaged accuracy.

3.1. CNN model from scratch

Inspired from [1], I started building a basic CNN model and tried to further improve the obtained results by using regularization and image augmentation as described below.

3.1.1. Basic CNN (B1)

The basic CNN model contains three convolutional layers, coupled with max pooling for auto-extraction of features from the images and also for down-sampling the output convolution feature maps (Figure 4). Note that I have used ReLU as non linearity for activating the convolutions. However, it has to be pointed out that nowadays it also becomes popular to use LeakyReLU to avoid the “dead ReLU” problem blocking the learning in the activation because of 0 in the negative part. After the convolution and pooling phases, a flatten layer is used to flatten out 128 of the 5 x 5 feature maps that I get as output from the third convolution layer. This is fed to dense (or fully connected) layers and the last one is activated by a sigmoid function to get the final prediction of whether the image should have the property (1) or not (0).

In total 1 723 009 parameters are learned as shown in the summary given in Figure 4.

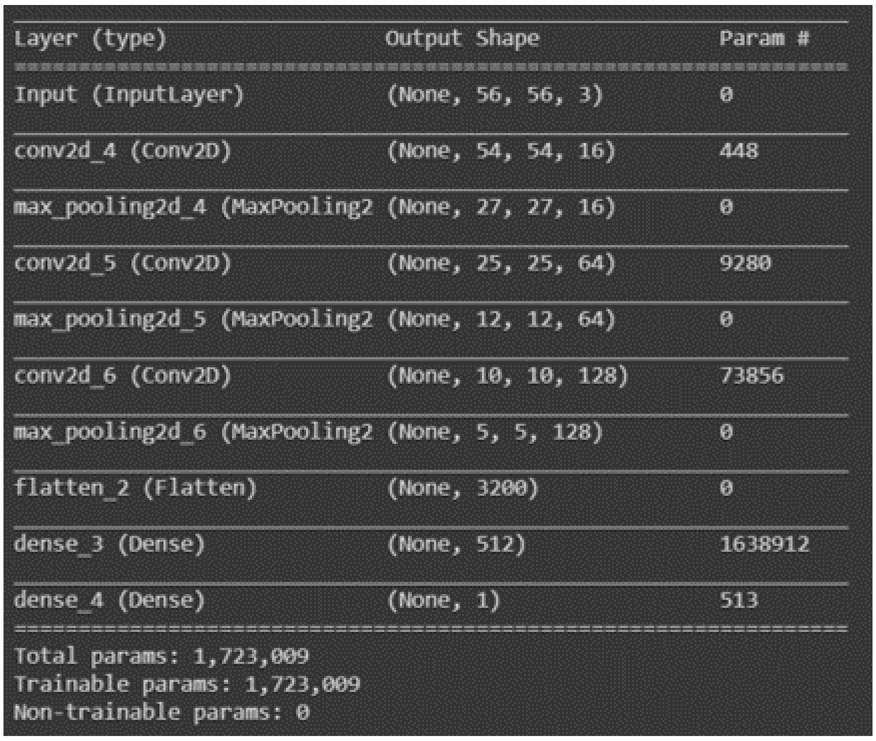

```
Layer (type)                 Output Shape              Param #
=================================================================
Input (InputLayer)           (None, 56, 56, 3)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 54, 54, 16)        448
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 27, 27, 16)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 25, 25, 64)        9280
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 10, 10, 128)       73856
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 5, 5, 128)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 3200)              0
_________________________________________________________________
dense_3 (Dense)              (None, 512)               1638912
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 513
=================================================================
Total params: 1,723,009
Trainable params: 1,723,009
Non-trainable params: 0
```

Figure 4: Summary of the basic CNN

For the modelling, I use a batch size of 124 and the training data has a total of 68957 samples, which indicates that there will be a total of 556 iterations per epoch. I train the model for a total of 15 epochs and validate it subsequently on the validation set of 12168 images. I use Adam for the optimization algorithm for the gradient descent and the default learning rate of 10^{-3}. The loss function that is used is the binary cross-entropy.

The only pre-processing that I apply to the data via **ImageDataGenerator** is a re-scaling of the pixels such that their values are comprised between 0 and 1 (original pixel values range from 0 to

255). The reason for that is because neural networks tend to work better with data having small values.

In the plot showing the model accuracies and errors, it clearly appears that after 3-4 epochs the model starts overfitting on the training data with a steep increase in the validation loss and in the training accuracies. There is not much change in the curve of the validation data which stabilizes around 78-79% average accuracy. Let’s try to further improve upon this model.

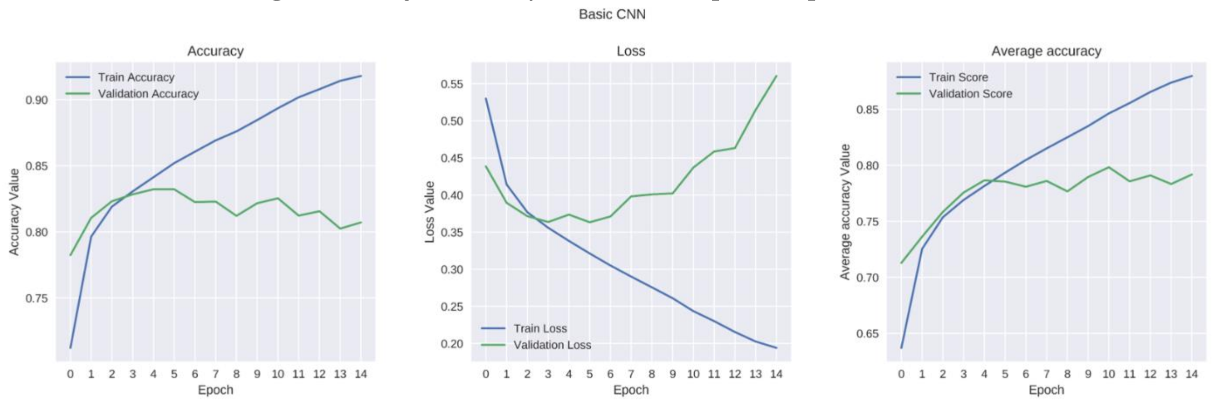
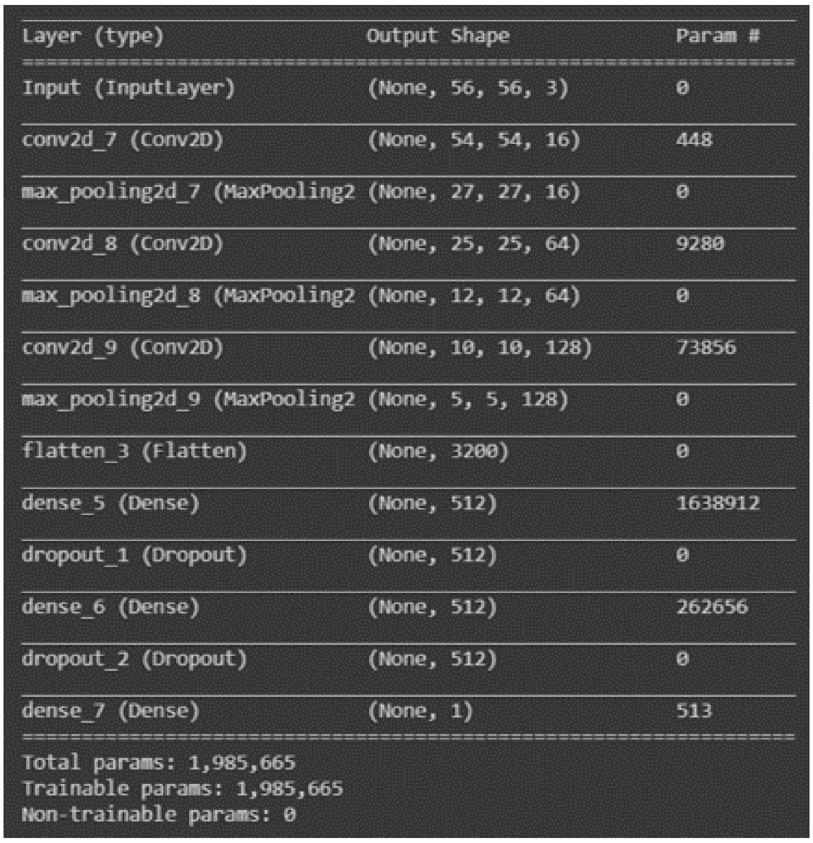


Figure 5: Model accuracy and errors of the basic CNN

3.1.2. Basic CNN model with regularization (B2)

I added one more dense hidden layer to the previous model. Besides this, I also included a dropout of 0.3 after each hidden dense layer to enable regularization. This procedure randomly masks neurons from a layer by setting their contribution to zero (here, there are 30% of the neurons of



```
Layer (type)                 Output Shape              Param #
=================================================================
Input (InputLayer)           (None, 56, 56, 3)         0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 54, 54, 16)        448
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 27, 27, 16)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 25, 25, 64)        9280
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 12, 12, 64)        0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 10, 10, 128)       73856
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 5, 5, 128)         0
_________________________________________________________________
flatten_3 (Flatten)          (None, 3200)              0
_________________________________________________________________
dense_5 (Dense)              (None, 512)               1638912
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_7 (Dense)              (None, 1)                 513
=================================================================
Total params: 1,985,665
Trainable params: 1,985,665
Non-trainable params: 0
```

Figure 6: Basic CNN model with regularization

the dense layers that are masked). As shown in Figure 6, I end up learning more parameters (1 985 665) compared to the previous model since I added an extra dense layer.

It clearly appears from the learning curves that with 15 epochs I still end up overfitting the model even though I have a better validation average accuracy around 82 %. The reason for model overfitting might be related to the fact that I may not have enough training data and the model keeps seeing the same instances over time across each epoch. A solution for that would be to implement a data augmentation strategy where the existing training data would also include slight variations of the original images as described in the next section.

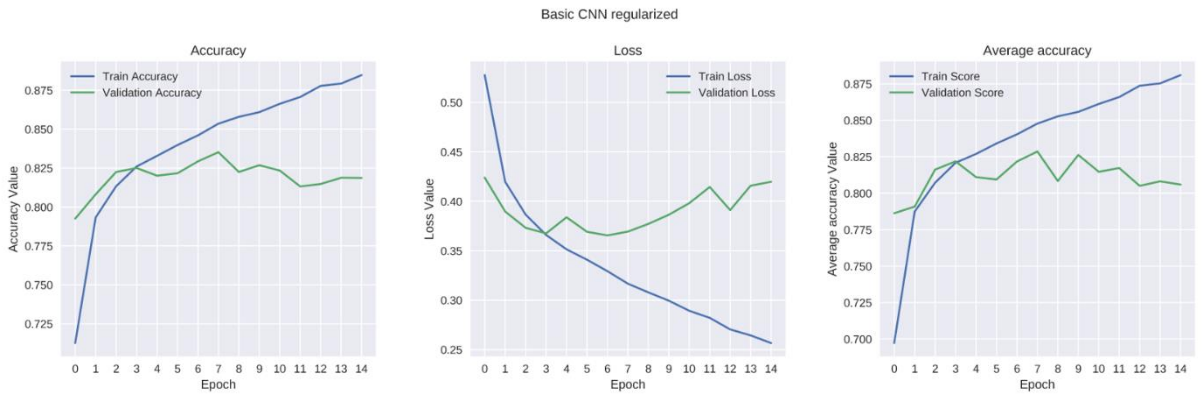


Figure 7: Model accuracy and errors of the basic CNN regularized

3.1.3. Basic CNN model with image augmentation (B3)

Let’s improve upon the regularized CNN model (B2) by adding in more data using a proper image augmentation strategy.

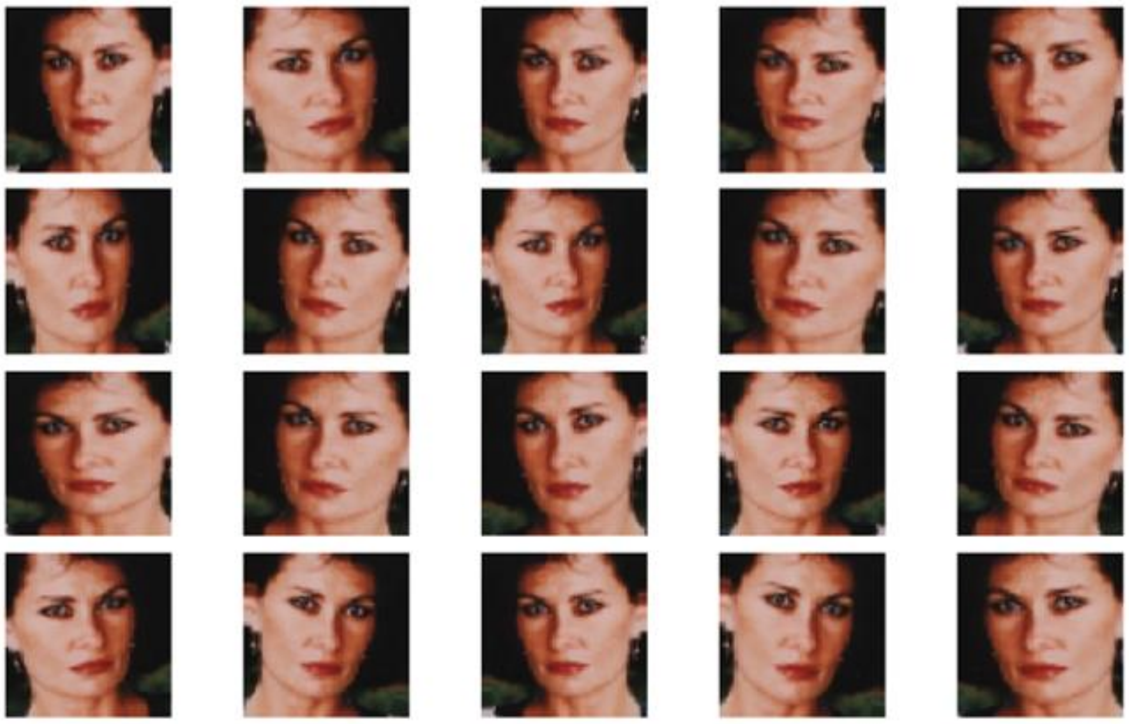


Figure 8: Augmented images

The idea behind image augmentation is to take in images from the training dataset and apply transformation operations to them, such as horizontal flip, rotations, translations, zooming, etc. to produce altered versions of existing images. For this, I use **ImageDataGenerator** and decide to include zooms of the original images (factor of 0.1 used), slightly rotated (by 10 degrees) and mirrored versions of them (horizontal flips) as illustrated in Figure 8.

I also decreased the learning rate to 10^{-4} to make sure that I do not get stuck in local minima during the learning phase. The learning curves are presented in Figure 10: We clearly observe that this model does not overfit as the accuracies and the losses of the training and validation sets are very similar. Consequently, this model generalizes much better compared to the previous two ones. Note that the validation average accuracy is around 80-81%, which is slightly below the accuracy obtained with the previous regularized model.

Let's now try to see whether we can improve these models by using transfer learning.

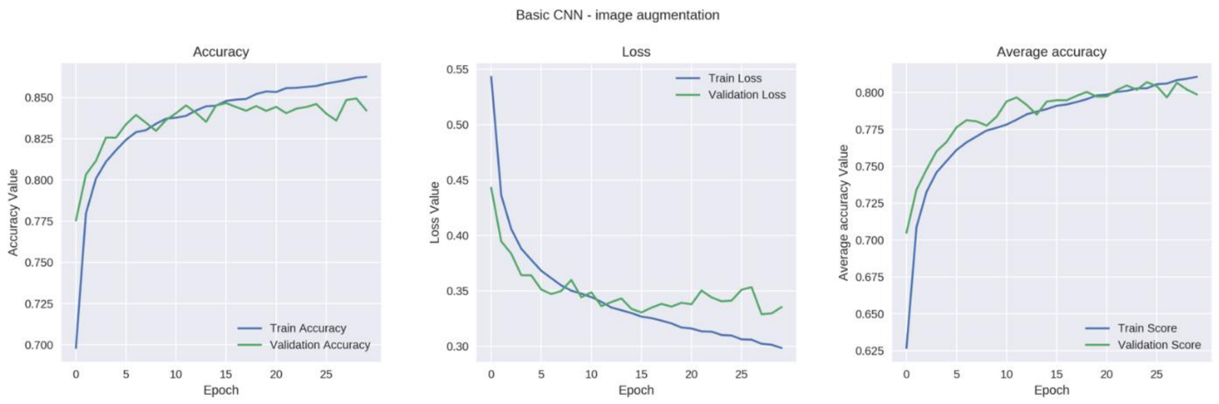


Figure 10 : Model accuracy and errors of the basic CNN with image augmentation

3.2. CNN using transfer learning

Building effective CNNs require estimating millions of parameters using a large amount of labelled training data. However, computational resources and large amounts of labelled data may not always be available. In addition, training CNNs from scratch can also be complicated due to overfitting and convergence problems, that is why it is often required to properly optimize many learning parameters of the network layers to achieve an acceptable convergence. To address all these issues, transfer learning can be used. This procedure consists of using pre-trained state of the art CNNs and transfer the acquired knowledge on a large amount of labelled source data (e.g. ImageNet database containing over a million images with 1000 categories) to fine tune CNN parameters for another target classification problem (e.g. face property detection).

Depending on the volume of available target data, different strategies of transfer learning exist. The first one consists of freezing all the convolution and pooling layers and only changing the classification unit containing the fully connected layers (including the last one activated by the sigmoid). Usually, this technique is applied when the volume of target data is small. The second approach, which is more adequate when having more training data consists of freezing a fewer number of convolution and pooling layers. For the remaining layers to train, pre-trained weights of the non-frozen layers can be used as initialization to perform gradient descent from there. When an important amount of target data is available, one could use the last approach of transfer learning. This consists of training the network from scratch using the pre-trained weights as initialization, instead of using random weights.

In the following, we will apply the different strategies of transfer learning based on the well known ResNet-50 model [3]. The main principle behind ResNets is that layers are reformulated as residual blocks. The idea of these blocks is that the input x goes though some convolution layers to get the output $f(x)$. This result is then added to the original (identity) input x to obtain $y(x) = f(x) + x$. The advantage of this structure is that one can train deeper networks without having the problem of vanishing gradient. As illustrated in Figure 11, compared to other networks such as GoogLeNet, ResNet-50 provides more accurate results. Moreover, ResNet-50 appears to require a smaller amount of operations and weight computation with respect to models like VGG-16 and VGG-19.

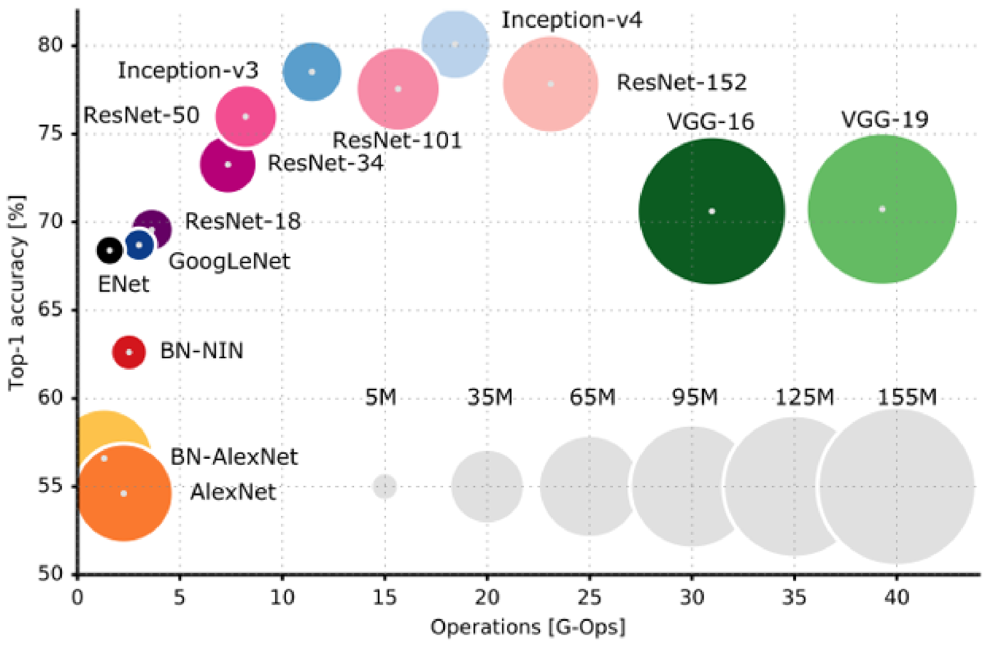
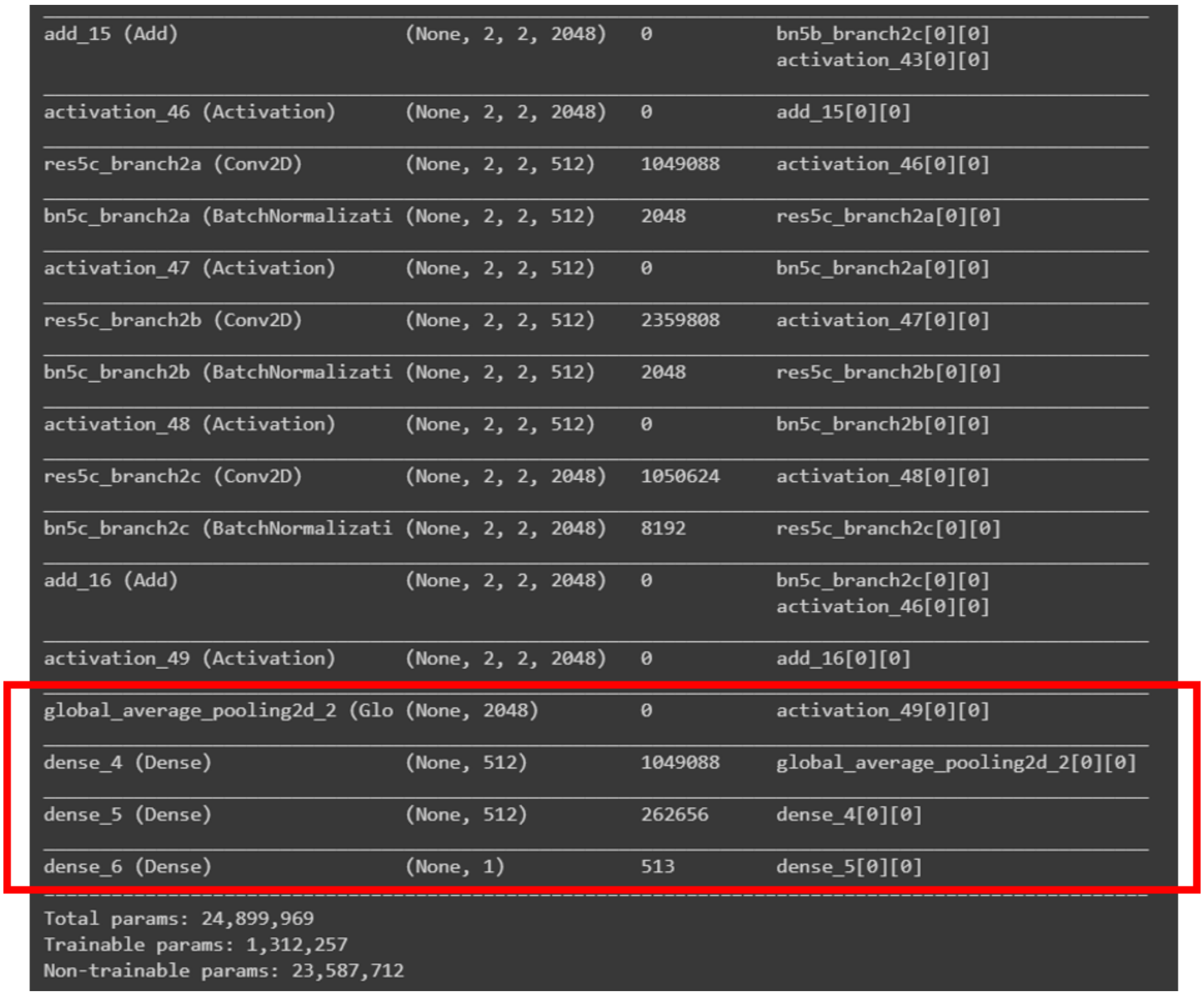


Figure 11:Top-1 crop accuracy versus amount of operations required for a single forward pass in multiple popular CNNs (extracted from [4]). The radius of the circles around each model is proportional to the total computed weights.

3.2.1. Fine tuning the classification unit with image augmentation (TL1)

The modified classification unit is highlighted with a red rectangle in Figure 12: After freezing all the layer weights of the ResNet-50 model, the last activation feature map is fed into a global average pooling layer (reducing each feature map to a single number by taking the average of all the values). This output is then given to two fully connected layers before using the last classification layer characterized by the sigmoid activation function. It appears from Figure 12, that 1 312 257 parameters are trained whereas 24 899 969 parameters are frozen. Note that in this case, I applied a re-scaling to the pixels using the pre-processing function associated to ResNet-50. This function converts the images from RGB to BGR, then zero-centre each colour channel with respect to the ImageNet dataset without re-scaling. I also applied the same image augmentation strategy used for the CNN described in section 3.1.3 and I kept the same learning rate (10^{-4}) for the gradient descent.



```
add_15 (Add)                    (None, 2, 2, 2048)   0         bn5b_branch2c[0][0]
                                                               activation_43[0][0]
________________________________________________________________________________________
activation_46 (Activation)      (None, 2, 2, 2048)   0         add_15[0][0]
________________________________________________________________________________________
res5c_branch2a (Conv2D)         (None, 2, 2, 512)    1049088   activation_46[0][0]
________________________________________________________________________________________
bn5c_branch2a (BatchNormalizati (None, 2, 2, 512)    2048      res5c_branch2a[0][0]
________________________________________________________________________________________
activation_47 (Activation)      (None, 2, 2, 512)    0         bn5c_branch2a[0][0]
________________________________________________________________________________________
res5c_branch2b (Conv2D)         (None, 2, 2, 512)    2359808   activation_47[0][0]
________________________________________________________________________________________
bn5c_branch2b (BatchNormalizati (None, 2, 2, 512)    2048      res5c_branch2b[0][0]
________________________________________________________________________________________
activation_48 (Activation)      (None, 2, 2, 512)    0         bn5c_branch2b[0][0]
________________________________________________________________________________________
res5c_branch2c (Conv2D)         (None, 2, 2, 2048)   1050624   activation_48[0][0]
________________________________________________________________________________________
bn5c_branch2c (BatchNormalizati (None, 2, 2, 2048)   8192      res5c_branch2c[0][0]
________________________________________________________________________________________
add_16 (Add)                    (None, 2, 2, 2048)   0         bn5c_branch2c[0][0]
                                                               activation_46[0][0]
________________________________________________________________________________________
activation_49 (Activation)      (None, 2, 2, 2048)   0         add_16[0][0]
________________________________________________________________________________________
global_average_pooling2d_2 (Glo (None, 2048)         0         activation_49[0][0]
________________________________________________________________________________________
dense_4 (Dense)                 (None, 512)          1049088   global_average_pooling2d_2[0][0]
________________________________________________________________________________________
dense_5 (Dense)                 (None, 512)          262656    dense_4[0][0]
________________________________________________________________________________________
dense_6 (Dense)                 (None, 1)            513       dense_5[0][0]
========================================================================================
Total params: 24,899,969
Trainable params: 1,312,257
Non-trainable params: 23,587,712
```

Figure 12: Summary of the last layers of the ResNet50 model where the classification unit (red rectangle) was fine tuned. Only 1 312 257 parameters are set to trainable, the rest are frozen.

The learning curves associated to this model are presented in Figure 13. We get a model with a very low validation average accuracy below 65 % and a relatively bad training average accuracy (<70 %). In addition, the model clearly overfits as there is a significant gap between the training and validation accuracies even after the first epoch. The overfitting has not been resolved by the image augmentation.

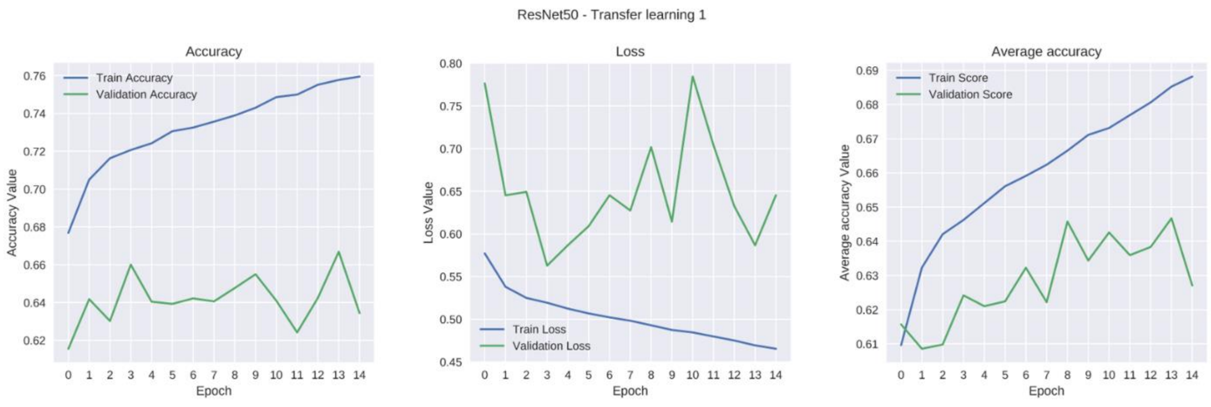
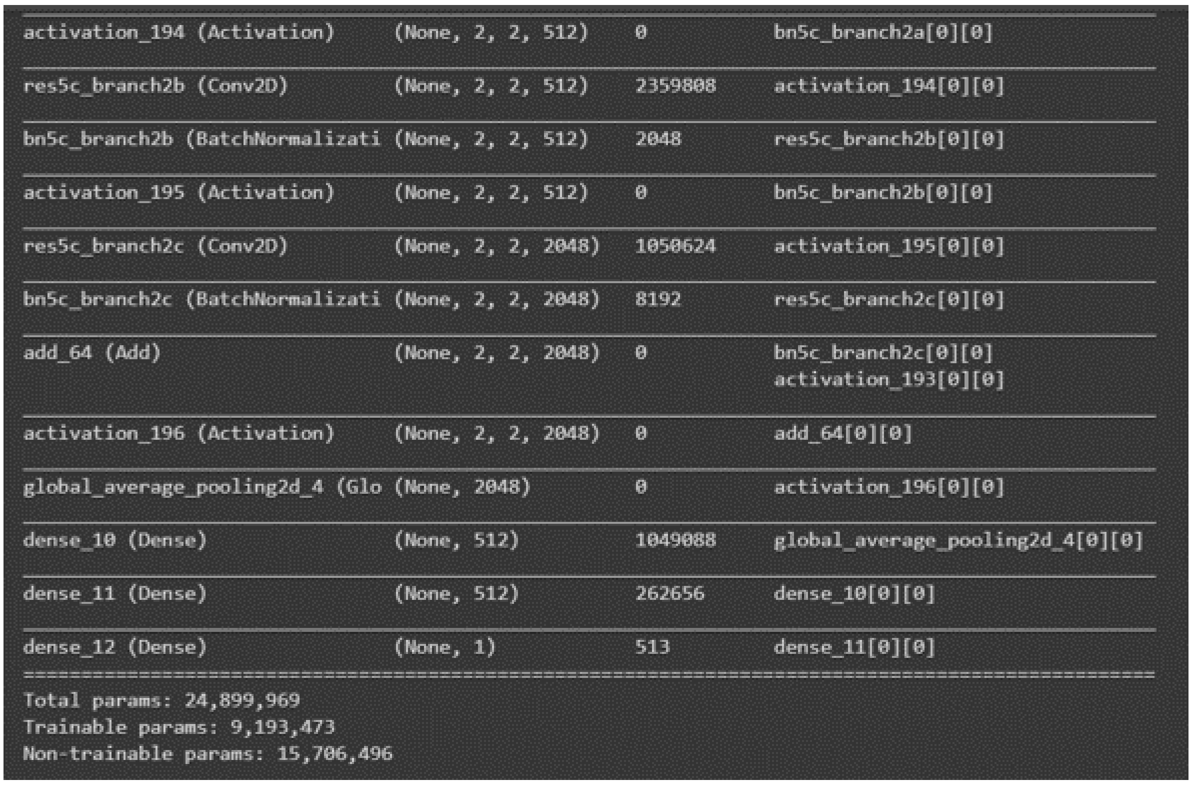


Figure 13: Model accuracy and errors of the pre-trained CNN using fine tuning of the classification unit together with image augmentation.

This model is so far the worst one. Including some regularization after each hidden dense layer would have probably improved the result together with a smaller learning rate. Another option could have been to start from weights which were not extracted from ImageNet but from a more appropriate database such as FaceNet [5].

3.2.2. Freezing 158 layers and add image augmentation (TL2)

I decided to do not train the first 158 layers of the ResNet-50 network (i.e. I froze all the layers before the block res5b_branch2b and unfroze the rest) to see whether I could improve the results obtained with the previous model (Figure 14).



```
activation_194 (Activation)     (None, 2, 2, 512)    0          bn5c_branch2a[0][0]
____________________________________________________________________________________________
res5c_branch2b (Conv2D)         (None, 2, 2, 512)    2359808    activation_194[0][0]
____________________________________________________________________________________________
bn5c_branch2b (BatchNormalizati (None, 2, 2, 512)    2048       res5c_branch2b[0][0]
____________________________________________________________________________________________
activation_195 (Activation)     (None, 2, 2, 512)    0          bn5c_branch2b[0][0]
____________________________________________________________________________________________
res5c_branch2c (Conv2D)         (None, 2, 2, 2048)   1050624    activation_195[0][0]
____________________________________________________________________________________________
bn5c_branch2c (BatchNormalizati (None, 2, 2, 2048)   8192       res5c_branch2c[0][0]
____________________________________________________________________________________________
add_64 (Add)                    (None, 2, 2, 2048)   0          bn5c_branch2c[0][0]
                                                                activation_193[0][0]
____________________________________________________________________________________________
activation_196 (Activation)     (None, 2, 2, 2048)   0          add_64[0][0]
____________________________________________________________________________________________
global_average_pooling2d_4 (Glo (None, 2048)         0          activation_196[0][0]
____________________________________________________________________________________________
dense_10 (Dense)                (None, 512)          1049088    global_average_pooling2d_4[0][0]
____________________________________________________________________________________________
dense_11 (Dense)                (None, 512)          262656     dense_10[0][0]
____________________________________________________________________________________________
dense_12 (Dense)                (None, 1)            513        dense_11[0][0]
============================================================================================
Total params: 24,899,969
Trainable params: 9,193,473
Non-trainable params: 15,706,496
```

Figure 14 : Summary of the last layers of the ResNet50 model where part of convolutional layers and the classification unit were fine tuned. The are 9 193 473 parameters that are set to trainable, the rest are frozen.

With such configuration, I obtained the results displayed in Figure 15. The validation accuracy has now improved to above 75 %. However, the presence of overfitting clearly appears due to the gap between the training and validation accuracies after the fifth epoch. Let's try to see whether this can be improved by training from scratch the ResNet-50 network by using pre-trained weights as initialization for all the layers of the architecture.

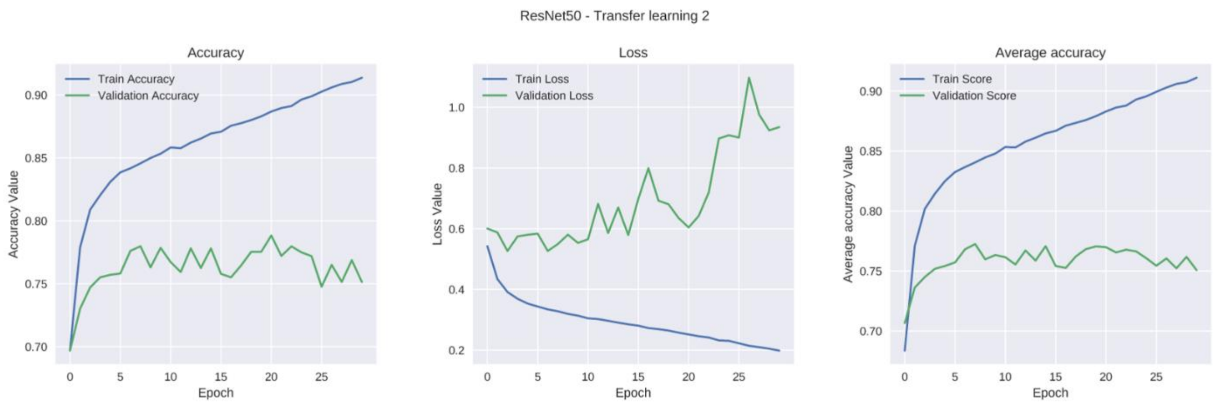
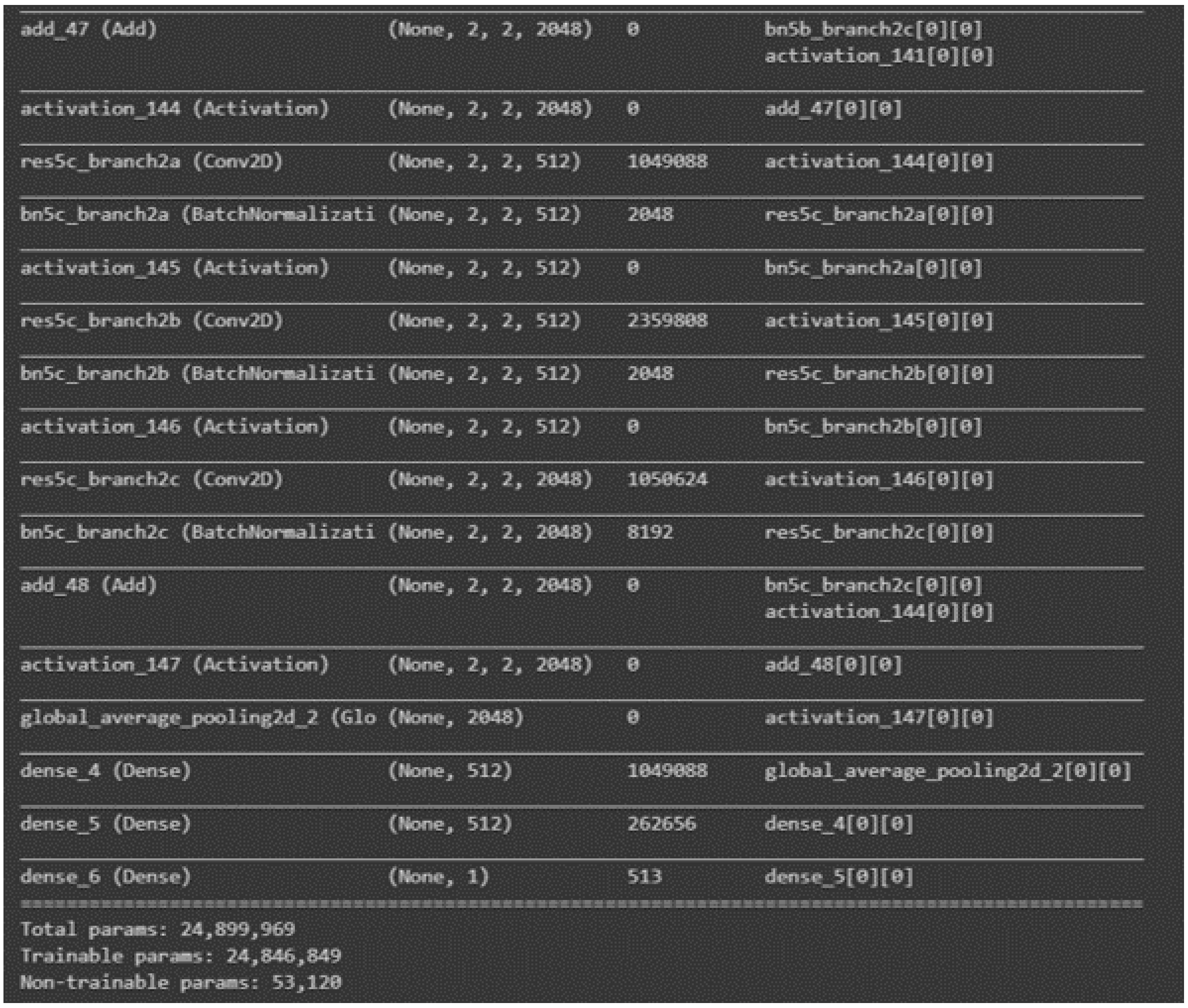


Figure 15: Model accuracy and errors of the pre-trained CNN where all the layers after the block res5b_branch2b have been unfrozen.

3.2.3. Training from scratch with image augmentation (TL3)

Finally, the entire ResNet-50 network was trained from scratch using the pre-trained weights derived from ImageNet. There are 24 846 849 parameters to train.



```
add_47 (Add)                    (None, 2, 2, 2048)   0          bn5b_branch2c[0][0]
                                                                activation_141[0][0]
________________________________________________________________________________________________
activation_144 (Activation)     (None, 2, 2, 2048)   0          add_47[0][0]
________________________________________________________________________________________________
res5c_branch2a (Conv2D)         (None, 2, 2, 512)    1049088    activation_144[0][0]
________________________________________________________________________________________________
bn5c_branch2a (BatchNormalizati (None, 2, 2, 512)    2048       res5c_branch2a[0][0]
________________________________________________________________________________________________
activation_145 (Activation)     (None, 2, 2, 512)    0          bn5c_branch2a[0][0]
________________________________________________________________________________________________
res5c_branch2b (Conv2D)         (None, 2, 2, 512)    2359808    activation_145[0][0]
________________________________________________________________________________________________
bn5c_branch2b (BatchNormalizati (None, 2, 2, 512)    2048       res5c_branch2b[0][0]
________________________________________________________________________________________________
activation_146 (Activation)     (None, 2, 2, 512)    0          bn5c_branch2b[0][0]
________________________________________________________________________________________________
res5c_branch2c (Conv2D)         (None, 2, 2, 2048)   1050624    activation_146[0][0]
________________________________________________________________________________________________
bn5c_branch2c (BatchNormalizati (None, 2, 2, 2048)   8192       res5c_branch2c[0][0]
________________________________________________________________________________________________
add_48 (Add)                    (None, 2, 2, 2048)   0          bn5c_branch2c[0][0]
                                                                activation_144[0][0]
________________________________________________________________________________________________
activation_147 (Activation)     (None, 2, 2, 2048)   0          add_48[0][0]
________________________________________________________________________________________________
global_average_pooling2d_2 (Glo (None, 2048)         0          activation_147[0][0]
________________________________________________________________________________________________
dense_4 (Dense)                 (None, 512)          1049088    global_average_pooling2d_2[0][0]
________________________________________________________________________________________________
dense_5 (Dense)                 (None, 512)          262656     dense_4[0][0]
________________________________________________________________________________________________
dense_6 (Dense)                 (None, 1)            513        dense_5[0][0]
================================================================================================
Total params: 24,899,969
Trainable params: 24,846,849
Non-trainable params: 53,120
```

Figure 16: Summary of the last layers of the ResNet50 model trained from scratch.

The results have now improved significantly compared to the previous two models. The overfitting phenomenon has disappeared, and we achieve the best average accuracy so far (around

85 %). Nonetheless, this is the most expensive model to train in terms of weight computations and operations.

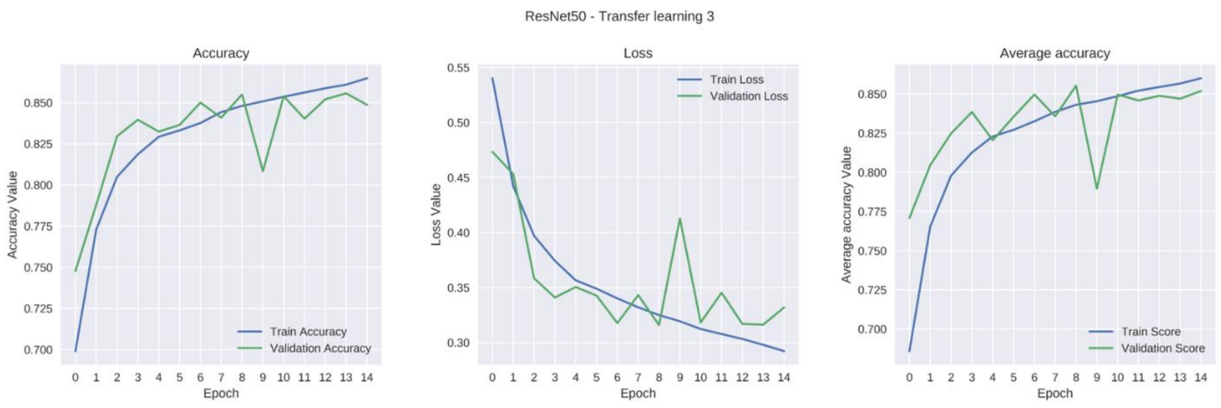


Figure 17 : Model accuracy and errors of the ResNet-50 CNN trained from scratch.

3.3. Comparison of the models

To evaluate the different models that were built, I tested them on the test dataset as defined in section 2.2. It clearly appears that using data augmentation and regularization for the basic CNN models have helped to improve the score. Regarding, the models derived from transfer learning, the one that performs better is the model that was trained from scratch by using as initialization the pre-trained weights from the ImageNet database.

Model	**Basic cnn B1**	**Basic cnn regularized B2**	**Basic cnn reg + aug B3**	**Res-Net50 TL1**	**Res-Net50 TL2**	**Res-Net50 TL3**
Averaged accuracy	0.8233	0.8412	0.8469	0.6092	0.746	0.8487

Table 1:Averaged accuracies obtained on test data with the different models

The worst model is the one where I only fine-tuned the classification unit from ResNet-50 (TL1) and the best model is the ResNet-50 trained from scratch (TL3). The confusion matrices displayed in Figure 18 show some interesting results. For example, it appears from the diagonal of these matrices that the worst model (TL1) gives the best predictions of the labels from class 1 compared to all the other models. The best model (TL3) gives the best predictions associated to class 0. Perhaps, a combination of the two models would have helped improving the results. Regarding the more basic models, techniques such as regularization and image augmentation (model B3) definitely helped improving the confusion matrices rates obtained on the test set compared to the simpler model B1.

All in all, model B3 is performing almost as well as model TL3. However, in terms of computational efficiency, the basic model outperforms. Consequently, I have finally decided to use model B3 to predict the final test set provided during the second phase of the challenge.

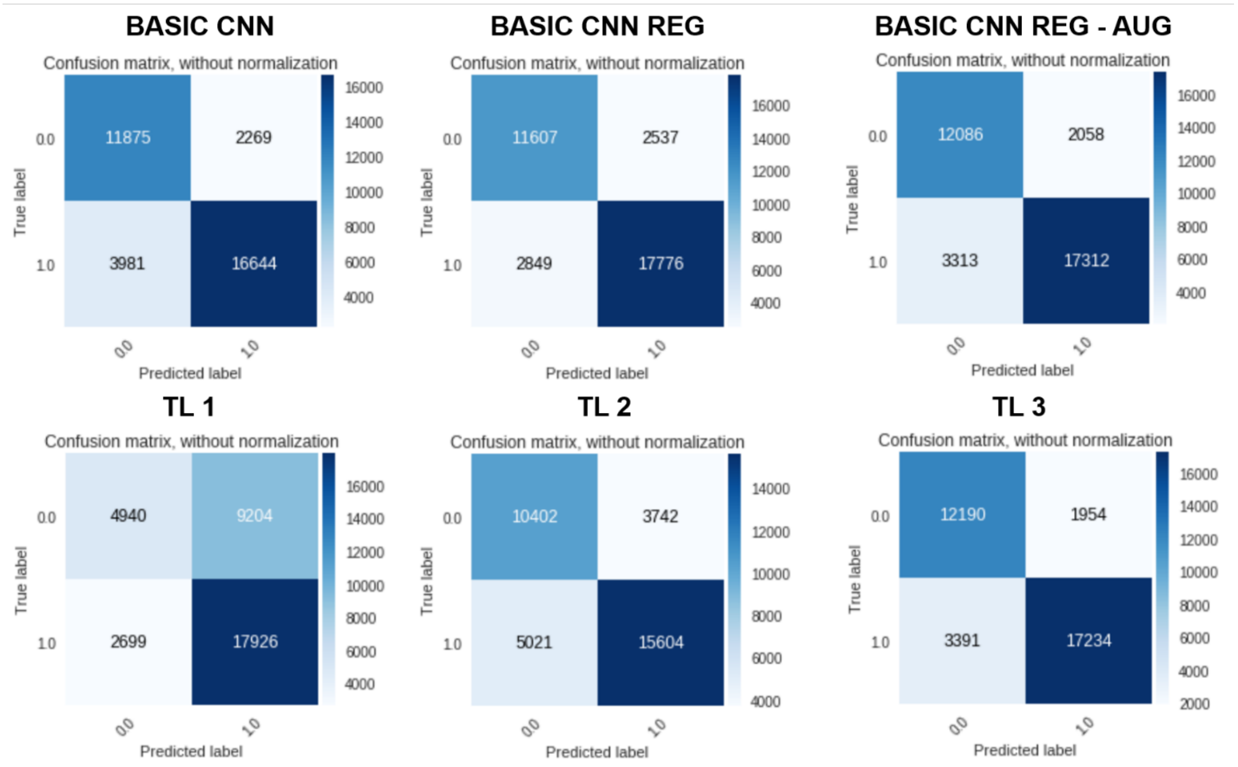


Figure 18: Confusion matrices extracted from the different models predicting the test data

4. Training a model for predicting the final test

All the tests presented in the preceding sections were done using the Colaboratory Jupyter Notebook environment of Google, using GPU. In order to predict the labels on the test set provided during the second phase, I switched to TPU and changed the code accordingly. As explained above, I decided to use model B3 for this purpose because of its good accuracy and its computational efficiency in terms of number of operations and weights to compute.

4.1. Results

Using all the samples from the synthesised training set described in section 2.1, I finally trained the model using batches of 124 samples for 100 epochs (each epoch taking approximately 100 s) and also reduced the learning rate by a factor of 10 when there was no improvement seen for 12 epochs.

In addition, using the hashing technique presented in section 2.1, I also identified some common images between the original training set and the final test set. More specifically, 213 images were identified as having the same hash. Some examples are shown in Figure 19. It is interesting to notice that the images appear visually the same but their difference is actually relatively important. For the identified duplicate images, I decided to replace the model predictions by the labels from the training set. With this approach, the best score I achieved on the leader board of the challenge was **0.858976.**

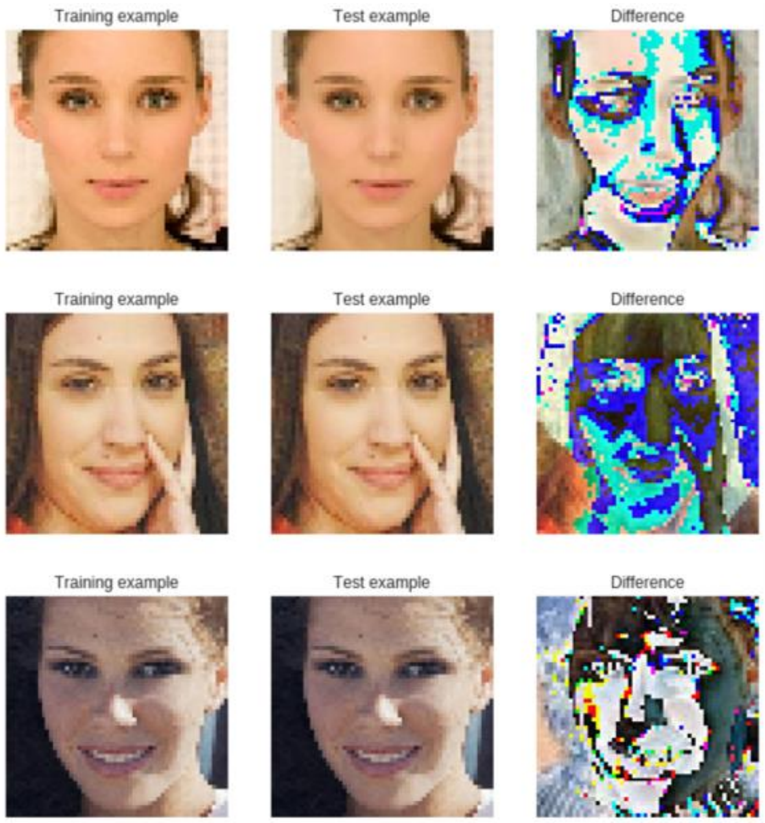


Figure 19: Common images between train and test sets

4.2. Possible improvements

Several approaches could be considered for further improving my final result with model B3. They can be summarized the following way:

- Considering using other type of initializations such as He et al. [6]. On my tests, I used the default initialization (‘glorot_uniform’ also known as Xavier initialization [7]).
- Combining the outputs from the fully connected layers with ensemble classifiers such as Random Forest or others. If more time would have been available, I would have probably tested some auto ML approaches as well (using for example the python tool TPOT [8]) to help me finding good models faster.
- Fine tuning more in depth the parameters related to the CNN chosen architecture. CNNs are well known to be very sensitive to the setting of their hyper-parameters. In this work, only few of them were tuned (mainly learning rate and epoch number). Other hyperparameters such as the batch size, the input size, the dropout rate, the pooling size would have required extra tuning. Some automatic fine-tuning methods such as random search or Bayesian optimization would be preferred to grid search. However, the less difficult and time-consuming approach still remains the hand-tuning.
- Another parameter that could have improved the score on the test predictions is the threshold used for converting the soft predictions into the integer classes 0 and 1. In all the models, I used a threshold of 0.5, but this may have required some adjustment for the model I used.

5. Hidden property

The main goal of the challenge was to set up a binary classification system for face property detection. However, the information about the property of images belonging to class 1 remains a mystery. By looking at the different images from the training set it was not obvious at all what could be the differences between classes 0 and 1.

One thing I noticed though was that several pictures from class 1 presented some female characteristics and the ones belonging to class 0 had some male characteristics. This appeared more obviously by looking at the average images of classes 0 and 1 (Figure 20). However, there are still several male faces in class 1 and several female faces in class 0. Consequently, the challenge was not about gender classification.

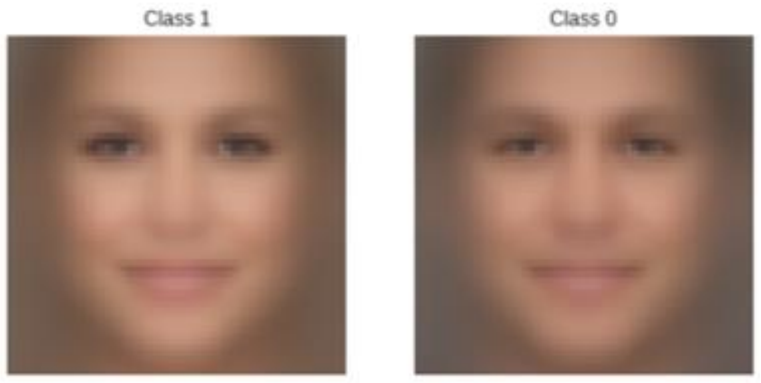


Figure 20: Average images of classes 1 and 0

In order to examine more precisely which are the regions from the image that contribute most towards a change in the output of the CNN, I computed saliency maps [9]. The main idea behind this concept is to calculate the gradient of the output category with respect to the input image. The results are shown in Figure 21. It appears that the pixels from the corner of the images are typically the ones that get activated after using the CNN architecture B3. The ones from the top right corner, where you may have some hair are often activated when we have class 1 images. Perhaps the hidden feature has to do with a combination of female or male face features coupled with some hair characteristics.

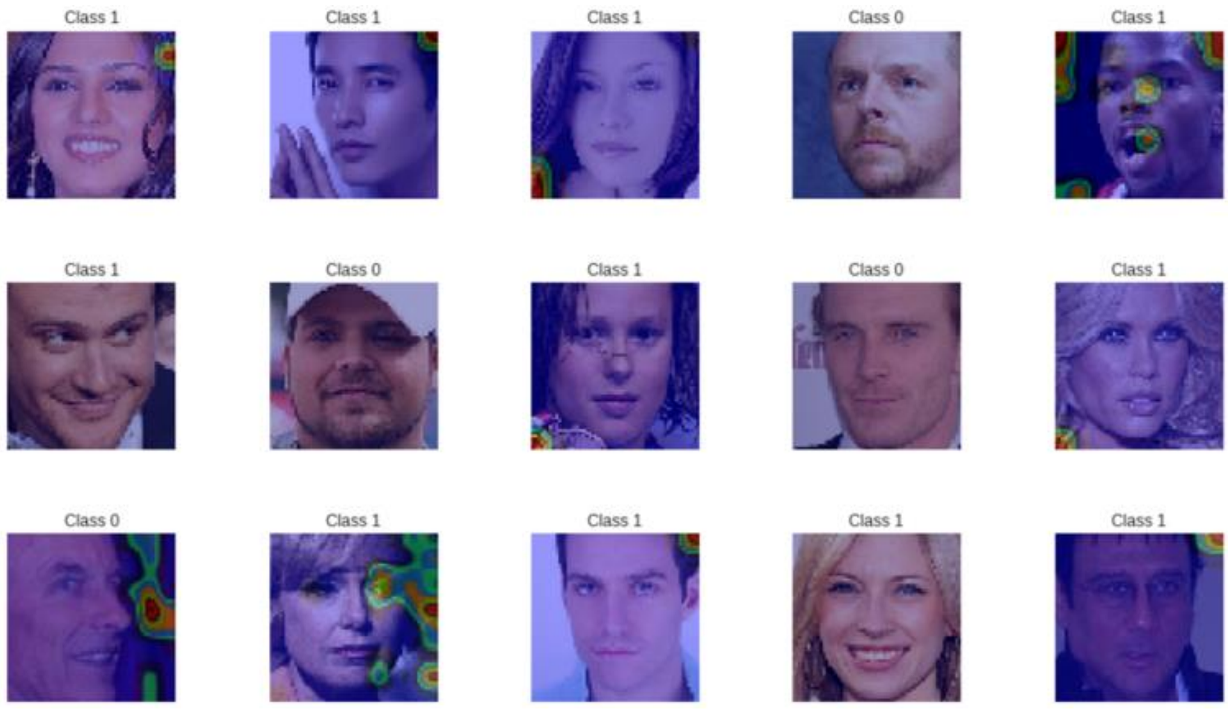


Figure 21: Saliency maps obtained for few images in the training set

6. Conclusion

The image classification problem considered in this challenge was not an easy one. In the end, the first person of the leader board achieved 0.87 in accuracy. The problem was even more difficult to address without knowing which property was behind all the images. The analysis of the different examples did not allow to conclude on which type of hidden feature I was trying to classify for.

As the challenge was related to computer vision, it seemed natural to use CNNs, that is why several architectures were tested. In the end, with a relatively simple one, an acceptable accuracy was obtained (**0.858976**) without needing to compute several millions of parameters as it was the case when using transfer learning approaches (freezing some convolutional layers or training from scratch).

References

[1] https://towardsdatascience.com/a-comprehensive-hands-on-guide-to-transfer-learning-with-real-world-applications-in-deep-learning-212bf3b2f27a

[2] https://blog.iconfinder.com/detecting-duplicate-images-using-python-cb240b05a3b6

[3] He, K., Zhang, X., Rem, S., and Sun, J. (2016). Deep residual learning for image recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016, pages 770-778.

[4] Canziani, A., Paszke, A., and Culurciello, E. (2016). An analysis of deep neural network models for practical applications. CoRR, abs/1605.07678.

[5] Schroff, F., Kalenichenko, D. and Philbin, J. (2015). Facenet: A unified embedding for face recognition and clustering. CoRR, abs/1503.03832.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In Proceedings of the IEEE international conference on computer vision, pages 1026–1034, 2015.

[7] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In AISTATS, volume 9 of JMLR Proceedings, pages 249-256. JMLR.org.

[8] https://epistasislab.github.io/tpot/

[9] Simonyan, K., Vevaldi, A., and Zisserman, A. (2013). Deep inside convolutional networks: Visualizing image classification models and saliency maps. CoRR, abs/1312.6034.