Project 2

Labeling and Classification on CelebA

Deep Learning Course (A. Y. 2019-2020)

University of Bologna

Report by
Lorenzo Visentini
Gianmarco Bolcato

Professor
Andrea Asperti

Agenda:

Introduction

During the last few years Facial Attribute Classification (FAC) became one of the most interesting fields in deep learning researchers. It involves face recognition for recommendation systems and other possible implementation. In other words, given a facial image, the task of FAC is to predict facial attributes.

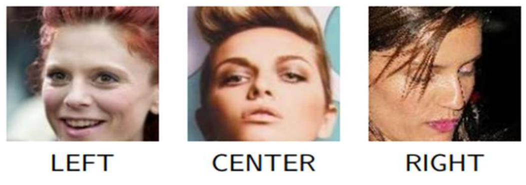
Our project goal is to create a classifier using Neural Networks to predict the face orientation and the face light source origin of images from a very known dataset in the Deep Learning field, called CelebA Dataset.

The **CelebFaces Attributes Dataset (CelebA)** is a large-scale face attributes dataset with more than 200'000 celebrity images, each with 40 attribute annotations. This dataset is great for training and testing models for face detection, particularly for recognizing facial attributes such as finding people with brown hair, smiling, or wearing glasses. Images cover large pose variations, background clutter, diverse people, supported by a large quantity of images and rich annotations. This data was originally collected by researchers at MMLAB, The Chinese University of Hong Kong.

Sample images

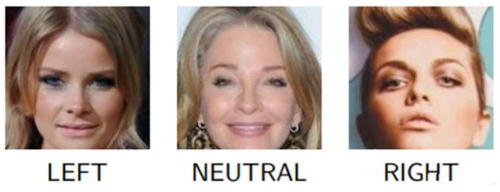
Face Orientation

The first request is to code a classifier which aims to classify images taken as input based on the orientation of the faces (i.e. left, center, right).



Source of light

The second request is related to build a classifier which recognizes where the source of light is in the photo (i.e. left, center, right).



Hardware

The hardware we used for this project was:

Lorenzo base hardware

Laptop Acer Aspire …

CPU Intel I7-7700U

GPU Intel

RAM 8 GB

SSD Samsung Evo 1 TB

Gianmarco base hardware

Laptop Asus VivoBook

CPU Intel I5-8250U

GPU Intel

RAM 8 GB

SSD

Colab hardware

Google Colab or “the Colaboratory” is a free cloud service hosted by Google to encourage Machine Learning and Artificial Intelligence research.

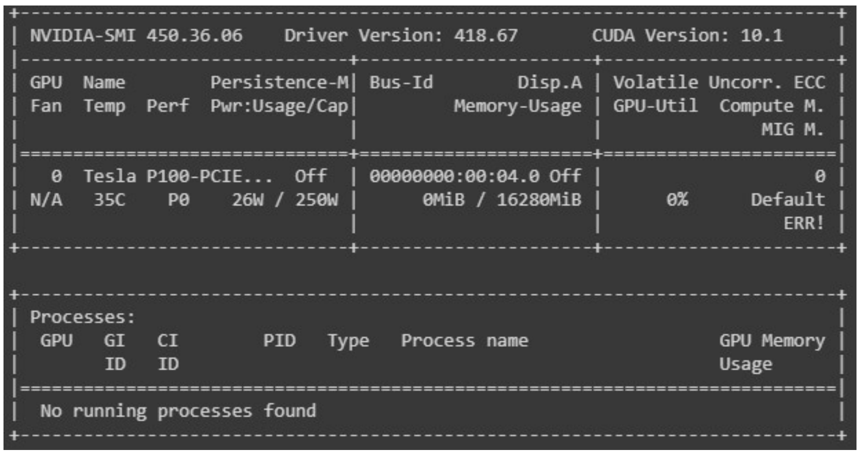
In particular, due to the low computational power of our base hardwares we decided to use Google Colab Jupyter Notebook, which notebooks allow you to combine executable code in Python and rich text in a single document.

The strength of this platform is to create your code as a notebook, with cells that can be run once a time. At the same time, Colab offers specific settings including the possibility to run the code on a gpu hosted runtime

Here you can see the configuration of the CPU provided by Google colab during our project:

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              2
On-line CPU(s) list: 0,1
Thread(s) per core:  2
Core(s) per socket:  1
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               63
Model name:          Intel(R) Xeon(R) CPU @ 2.30GHz
Stepping:            0
CPU MHz:             2300.000
BogoMIPS:            4600.00
Hypervisor vendor:   KVM
Virtualization type: full
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            46080K
NUMA node0 CPU(s):   0,1
Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc rep_good
nopl xtopology nonstop_tsc cpuid tsc_known_freq pni pclmulqdq ssse3 fma cx16 pcid sse4_1
sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm invpcid_single
ssbd ibrs ibpb stibp fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid xsaveopt arat
md_clear arch_capabilities
```

And here the GPU specifications:



```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.36.06    Driver Version: 418.67       CUDA Version: 10.1     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   35C    P0    26W / 250W |      0MiB / 16280MiB |      0%      Default |
|                               |                      |                 ERR! |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

During the last few days, we had the opportunity to quickly test only the Transfer Learning models on an extern machine with a hardware upgraded.

Hardware

CPU Intel I7 7700

GPU RTX 2060

RAM 16 GB

SSD 256 GB

Software

Python

Programming language at a high level, very used in the Artificial Intelligence (Machine Learning and Deep Learning) field, thanks to the import of specific libraries.

In our project, we used the *Keras* library API for everything related to models and KPI.

Google Colab

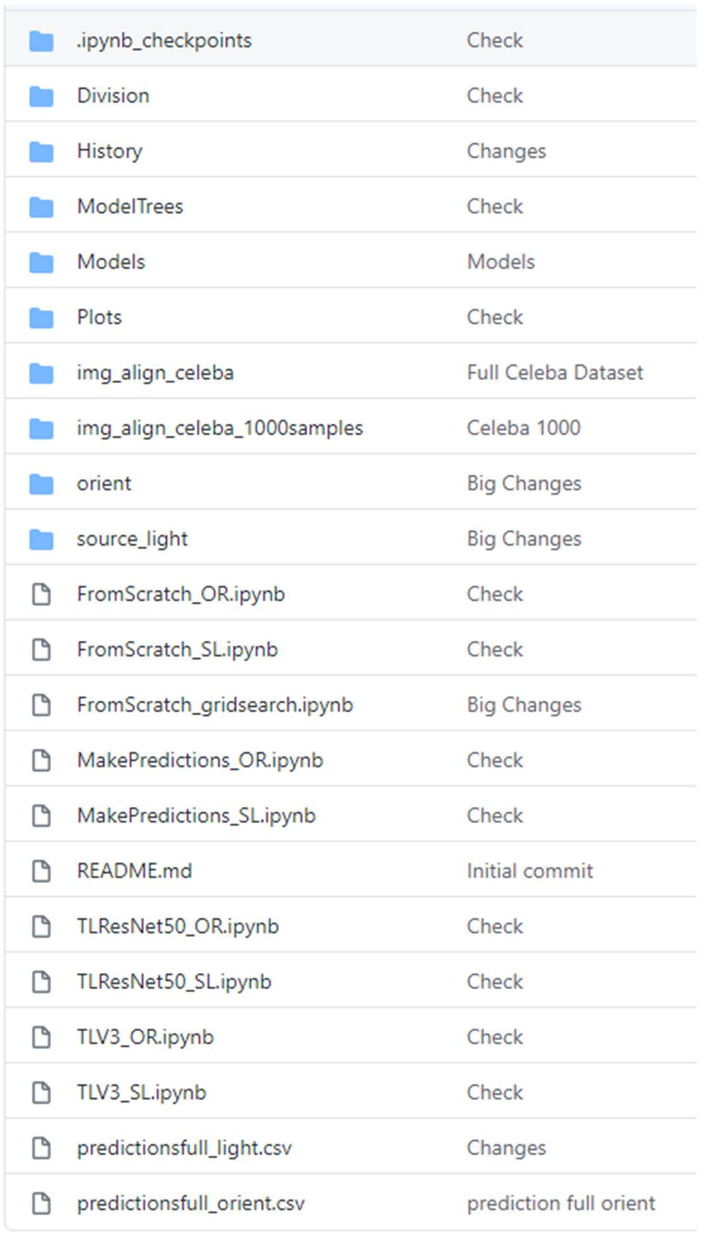
Describe early in this paper. Employed, as said before, in order to overcome all the hardware difficulties.

GitHub and GitHub Desktop

GitHub is a hosting service for software projects and a version control using Git. It offers the possibility to work on repository that can be modified by all the members of the group with authorization.

We used GitHub Desktop as the desktop application of GitHub. The advantage is an easy and complete interface, provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.

Our GitHub project's repository:



.ipynb_checkpoints	Check
Division	Check
History	Changes
ModelTrees	Check
Models	Models
Plots	Check
img_align_celeba	Full Celeba Dataset
img_align_celeba_1000samples	Celeba 1000
orient	Big Changes
source_light	Big Changes
FromScratch_OR.ipynb	Check
FromScratch_SL.ipynb	Check
FromScratch_gridsearch.ipynb	Big Changes
MakePredictions_OR.ipynb	Check
MakePredictions_SL.ipynb	Check
README.md	Initial commit
TLResNet50_OR.ipynb	Check
TLResNet50_SL.ipynb	Check
TLV3_OR.ipynb	Check
TLV3_SL.ipynb	Check
predictionsfull_light.csv	Changes
predictionsfull_orient.csv	prediction full orient

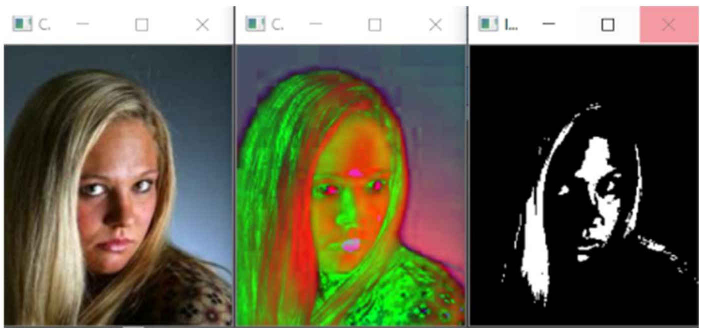
The link to the repo is: https://github.com/Lor3nzoVis3/DeepLearning_ProjectLV_GB

Dataset setup

Firstly, before adopting the strategy “try, observe and refine”, we spent time building up the whole structure.
The approach is the same for both the problems. Pipeline was to start classifying manually the Celeba Dataset, splitting images by the labels. Then, focus on the transfer learning models, identify two named CNNs and fine-tune them. With the same approach, build the From Scratch CNN studying possible configurations and fine-tune the one chose. Finally, test the best model – identified by best accuracy and best loss without overfitting. At the end critically analyse the results, evaluating them and making a final comparison.

First, it is necessary to set up the datasets. We decided to split 10’000 images for each task, face orientation and source light, sharing the activity with the other group. The datasets preparation has been a very time-consuming activity, taking a lot of time even though the implementation of some automation helped the manual division - specifically, a python code to insert the files in distinct folders.
Strictly connected to the requests of the project, the folders were 3 for both the problems, splitting the images in center, left and right.
The light, especially in images with medium-low resolution, is not easy to detect as the reflection can create issues about where the source is coming from. Adding to this, lights sometimes provoked misunderstanding due to noise of the background and artificial setup made by photographers.
The importance of the input of a convolutional neural network is very high and consequently it’s indispensable to clean data as well as perform very accurate manual classification. And since detecting the source light is not exclusively divided – light can come from different positions and, as said before, reflections can cause some issues – we adopted some rules at this stage.
Firstly, the classes decided are only three, avoiding mixed classes such as center-left, center-right. The division is led by the portion of the image – or portion of the face - that has mostly of the light.
Secondly, some transformation to Celeba images has been performed: HSV (Hue, Saturation, Value) color model which describes colors in terms of their shade, pixel-value based threshold pointing out only pixels’ values that are higher than 170 – founded by trying – and traditional image.

Example:



This solution has enhanced the clarity of the origin of the light.
Hence, we applied some operations similar to a pre-processing stage.
During the manual classification, many pictures have been discarded since the categorization was nearly impossible, despite the efforts.

Train - validation - test

Once we got 10’194 images splitted in 3 folders - right, center, left - for face orientation and 10’000 images splitted in 3 folders - right, center, left - for light source, the focus has moved on setting up the folders preparatory for the models.
An initial random shuffle has been made in order to avoid any bias during the division.
The proportions for the training, validation and test set were 60, 25 and 15%. After a few compiles and runs of the code, we had to change the proportions of the directories as some bias has been shown up, in particular due to the low portion of the train compared to the percentage of the validation – the validation accuracy was too often higher than the training accuracy.
Final proportions opted are 70% train, 15% validation and 15% testing.

At the end we got 10194 for face orientation splitted in:
7135 images belonging to 3 classes in the training set
1529 images belonging to 3 classes in the validation set
1530 images belonging to 3 classes in the test set

And 10000 images for source light splitted in:
6999 images belonging to 3 classes in the training set
1500 images belonging to 3 classes in the validation set
1501 images belonging to 3 classes in the test set.

Transfer Learning

Transfer learning is a research problem in machine learning that focuses on storing knowledge gained while solving one problem and applying it to a different but related problem. It consists of taking features learnt on one, first, problem and applying them on a new second, but related, problem.
In the Convolutional Neural Networks field, an existent neural network, pre-trained on a similar problem, is transposed to the main problem of interest.
The benefits of the transfer learning approach are many and often related to the request of the project. Usually, the starting datasets are not large, not big enough to achieve important results in terms of accuracy and loss. The use of data augmentation can help but at the same time can also introduce bias that get the things worse – in this paper we will talk about data augmentation. In addition to this, the risk of overfitting is very high due to the implementation of a complex neural network, with millions of parameters.
Hence, the Transfer Learning comes in handy allowing the use of pre-trained neural networks. This approach overcomes the issue of the expensiveness of the training since it takes a lot of time and requires dedicated hardware.
On the other hand, transfer learning is not always the best solution. It is necessary to study in depth the project and decide if this is similar to the problem solved by state-of-art CNNs.
For the project we opted for two very known CNNs, GoogLeNet InceptionV3 and ResNet50.
The reason why the choice of these two state-of-art CNNs relies on that both have introduced particular modules or news.
For both the convolutional neural networks, the transfer learning approach used is the same. Leveraging on the strength, pre-trained layers are freezed adopting the respective weights, while only the top layers are trained on the new dataset. This creates lots of advantages using weights of InceptionV3 and ResNet50 on low-mid layers - where the features learned are general and local (such as colors, patterns, etc.) - and train final layers' weights where the features are specific.

Transfer Learning issue

In the early stage of fine-tuning, the results of the Transfer Learning related to the Inception V3 and ResNet50 models were very poor. Both accuracy and validation accuracy flattened to values close to 50% and loss and validation loss a bit high too.
The issue is known in the deep learning field, in particular in blogs of Keras users where many face the problem of applying transfer learning to some of state-of-art CNNs.
The problem itself is strictly related to how Keras had implemented the Batch Normalization layer and how the layer computes the regularization. Batch normalization is a form of regularization where

the input, in batches, is normalised in order to have the mean around 0 and the standard deviation close to 0.
But, in Keras the implementation is a bit particular since the computation is different if the model is in training mode or inference mode (during predict).

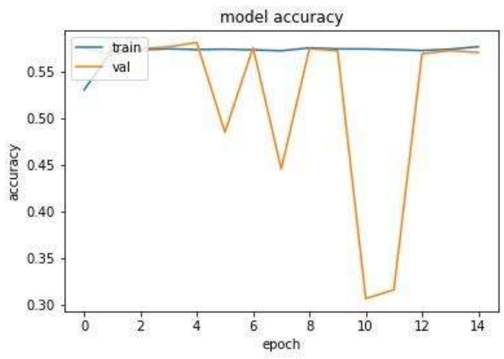
Patch

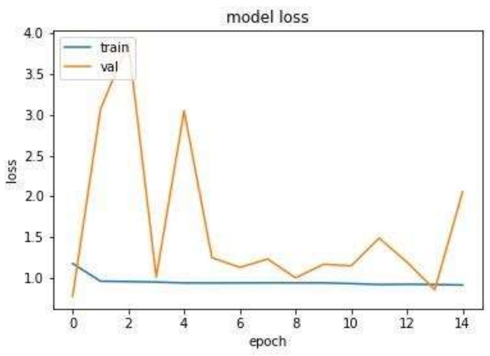
The problem with the implementation of Keras is that, during transfer learning and when a BN layer is frozen, it still updates the mini-batch statistics during training. This issue is due to a problem related to Keras development.
During training your network will always use the mini-batch statistics either the BN layer is frozen or not; also, during inference you will use the previously learned statistics of the frozen BN layers. As a result, if you fine-tune the top layers, their weights will be adjusted to the mean/variance of the new dataset. Nevertheless, during inference they will receive data which are scaled differently because the mean/variance of the original dataset will be used.
So even if BN is frozen it updates the mini-batch statistic, Unfortunately, by doing so you get no guarantees that the mean and variance of your new dataset inside the BN layers will be similar to the ones of the original dataset.
We found that a better approach is to use the moving mean and variance that it learned during training when the BN is frozen. A solution would be to reset the moving mean and the variance of the BN layer in order to get close to the variance and mean of the new dataset. We know that is not the perfect solution but at least if fits for us and the results that we obtained with the patch are very interesting. We could say that it is a local maximum.

The batch normalization issue is solved from Tensorflow 2.X but using specific syntax in the code (we used the syntax of the 1.X).

It is necessary to point out the differences in results between the model with the “patch” applied and the transfer learning model trained without consider the Batch Normalization issue.
The comparison is made between the same best inception model (orient) with and without the patch code.

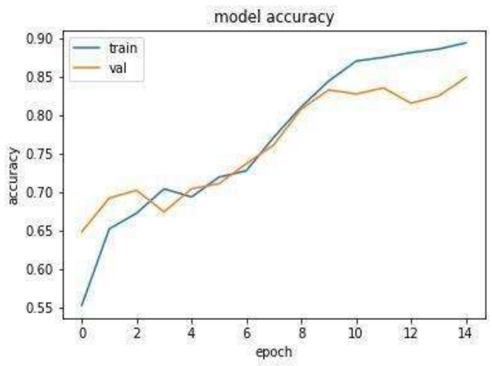
Without Patch

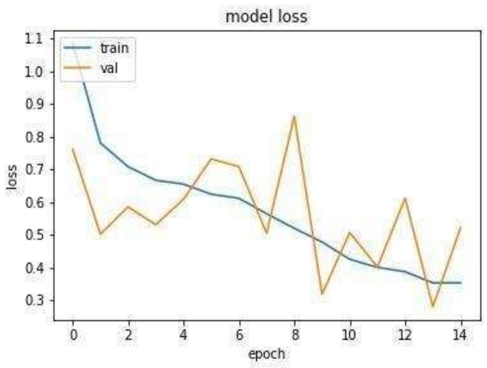




The values reached of the two metrics are very unsatisfactory.
The training accuracy, in 15 epochs, does not overcome the 60%, precisely 0.5774. The validation accuracy seems very inaccurate ending with a value of 0.5711.
Observing the loss plot, the training loss is never less than 0.9 (0.9175) while the validation loss ends up with 2.0593.

With Patch





The large differences in results obtained between the two are really evident. This has brought adopting the patch code for all the transfer learning models while is not reported in the From Scratch CNN – remember that the main cause is the Batch Normalization behaviour during transfer learning application.

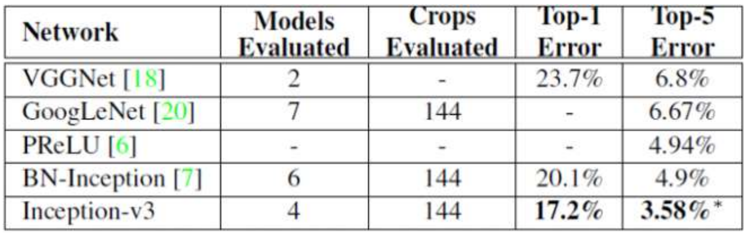
Data Augmentation

Data augmentation is a tool that can be used to enhance the robustness of the mode, of a particular Convolutional Neural Network.
In particular, it refers to all the possible transformations and processes that are done on the input in order to create a bit of noise or train the model to be more reliable even with different images as input. In Keras, Data augmentation can be done by calling the ImageDataGenerator, a function that allows different transformations to the batch in input during the learning phase. It is important to highlight that this process happens on fly, which means that it is not a real augmentation of the input dataset but the pre-processing involves the input batches coming with the 'flow_from_directory' function. The number of images is the same, despite its undergoing transformations.
The transformations can be many, starting from rotation – flipping -, translation along horizontal axes, translation along vertical axes, zoom and more. In translations or zoom out pixels, that initially do not exist, the filling is made typically by taking the nearest pixels.
Data augmentation can be a powerful tool but can also introduce bias. It is important to learn empirically if the utilisation improves the performance or, in contrary, decreases metrics values.

In the project, data augmentation will be applied to all the different models underlined and compare the results with models without data augmentation.

InceptionV3

With 42 layers, a lower error rate is obtained so it becomes the 1st Runner Up for image classification in ILSVRC (ImageNet Large Scale Visual Recognition Competition) 2015.



Network	Models Evaluated	Crops Evaluated	Top-1 Error	Top-5 Error
VGGNet [18]	2	-	23.7%	6.8%
GoogLeNet [20]	7	144	-	6.67%
PReLU [6]	-	-	-	4.94%
BN-Inception [7]	6	144	20.1%	4.9%
Inception-v3	4	144	**17.2%**	**3.58%***

The pick of the InceptionV3 is due to the introduction of an innovation after the VGG-16 and VGG-19 architectures.

The considerations that brought new research studies from Google were many. It had been discovered that the depth of the neural network was the first and the most important feature which allows to reach high results. At the same time, this has some drawbacks: firstly, a large CNN can be prone to overfitting as the number of parameters increase - VGG-16 138 million and VGG-19 144 million of parameters. Then, the computational effort increases a lot consequently.

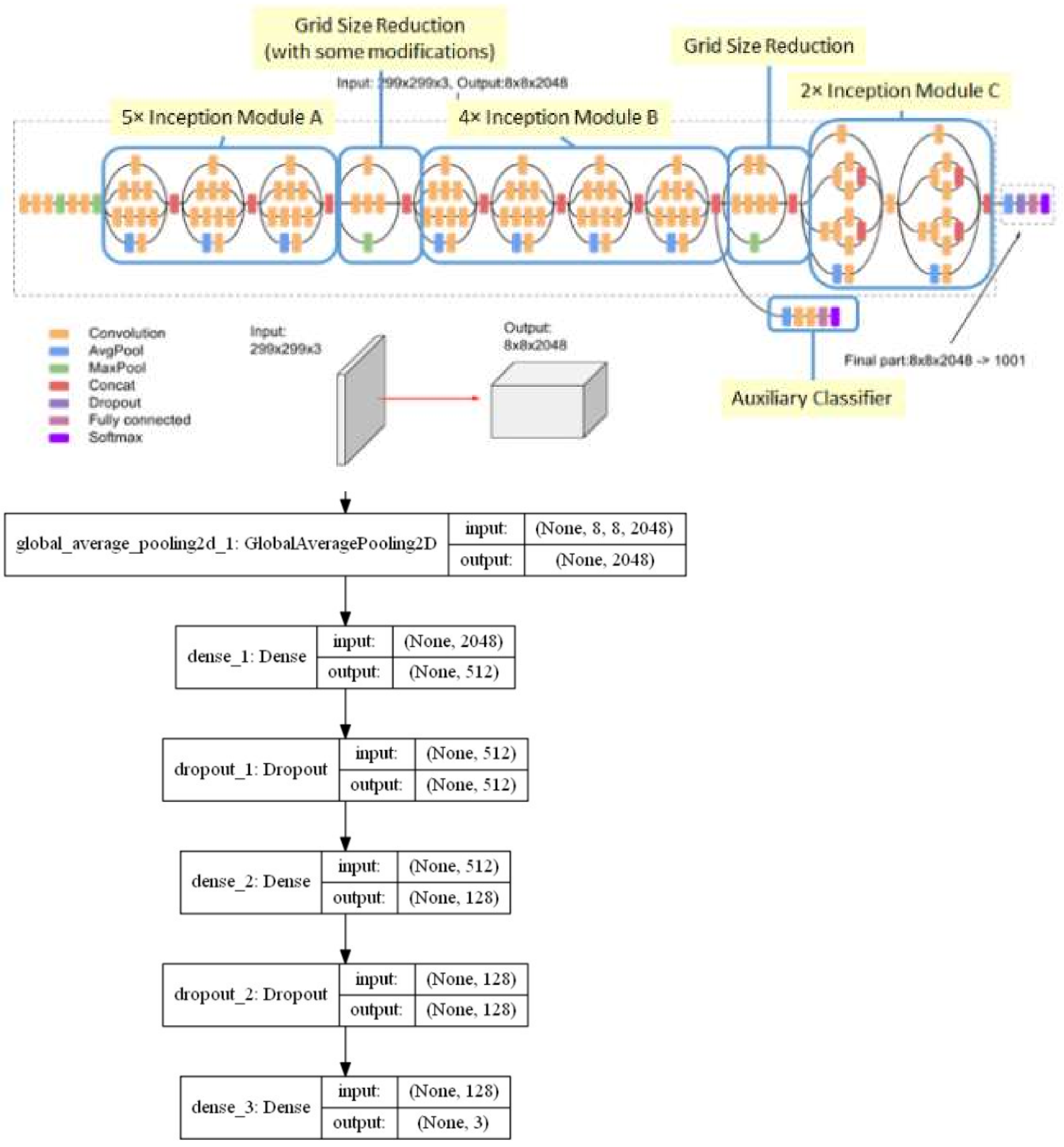
Because of the computational budget that researchers and companies had (and have), this power calculation has to be done.

Hence, Google introduced in Inception v1 paper, inception modules in order to decrease in an efficient way the number of parameters without losing information. Thinking about width instead of only depth.

Inception v2 and v3 versions and papers brought factorization as well as batch normalization in the auxiliary classifiers (previously used for reducing vanishing gradient problem, now also for regularization).

So, three different modules had been created: inception module A, inception module B and C, differentiating for various types of factorization and promoting high dimensional representations.

Architecture



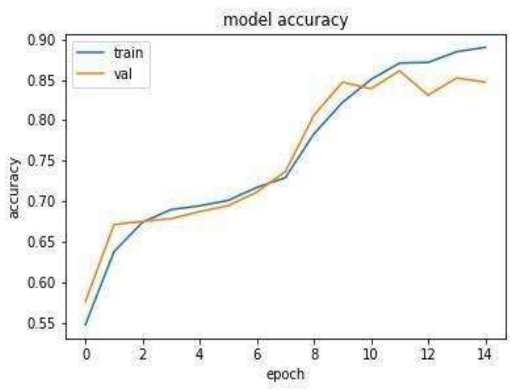
global_average_pooling2d_1: GlobalAveragePooling2D	input:	(None, 8, 8, 2048)
	output:	(None, 2048)
dense_1: Dense	input:	(None, 2048)
	output:	(None, 512)
dropout_1: Dropout	input:	(None, 512)
	output:	(None, 512)
dense_2: Dense	input:	(None, 512)
	output:	(None, 128)
dropout_2: Dropout	input:	(None, 128)
	output:	(None, 128)
dense_3: Dense	input:	(None, 128)
	output:	(None, 3)

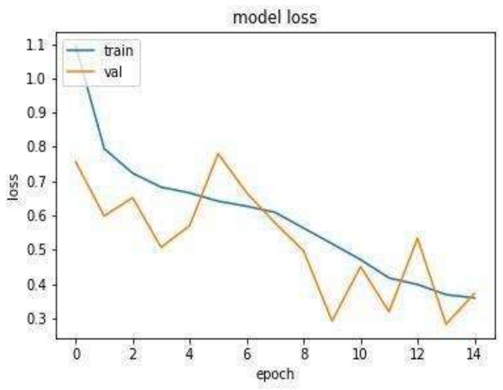
The top layers inserted follow the logic of flattening the last pre-trained layer with a Global Pooling layer. This configuration substitutes the Pooling layer and after the Flatten layer. The motivation below the choice of the Global Average Pooling instead of the Global Max Pooling is due to the willingness of taking the average of features instead of picking the extreme values.
Then, two blocks of Dense layer – with 512 neurons and 128 neurons – followed with a Dropout layer.
Last, the Dense layer with the number of neurons equal to the number of class.

Orient Results:

The best model is the Inception V3, with 32 of batch size, 16 is the depth of the convolutional 2D layers, the value of the drop rate 0.55. The optimizer choose is the AdaBound, a new optimizer that seems – empirically – to achieve the results of the SGD (Stochastic Gradient Descent) with almost the speed of the Adam optimizer. The learning rate adopted is 0.003 in order to arrive with more precision to the minimum.

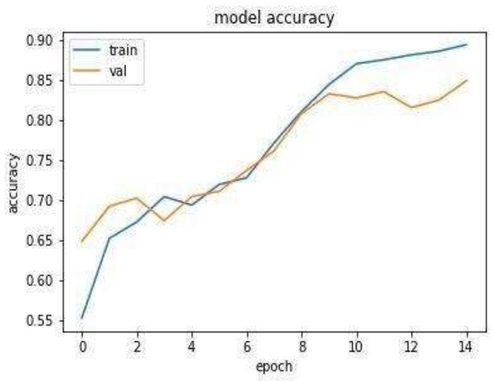
Without Data Augmentation

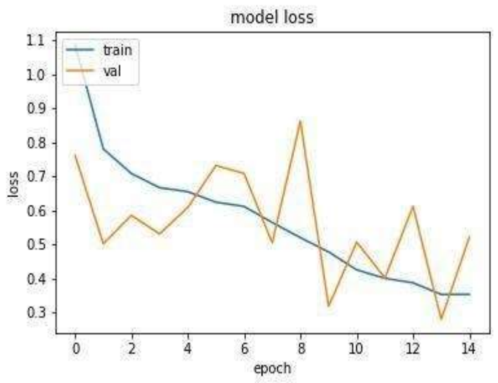




The model training accuracy is 0.8893 while the validation accuracy is 0.8472.
The model training loss is 0.3715 while the validation loss is 0.3824.
The results are good even though in the last epochs the model tends slowly to overfit.

With Data Augmentation – <u>Best model</u>





Results between the model with data augmentation and the model without it are really close. Despite the nearly invisible differences, the first one has been chosen as the best due to the more robustness – lower error during the inference stage. This is probably due to the use of a pre-processing stage.

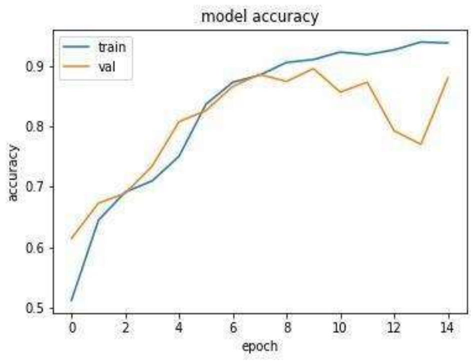
Results:

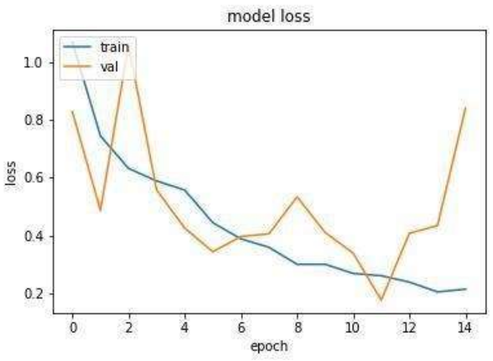
InceptionV3 Orient	Accuracy	Loss
Train	0.8943	0.3529
Validation	0.8490	0.5211
Test	0.8604	0.3001

Source Light results:

The best model is the Inception V3, with 32 of batch size, 16 is the depth of the convolutional 2D layers, the value of the drop rate 0.4. The Dropout percentage is a bit lower than the value used on the orient dataset to intentionally allow the CNN to learn more from the input images.
The optimizer choose is the AdaBound with a learning rate adopted equal to 0.005, the best compromise employed to give more importance to the previous steps and equilibrate a lower dropout rate.

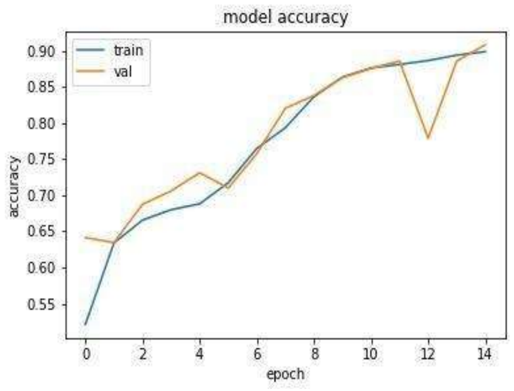
Without augmentation

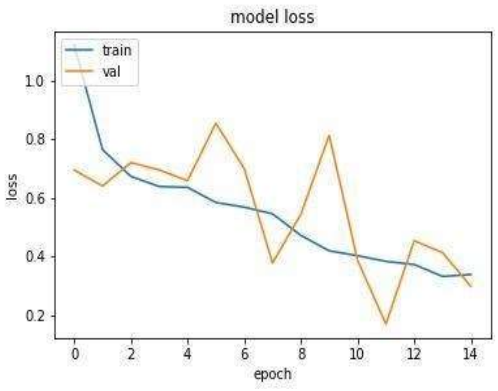




The model training accuracy is 0.9324 while the validation accuracy is 0.8885.
The model training loss is 0.2215 while the validation loss is 0.8375.
The results are really interesting even though in the last epochs the model appeared to be more unstable, with a very high value of validation loss in the 15th epoch.

With Data Augmentation – <u>Best model</u>





Results:

InceptionV3 Light	Accuracy	Loss
Train	0.8985	0.3382
Validation	0.9080	0.2984
Test	0.9034	0.3977

ResNet50

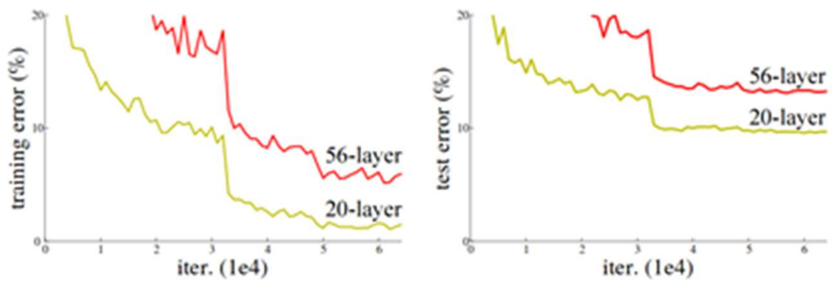
ResNet50 is a variant of the ResNet model which has 48 Convolution layers along with 1 MaxPool and 1 Average Pool layer.

Below the results achieved in image classification in ILSVRC (ImageNet Large Scale Visual Recognition Competition) in 2015.

method	top-1 err.	top-5 err.
VGG [40] (ILSVRC'14)	-	8.43†
GoogLeNet [43] (ILSVRC'14)	-	7.89
VGG [40] (v5)	24.4	7.1
PReLU-net [12]	21.59	5.71
BN-inception [16]	21.99	5.81
ResNet-34 B	21.84	5.71
ResNet-34 C	21.53	5.60
ResNet-50	20.74	5.25
ResNet-101	19.87	4.60

In a Convolutional Neural Network (CNN), as the number of layers increase, so does the ability of the model to fit more complex functions. Therefore, a greater number of layers is always better, but deeper neural networks are more difficult to train.

ResNet is born to address the degradation problem in a deep network. Adding more layers to a sufficiently deep neural network would first see saturation in accuracy and then the accuracy degrades, as we can see from the following picture.

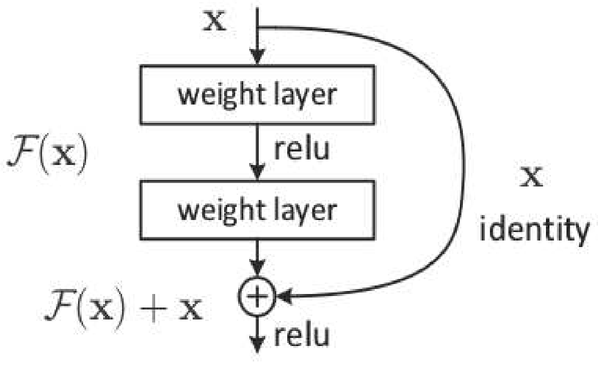


This problem is know as Vanishing gradient problem where going further with the learning, the gradient during backpropagation approach to zero, making the network very hard to train – since the weights update too slowly.

The weights of a neural network are updated using the backpropagation algorithm. The backpropagation algorithm makes a small change to each weight in such a way that the loss of the model decreases. It updates each weight such that it takes a step in the direction along which the loss decreases. This direction is nothing but the gradient of this weight (with respect to the loss).

Using chain rule, we can find this gradient for each weight. It is equal to (local gradient) x (gradient flowing from ahead). Here comes the problem. As this gradient keeps flowing backward to the initial layers, this value keeps getting multiplied by each local gradient. Hence, the gradient becomes smaller and smaller, making the updates to the initial layers very small, increasing the training time considerably.

Thus, the residual block has been introduced (shown below).



Thinking about a sufficiently DNN that calculates a sufficiently strong set of features that is necessary for the task in hand (ex: Image classification). If we add one more layer of the network to this already very DNN, what will this additional layer do? If already the network could calculate strong features then this additional layer does need to calculate any extra features, rather just copy the already calculated features i.e. perform an identity mapping (kernels in the added layer produce exact same features to that of the previous kernel).

The idea of a residual block is completely based on the intuition that was explained before.

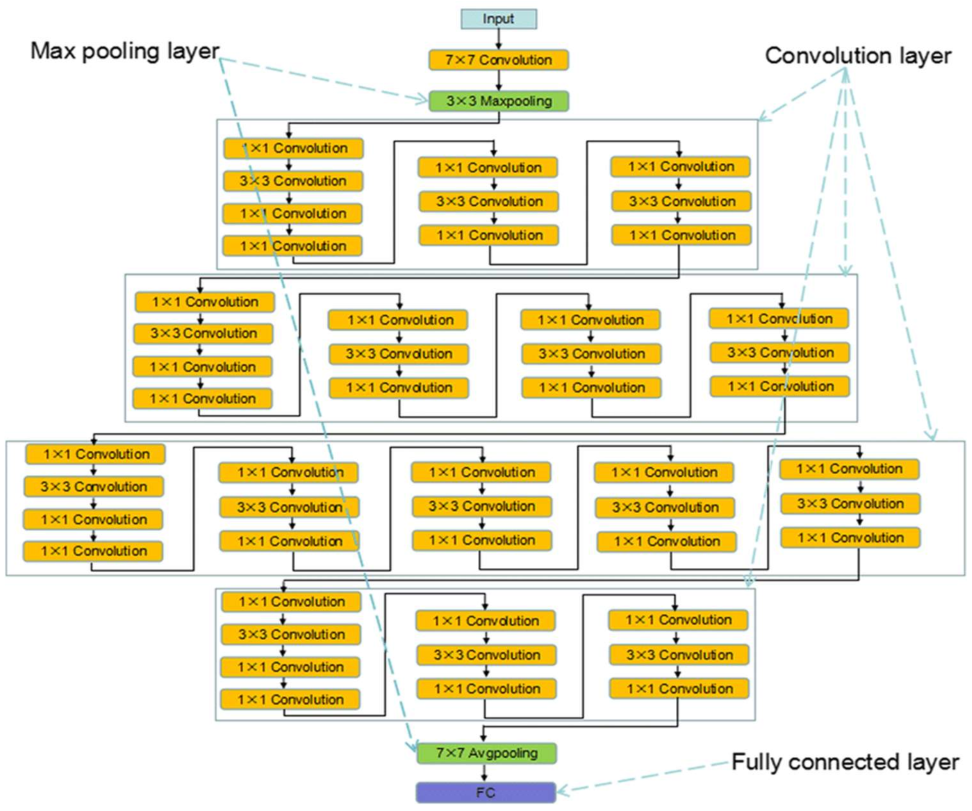
The residual connection directly adds the value at the beginning of the block, x, to the end of the block (F(x)+x). This residual connection does not go through activation functions that "squashes" the derivatives, resulting in a higher overall derivative of the block.

The Skip Connections between layers add the outputs from previous layers to the outputs of stacked layers. This results in the ability to train much deeper networks than what was previously possible.

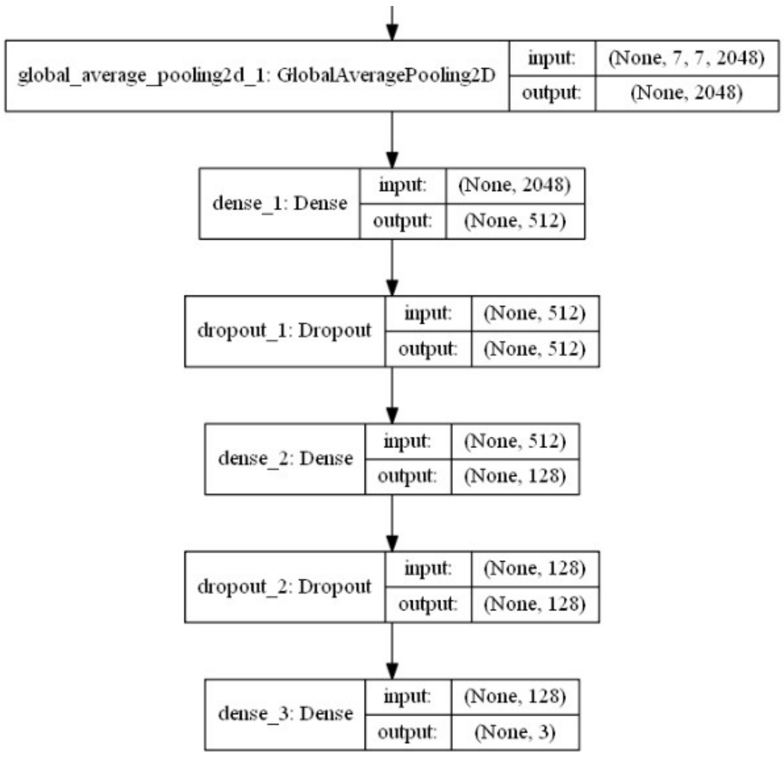
Instead of hoping each few stacked layers directly fit a desired underlying mapping, it explicitly let these layers fit a residual mapping – so, build the layer as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions.

The ResNet architecture, shown below, should now make perfect sense as to how it would not allow the vanishing gradient problem to occur. ResNet stands for Residual Network.

ResNet50 Architecture



From this architecture we add these blocks at the end of the network:

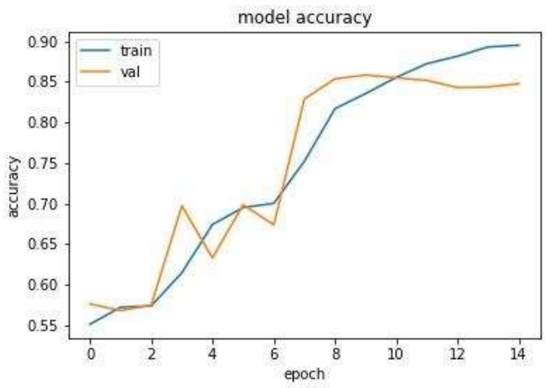


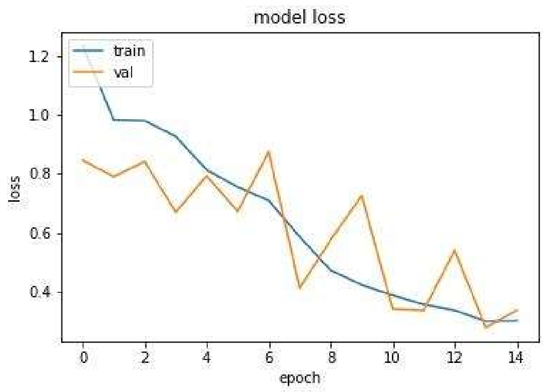
The top layers inserted follow the logic of flattening the last pre-trained layer with a Global Pooling layer. This configuration substitutes the Pooling layer and after the Flatten layer. The motivation below the choice of the Global Average Pooling instead of the Global Max Pooling is due to the willingness of taking the average of features instead of picking the extreme values.
Then, two blocks of Dense layer – with 512 neurons the first one and 128 the second – followed with a Dropout layer, the same top layers' configuration as the InceptionV3.
Last, the Dense layer with the number of neurons equal to the number of classes.

Orient results:

The best model for the face orientation task looking at ResNet50 transfer learning is obtained by setting the batch size to 32, 15 epochs, 0.6 of drop rate and 0.005 as learning rate of the AdaBound optimizer.

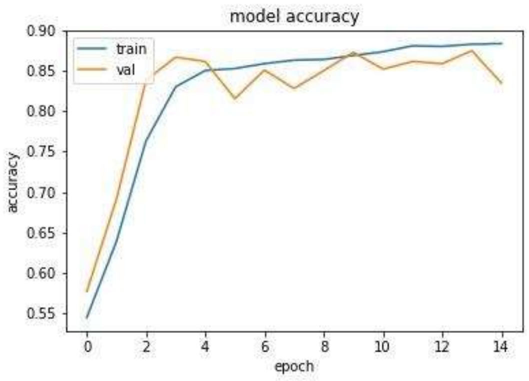
No Data Augmentation and with AdaBound

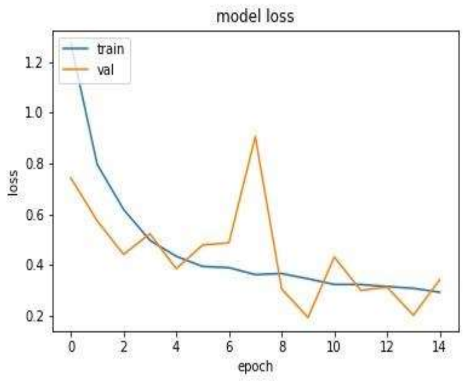




The model training accuracy is 0.8997 while the validation accuracy is 0.8504.
The model training loss is 0.2989 while the validation loss is 0.3211.
Without Data Augmentation results are good, even though there is a little decrease in validation accuracy after the 9 epoch – the 9th epoch sees the best validation accuracy. The losses are really both close to 0.3.

With Adam optimizer and Data Augmentation



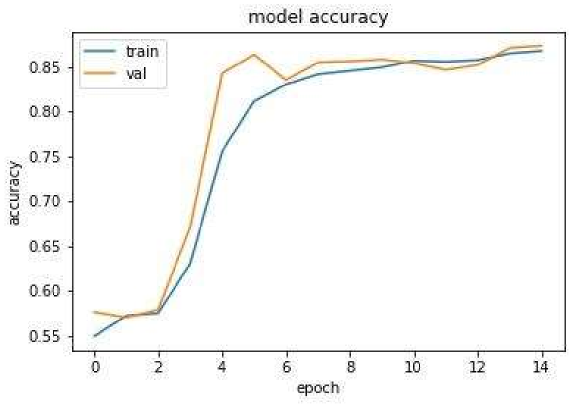


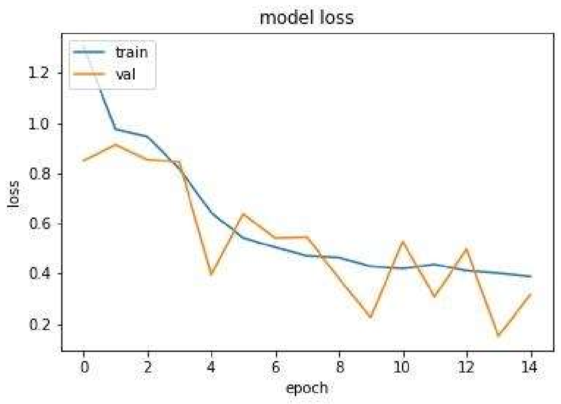
The model training accuracy is 0.8871 while the validation accuracy is 0.8382.

The model training loss is 0.3108 while the validation loss is 0.3656.

The validation accuracy drops after the 13th epoch, maybe a symptom of overfitting if the epochs would be increased. The Adam works well, fast achieving good values but with a great risk of overfitting when the epochs increase.

With Data Augmentation and with AdaBound – Best model





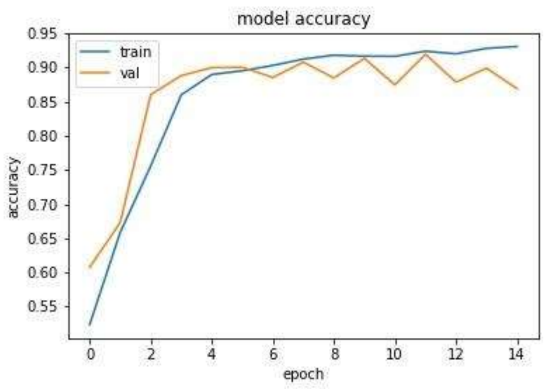
Resnet50 Orient	Accuracy	Loss
Train	0.8495	0.4006
Validation	0.8637	0.3220
Test	0.8656	0.3262

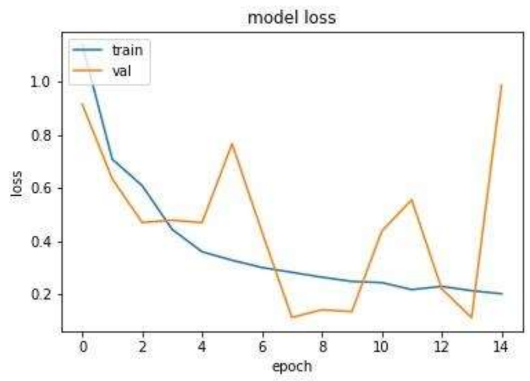
The results are satisfactory where the predict on test directory has brought high results. The validation and test accuracy are higher than the training one, this could be due to luck – there is a component of randomness. Regardless, the difference is small to think about biases even though could be a possibility.

Source light results:

The best model for the source light looking at ResNet50 transfer learning is obtained by setting the batch size to 32, 15 epochs, 0.6 of drop rate and 0.005 as learning rate of the AdaBound optimizer.

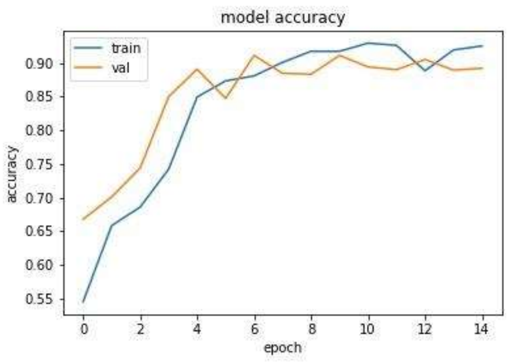
No Data Augmentation and with AdaBound

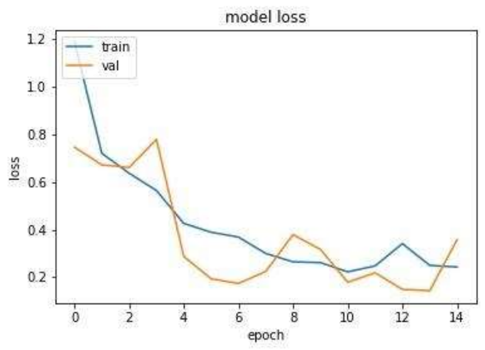




The model training accuracy is 0.9386 while the validation accuracy is 0.8673.
The model training loss is 0.2078 while the validation loss is 0.6327.
The accuracy is really high for training, while the validation accuracy drops and validation loss rockets on the 15th epoch synonyms of an increase in overfitting - it has to be kept in mind since also the loss is important.

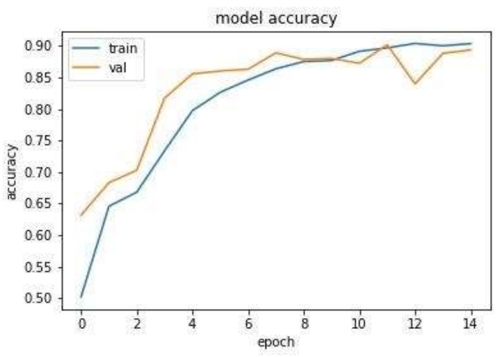
With Adam optimizer and Data Augmentation

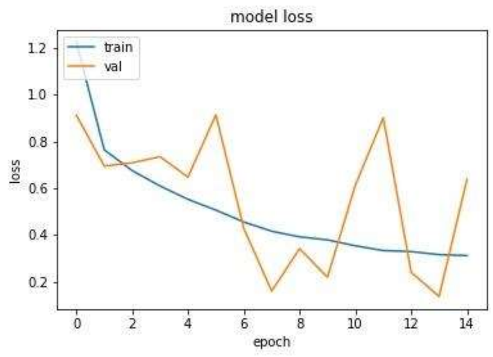




The model training accuracy is 0.9265 while the validation accuracy is 0.8911.
The model training loss is 0.2322 while the validation loss is 0.3840.
The increase in performance between the model with data augmentation is significant: the contrast between the training loss and validation loss is less visible, followed by an increase in the validation accuracy.

With Data Augmentation and with AdaBound *<u>– Best model</u>*





Resnet50 Light	Accuracy	Loss
Train	0.9037	0.3112
Validation	0.8937	0.6381
Test	0.8770	0.4817

The ResNet50 model really works well with the Source Light dataset. Even though the validation loss is not the best obtained till now – maybe highlighting the difficulty of the task, the validation accuracy chases strictly the training accuracy. The test values remark the righteousness of the model and led to the choice of the AdaBound optimizer on the Adam optimizer – the results are really close.

FromScratch

The custom built Convolutional Neural Network is called FromScratch.

It is coded to allow a comparison between a custom CNN and Transfer Learning, maybe overcoming the possible disadvantages of the last one or the utilization of an entire pre-trained CNN. As it is fully trained on the specific datasets of the project, the weights, since the early layers, learn the features that are own of the CelebA Dataset.

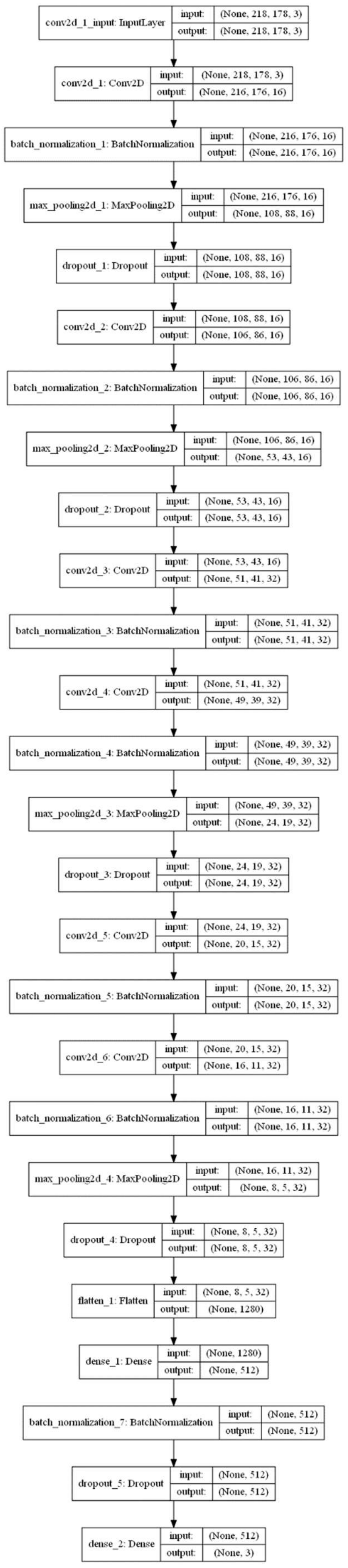
The aim is to create a neural network that works well as a classifier even less deep and less large than the previous adopted – the number of internal parameters will be taken into account.

The blocks within the 'From Scratch' are inspired from the VGG-19 but some modifications are brought in order to renew the architecture with the more recent layers, such as Batch Normalization layer.
The layers used in the custom model (will be shown shortly after this introduction) are:

- Convolutional2D layers
  Layers that apply the convolution operations to the input. The operation is performed to specific regions of the input at a time using a kernel – called also filter – then the application of that filter systematically across the entire input image allows the filter to discover the feature anywhere in the image.
- Dense layers
  Also called fully connected layers because each input neuron is connected to each output neuron.
- Flatten layer
  Its work is to flatten the input and generates an 1D array as output.
  If the input is (5, 5, 20), then the flatten layer squeeze the input along 1D array (5x5x20)
- Max Pooling layer
  Pooling layers are applied in order to reduce dimensionality. It's possible to choose between average pooling and max pooling layers. While the first creates the average of the values within the pool, the second one keeps the max values.
  We adopted the max pooling layers in order to highlight the extreme and most important features.
- Dropout layers
  It allows to simulate sparsity in data disabling some neurons at time. The number of neurons disabled is dependent from the rate chosen.
- Batch Normalization layers
  It is a form of regularization of the data moving them to mean near to 0 and standard deviation near to 1.

The advantage is to speed-up the training and, at the same time, slightly prevent overfitting – as a form of regularization. A discussion is, at the moment, raised in the deep learning community on where to put this layer. On one hand, researchers think that it would be better to insert the batch normalization right before the activation, while others found that batch normalization performs well right after the activation layer. We adopted the second one.

Architecture:



conv2d_1_input: InputLayer input: (None, 218, 178, 3) output: (None, 218, 178, 3)
conv2d_1: Conv2D input: (None, 218, 178, 3) output: (None, 216, 176, 16)
batch_normalization_1: BatchNormalization input: (None, 216, 176, 16) output: (None, 216, 176, 16)
max_pooling2d_1: MaxPooling2D input: (None, 216, 176, 16) output: (None, 108, 88, 16)
dropout_1: Dropout input: (None, 108, 88, 16) output: (None, 108, 88, 16)
conv2d_2: Conv2D input: (None, 108, 88, 16) output: (None, 106, 86, 16)
batch_normalization_2: BatchNormalization input: (None, 106, 86, 16) output: (None, 106, 86, 16)
max_pooling2d_2: MaxPooling2D input: (None, 106, 86, 16) output: (None, 53, 43, 16)
dropout_2: Dropout input: (None, 53, 43, 16) output: (None, 53, 43, 16)
conv2d_3: Conv2D input: (None, 53, 43, 16) output: (None, 51, 41, 32)
batch_normalization_3: BatchNormalization input: (None, 51, 41, 32) output: (None, 51, 41, 32)
conv2d_4: Conv2D input: (None, 51, 41, 32) output: (None, 49, 39, 32)
batch_normalization_4: BatchNormalization input: (None, 49, 39, 32) output: (None, 49, 39, 32)
max_pooling2d_3: MaxPooling2D input: (None, 49, 39, 32) output: (None, 24, 19, 32)
dropout_3: Dropout input: (None, 24, 19, 32) output: (None, 24, 19, 32)
conv2d_5: Conv2D input: (None, 24, 19, 32) output: (None, 20, 15, 32)
batch_normalization_5: BatchNormalization input: (None, 20, 15, 32) output: (None, 20, 15, 32)
conv2d_6: Conv2D input: (None, 20, 15, 32) output: (None, 16, 11, 32)
batch_normalization_6: BatchNormalization input: (None, 16, 11, 32) output: (None, 16, 11, 32)
max_pooling2d_4: MaxPooling2D input: (None, 16, 11, 32) output: (None, 8, 5, 32)
dropout_4: Dropout input: (None, 8, 5, 32) output: (None, 8, 5, 32)
flatten_1: Flatten input: (None, 8, 5, 32) output: (None, 1280)
dense_1: Dense input: (None, 1280) output: (None, 512)
batch_normalization_7: BatchNormalization input: (None, 512) output: (None, 512)
dropout_5: Dropout input: (None, 512) output: (None, 512)
dense_2: Dense input: (None, 512) output: (None, 3)

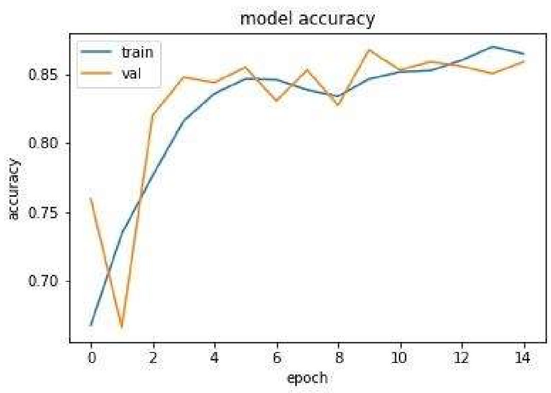
Some trade-off situations have popped out during the fine-tuning of hyperparameters.
The hyperparameters interested in the deep search are the batch size, the number of epochs, the drop rate and finally the learning rate of the optimizer.
The batch size is an important figure as it is the number of images during training and validation since the usage of batch normalization, that computes the normalization within the batch, the batch. It ended up that small batches can offer a regularizing effect, perhaps due to the noise they add to the learning process. Generalization error is often best for a batch size of 1. Training with such a small batch size might require a small learning rate to maintain stability because of the high variance in the estimate of the gradient.
The number of epochs always moved between 10, 15 and 20 both for computational reasons and for preventing KPI getting worse.
The Dropout layers are placed right after the Max Pooling layers and not before to prevent the usefulness of both the types of layers – if the dropout layer is before the pooling layer, we lose too much information. During fine tuning, we found that a low drop rate – around 20/30% - and then an aggressive rate (70%) after the dense layer, helped to prevent overfitting and increase the robustness of the model.
Small learning rate – 0.001 to 0.005 has been found performing well even if a limited number of epochs. Talking about the choice of the best optimizer, in the early stage of the project the Adam was the one which has been used, reaching good values of validation (and test) accuracy while the validation loss could improve more. SGD, Stochastic Gradient Descent performs very well on this custom CNN probably because it is known to better generalize than the Adam optimizer.
So, we adopted the strategy of firstly reaching good results with Adam and then switching to SGD to fine-tune the From Scratch neural network.

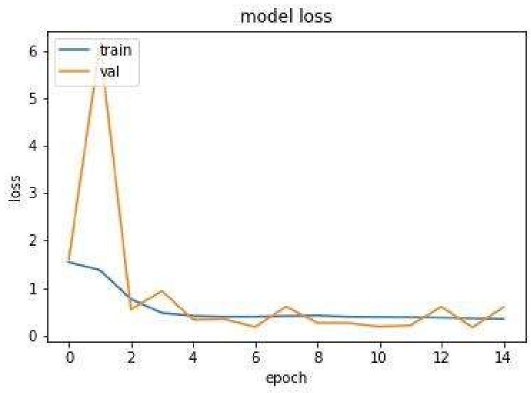
The total number of parameters of the CNN is restrained thinking at other CNNs and at the VGG, source for our custom blocks: less than 700’000.

Orient results:

The best ‘From Scratch’ custom CNN is composed of 2D Convolutional layers with 16 depth, batch size of 16, dropout rate of 0.7 and the learning rate value is 0.005. The optimizer chosen is the Stochastic Gradient Descent.

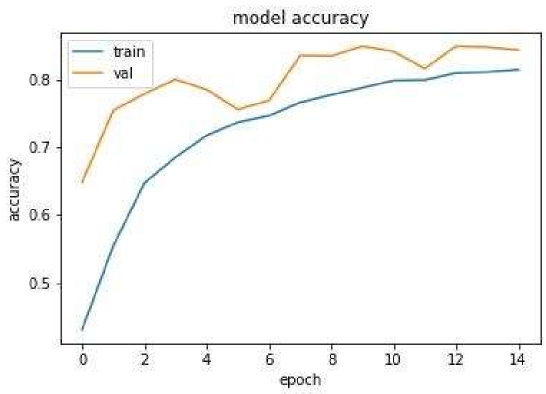
No Data Augmentation and with AdaBound

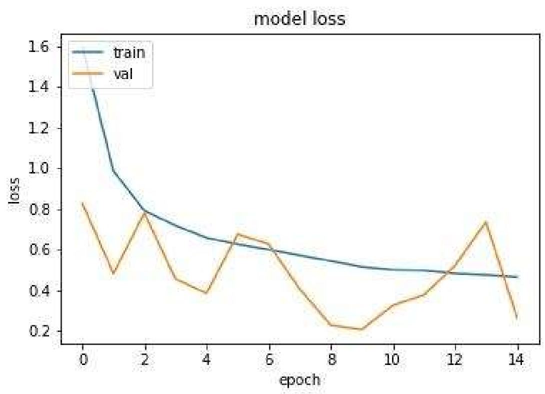




The model training accuracy is 0.9265 while the validation accuracy is 0.8911.
The model training loss is 0.2322 while the validation loss is 0.3840.

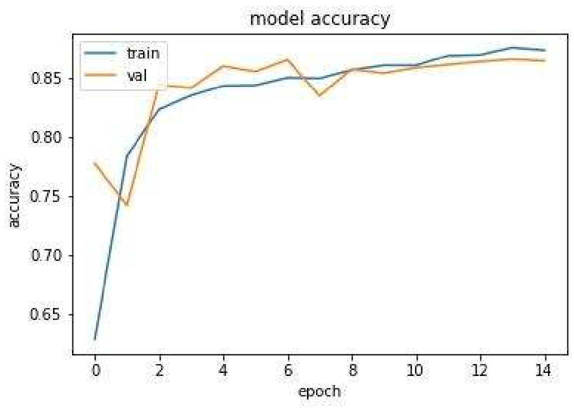
With Data Augmentation and with Stochastic Gradient Descent (SGD)

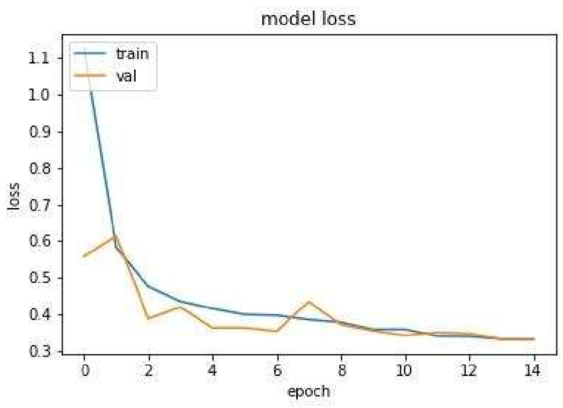




The model training accuracy is 0.9265 while the validation accuracy is 0.8911.
The model training loss is 0.2322 while the validation loss is 0.3840.
The application of Data Augmentation in the custom convolutional neural network brings a great bias in the results. The validation metrics are always above the line of the training metrics and this cannot be due to the randomness inherent in the learning phase.
One possible motivation is that the custom model trains whole on the dataset, thus there is no need on perturbing the inputs in order to decrease possibility of overfitting. While the data augmentation can be a benefit in transfer learning applications where the depth of the network can lead to overfit primarily when the project dataset is smaller than the pre-training one.

No Data Augmentation and with Stochastic Gradient Descent (SGD) – <u>Best model</u>



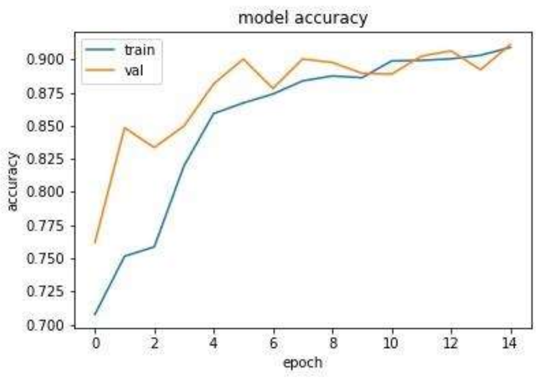


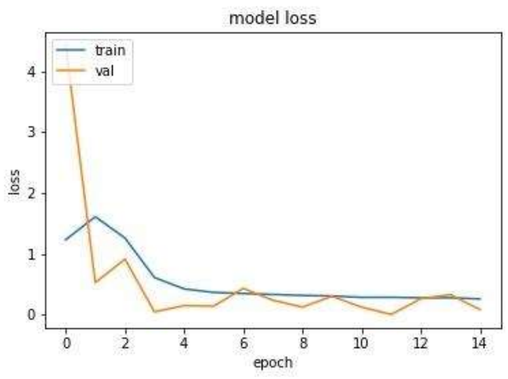
FromScratch Orient	Accuracy	Loss
Train	0.8734	0.3312
Validation	0.8645	0.3115
Test	0.8809	0.3093

Source light results:

The best 'From Scratch' custom CNN is composed of 2D Convolutional layers with 16 depth, batch size of 16, dropout rate of 0.7 and the learning rate value is 0.005. The optimizer chosen is the Stochastic Gradient Descent. It has the same architecture and hyperparameters of the CNN dedicated to the orient classification.

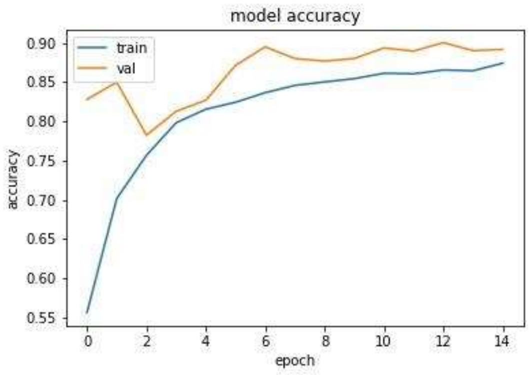
No Data Augmentation and with AdaBound

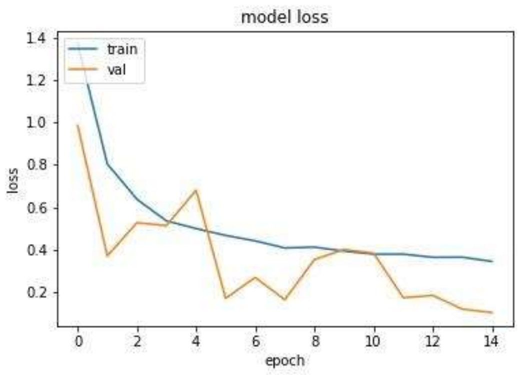




The model training accuracy is 0.9113 while the validation accuracy is 0.9178.
The model training loss is 0.2759 while the validation loss is 0.1232.

With Data Augmentation and with Stochastic Gradient Descent (SGD)



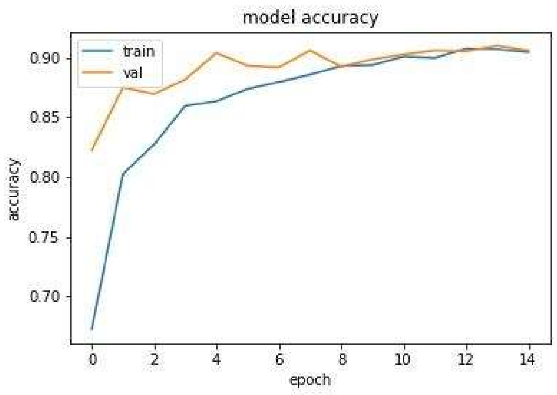


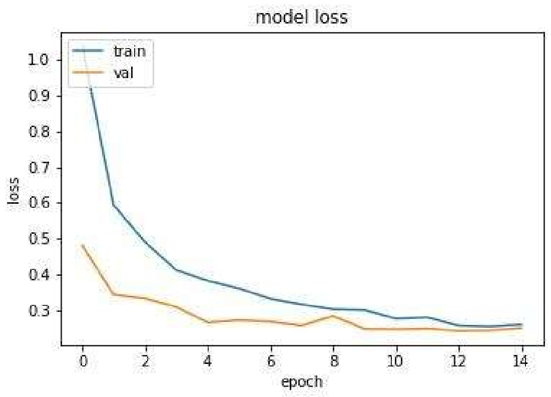
The model training accuracy is 0.9265 while the validation accuracy is 0.8911.

The model training loss is 0.2322 while the validation loss is 0.3840.

The same considerations made above in the orient dataset can be done also for the source light task.

No Data Augmentation and with Stochastic Gradient Descent (SGD) – <u>Best model</u>





FromScratch Accuracy	Accuracy	Loss
Train	0.9046	0.2599
Validation	0.9059	0.2500
Test	0.9019	0.2571

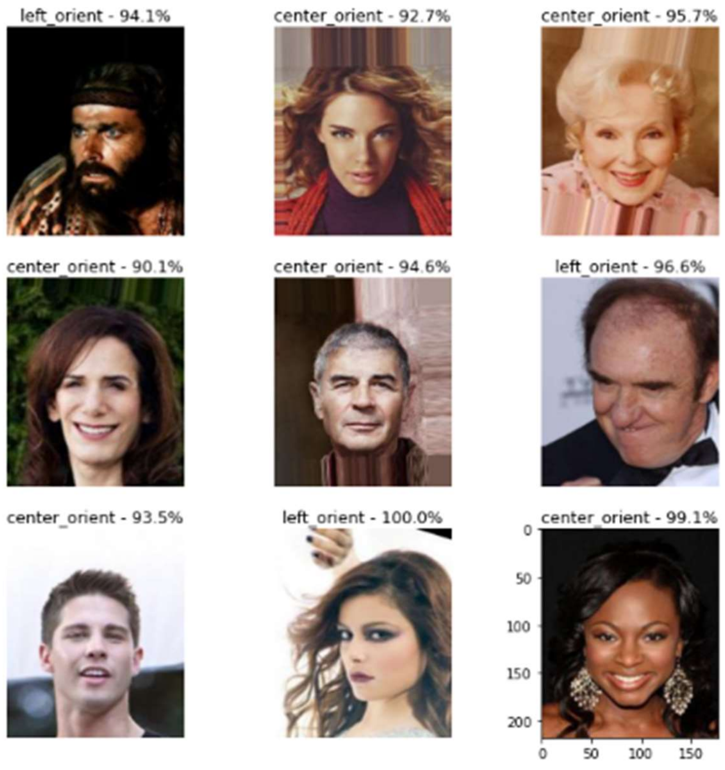
Despite the AdaBound worked very well also in this model – confirming the quality of this new optimizer – the test accuracy and loss led the choice towards SGD. The reason is that, if the validation is important for detecting overfitting and tuning the parameters, the test set is important too for performance evaluation (since the images are unseen during training).

Results on whole dataset

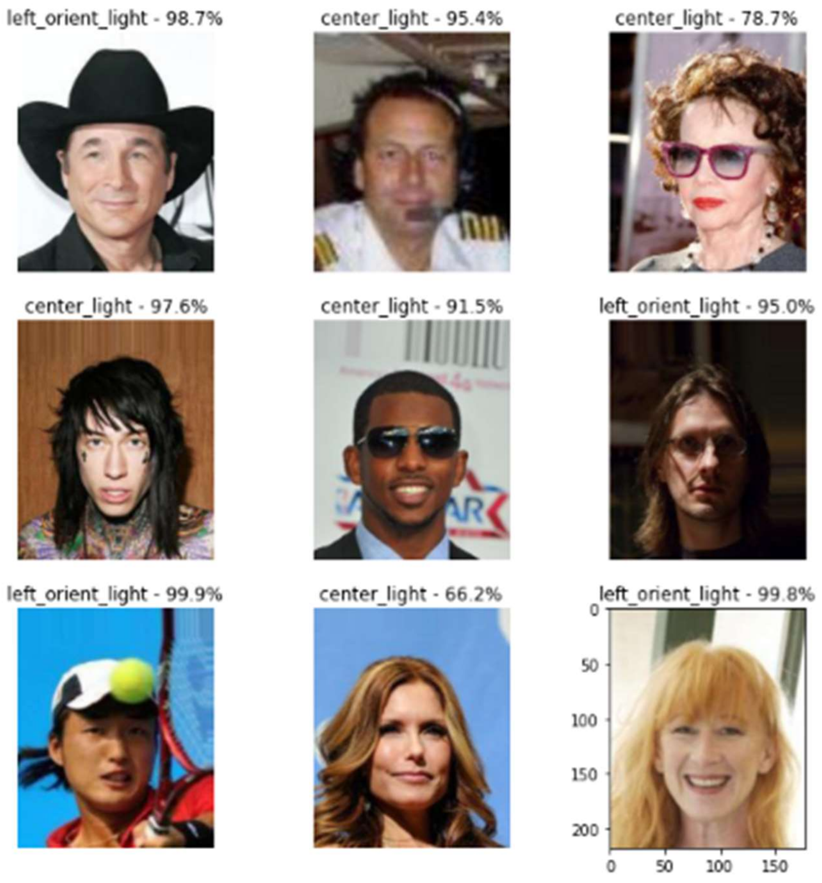
After several iterations for finding the best model and testing data augmentation and various optimizers we found the final 3 best models for each algorithm: inceptionv3, resnet50 and from scratch, each model splitted in 2: face orientation and source light. Finally we got 6 best performing models and we tried the test set on them.

Test set	Accuracy	Loss
INCEPTIONV3 (face orientation)	0.8603	0.3000
INCEPTIONV3 (source light)	0.9035	0.3977
RESNET50(face orientation)	0.8656	0.3261
RESNET50(source light)	0.8770	0.4817
FROM_SCRATCH(face orientation)	0.8809	0.3092
FROM_SCRATCH (source light)	0.9018	0.2570

Thinking about the validation metrics' values achieved and the behaviour with unseen images, we decided to choose the 'From scratch model' for both the experiment: face orientation and source light.
We run the models on the entire dataset, and here you can find a demo for both the experiments on 9 images per dataset.
Face orientation:



Source light:



The predictions made for face orientation and source light origin on the whole CelebA dataset are saved in a .csv file. The csv is composed of two columns that identify the labels and the predictions in percentage terms.

Final considerations

The results are quite good, with the custom neural network that has achieved what expected, so the transfer learning models after the patch.
Data augmentation worked better on source light as it increased the possible scenarios transforming on the fly the input images. While on the orient task, data augmentation sometimes also brought biases.
The reason why the source light task has obtained better results in comparison with the orient problem is maybe motivated by some biases unconsciously introduced in the manual classification. On the other hand, while the orient task requests a global understanding of the image, the source of light is often due to the gradient of light (from pixels which have high values, around 255, to pixels close to 0 values) - the most difficult images are, as fully explained in the introduction, the ones which have reflections or some artificial light setup.

In conclusion, we followed the pipeline by firstly pre-process the datasets, then we focused on the transfer learning approach by choosing InceptionV3 and ResNet50 models and fine-tune them based on the choice of hyperparameters, the type of optimizer and the use of Data Augmentation.
The same work has been done in order to build the custom CNN.
Finally, we have brought the two best models overall - one for each task - and apply them on the whole CelebA dataset.

Bibliography and Sitography

Transfer learning

Papers:

Survey on transfer learning: https://www.cse.ust.hk/~qyang/Docs/2009/tkde_transfer_learning.pdf

Articles:

Transfer Learning: https://cs231n.github.io/transfer-learning/

Transfer Learning in Keras: https://medium.com/@14prakash/transfer-learning-using-keras-d804b2e04ef8

Inception v3

Papers:

Inception v1: https://arxiv.org/pdf/1409.4842v1.pdf

Inception v2 and v3: https://arxiv.org/pdf/1512.00567v3.pdf

Articles:

Review inception v3: https://medium.com/@sh.tsang/review-inception-v3-1st-runner-up-image-classification-in-ilsvrc-2015-17915421f77c#:~:text=With%20multi%2Dmodel%20multi%2Dcrop,which%20will%20be%20reviewed%20later.

Inception Networks: https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202

Residual Learning

Papers:

ResNet paper: https://arxiv.org/pdf/1512.03385.pdf

Articles:

Introduction to ResNet: https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4

Vanishing gradient problem and residual connections: https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484

CNN From Scratch

Papers:

First Convolutional Neural Network paper by LeCun
https://www.eecis.udel.edu/~shatkay/Course/papers/NetworksAndCNNClasifiersIntroVapnik95.pdf

Articles:

All different types of convolutions
https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215

Efficient way to build a CNN: https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-ii-hyper-parameter-42efca01e5d7

Batch Normalization

Papers:
Batch Normalization paper: https://arxiv.org/pdf/1502.03167.pdf%20http://arxiv.org/abs/1502.03167.pdf
BN and Dropout: https://link.springer.com/article/10.1007/s11042-019-08453-9

Articles:
Batch Normalization in training: https://machinelearningmastery.com/batch-normalization-for-training-of-deep-neural-networks/
BN in Keras: https://keras.io/api/layers/normalization_layers/batch_normalization/

Batch Normalization problem in Keras
Articles:
Problem: https://github.com/keras-team/keras/pull/9965
http://blog.datumbox.com/the-batch-normalization-layer-of-keras-is-broken/

Optimizers

Papers:
Improving generalization performance by switching from Adam to SGD: https://arxiv.org/abs/1712.07628
AdaBound: https://arxiv.org/abs/1902.09843

Articles:
An overview of the optimizers: https://ruder.io/optimizing-gradient-descent/
Adam and then SGD: https://towardsdatascience.com/adam-in-2019-whats-the-next-adam-optimizer-e9b4a924b34f
SGD or Adam: https://shaoanlu.wordpress.com/2017/05/29/sgd-all-which-one-is-the-best-optimizer-dogs-vs-cats-toy-experiment/
AdaBound: https://medium.com/syncedreview/iclr-2019-fast-as-adam-good-as-sgd-new-optimizer-has-both-78e37e8f9a34

Data Augmentation

https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/
https://www.pyimagesearch.com/2019/07/08/keras-imagedatagenerator-and-data-augmentation/