Multi Task Learning on Face Images with a Task Constrained Deep Convolutional Neural Network

Emanuele Alessi (1486470)

Gianmarco Forcella (1725967)

Summary

[1. Introduction 3](#_Toc14352062)

[2. Theoretical overview 4](#_Toc14352063)

[2.1 Convolutional Neural Networks 4](#_Toc14352064)

[2.2 Multi Task Learning 4](#_Toc14352065)

[2.3 TensorFlow 5](#_Toc14352066)

[2.4 OpenCV 6](#_Toc14352067)

[3. Literature Study 7](#_Toc14352068)

[4. Project setup 8](#_Toc14352069)

[4.1 The MAFL Dataset 8](#_Toc14352070)

[5. Data Pre-processing 10](#_Toc14352071)

[6. Data Augmentation 12](#_Toc14352072)

[7. Implementation of a the Multi Task Learning Convolutional Neural Network 14](#_Toc14352073)

[7.1 The Weight initialization problem 16](#_Toc14352074)

[7.2 Prediction 17](#_Toc14352075)

[7.3 Minimizing loss 17](#_Toc14352076)

[7.4 Training and accuracy testing 18](#_Toc14352077)

[8. Model Evaluation 20](#_Toc14352078)

[8.1 Test Run 20](#_Toc14352079)

[8.2 Landmarks results on test images 25](#_Toc14352080)

[9. Conclusions 28](#_Toc14352081)

[References 29](#_Toc14352082)

# Introduction

In Computer Vision tasks, it is often asked to have a Neural Network that is capable of learning and predicting more things at once. While sometimes this resolves in having more than one Network for every single task, this became inefficient in a very short time. The reasons for this can be easy and straightforward to think of:

1. **High computational power**: this would have required at least a distributed system to make it work. Otherwise, the server would have just overloaded with calculus and could, eventually, crash;
2. **Inefficiency**: for the above reason, it then becomes hard to maintain such an application. Any Machine Learning developer who could face this issue would have to check n computers and their performance with the networks
3. **Response time**: training and predictions would have to be really fast, otherwise the operation could be useless for the final user (might get tired of waiting)

In the early 2000s, though, researchers finally proposed a “way out” to this problem. If we humans can be multitasking, so can computers and, so, can neural networks.

In the following report, we will present our implementation of a Multi Task Learning Convolutional Neural Network, applied to Computer Vision by recognizing the following values:

* Whether the subject wears eyeglasses or not
* Whether the subject has a beard or not
* The subject’s sex
* Whether he/she is smiling
* If the subject is young
* And his/her landmarks (that is, the facial points such as eyes, nose and mouth)

In the following chapters, we will describe our architectural choices while also providing the source code and we will show some “numerical facts” to see if our model has been a good choice or not for our task.

# Theoretical overview

Before going deeper with describing the problem, let us first describe what a Convolutional Neural Network is and give just a quick example on how it is possible to implement one with the most used Deep Learning technologies.

## Convolutional Neural Networks

A Convolutional Neural Network is a Neural Network that is part of the “Deep Learning branch” (since it holds, usually, a minimum of 7 layers) and is considered one of the most powerful network types when it comes to image processing, thanks to the key its structure:

1. Neurons are distributed in 3 dimensions and not all of them are connected to the next layer: only the next to last is fully connected;
2. Weights are shared all along the net.

By exploiting the strong spatially local correlation that each hidden layer can hold, CNN have been proven to be the best choice against Multilayer Neural Networks.

As far as the type of layers that a Convolutional Neural Networks can have, there can be 3 types:

* Convolutional: they compute the output of neurons connected to the input thanks to a kernel, which slides over the input and performs a dot product with the input of the filter and the positions that are close to the input; the output is influenced by some hyperparameters;
* Pooling: usually inserted between one convolutional layer and another, they are used to reduce the number of parameters and computation in the network, in order to avoid overfitting;
* Fully connected layer: usually placed at the end of the Convolutional Neural Network. Since all neurons are fully connected, these layer are going to be treated as a normal neural network.

## Multi Task Learning

The Multi-task learning is a subfield of Machine Learning which started to gain popularity back in 1997. Its objective is to have, in a Machine Learning system, a model which can learn and solve multiple tasks at the same time.

It works because regularization induced by requiring an algorithm to perform well on a related task can be superior to regularization that prevents overfitting by penalizing all complexity uniformly.

## TensorFlow

**TensorFlow** is probably the most famous framework for working out any large-scale Machine Learning: originally created by the *Google Brain Team*, it is an open-source library which bundles mainly Deep Learning models and algorithms.

The library can train and run Deep Neural Networks for many tasks, ranging from digit classification to image recognition.

But how does it work?

TensorFlow allows the creation of so-called “dataflow graphs”; structures that describe how data moves through a graph. Here:

* A node represents a mathematical operation;
* An edge between two nodes symbolize a “Tensor” (short for multidimensional array).

The nodes, though, are not executed in Python: to ensure a higher speed of computation, in fact, the library executes these operations in C++, so that they can be worked out at low-level.

Another great advantage is that the developer can choose to execute calculations either on the CPU or the GPU, to ensure more computational power to the program.

As of 2019, TensorFlow is accredited as one of the most used libraries for Deep Learning and it keeps growing, even with a recent release for JavaScript.

As we felt that TensorFlow was what we needed for this task (since it is more powerful than Keras), we decided to abandon the advantage of having less and more concise code lines in favor of more computational power.

For this reason, we will not list a code example here as our project was entirely made with TensorFlow.

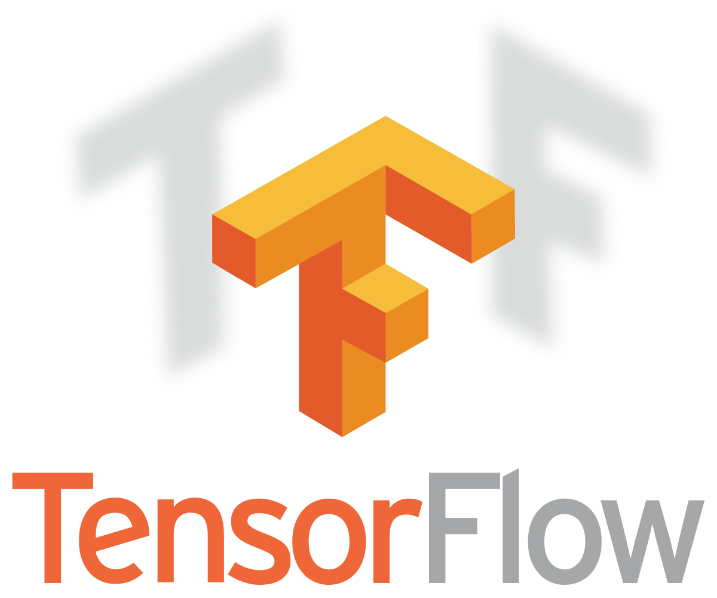


Figure 1 - TensorFlow logo

## OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications.

The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, etc.

OpenCV is a cross-platform library using which we can develop real-time computer vision applications. It mainly focuses on image processing, video capture and analysis including features like face detection and object detection and it has more than 47 thousand people of user community and estimated number of downloads exceeding 18 milion.

The library is used extensively in companies, research groups and by governmental bodies.

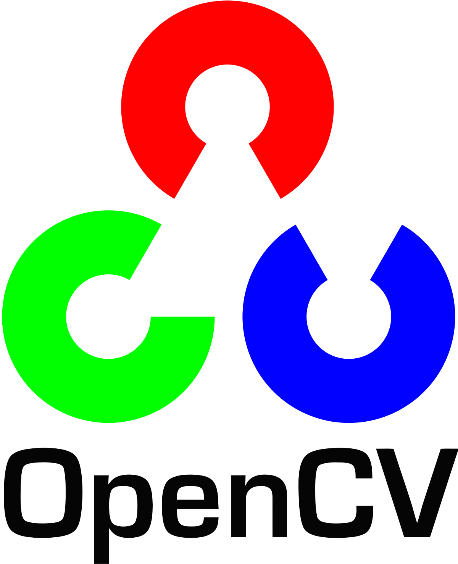


Figure 2 - OpenCV logo

# Literature Study

In order to implement our Multi Task Learning Deep Neural Network, we decided to take inspiration from this paper and this as well.

These two have a huge issue, though: they are quite old (published 5 years ago). Due to the age, it is likely that the results contained are not valid anymore for many reasons.

To avoid in doing a non-performant model, we decided to try to implement our alternative solution, which goal is to have a better accuracy in terms of landmarks localization and avoid the overfitting that might happen with the classification tasks. To enhance this, we also implemented techniques of data augmentation (which we will discuss later on this paper) and kept track of all losses and RMSE.

Also, in the papers, the researchers used a task-wise early stopping technique on each task, that was able to stop the training of the network on a specific task when it was detected that overfitting was happening. In our alternative model, instead, we decided to implement regularization and batch normalization to avoid the same problem.

After an accurate study of our dataset, we decided to make prediction on the following six tasks:

* Landmarks
* Eyeglasses
* Gender
* Facial Hair (whether the subject has a beard or not)
* Smile
* Age (if the subject is young or not)

# Project setup

Let us know enlist the technologies that we used while working on this project (in order of importance).

* **Python 3.6.x**
* **TensorFlow**
* **The MAFL dataset** (which is a collection of frontal and profile views of face images that holds information such as the position of the landmarks, whether the subject wears glasses and so on).
* **NumPy** (used to process the dataset and to calculate metrics of the Neural Network)
* **PIL** (used to load the images and to convert them into NumPy array)
* **Pandas** (used to process the csv file of landmarks)
* **Tqdm** (used for printing a progress bar of the neural network training process)
* **PyCharm as our IDE**

To check out the full project, please refer to [this](https://github.com/leleea7/Neural-Networks) GitHub repository, which also contains all the papers we took inspiration from for our work.

## The MAFL Dataset

The Multi-Attribute Facial Landmark (MAFL) dataset, proposed by Zhang et al., is annotated with 5 facial landmarks (right corner eye, left corner eye, nose, left corner mouth, right corner mouth) with 40 different facial attributes. There are 20,000 faces present in the database. MAFL can be downloaded from <https://github.com/zhzhanp/TCDCN-face-alignment>. Here is a sample of the images present in MAFL:



Figure 3 - Sample of images from MAFL dataset

Proposed by Liu et al. and reproduced from the Multimedia Laboratoryof the The Chinese University of Hong Kong (<http://mmlab.ie.cuhk.edu.hk/projects/TCDCN.html>).

The attributes of annotation include pointy-nose, bands, moustache, wavy hair, wearing a hat and so on. These images are included in the CelebA dataset as well, which is a large-scale face attributes dataset with more than 200000 celebrity images, each with 40attribute annotations. The images in this dataset cover large pose variations and background clutter. CelebA has large diversities, large quantities, and rich annotations, including: 10177 number of identities, 202599 number of face images, and 5 landmark locations, 40 binary attributes annotations per image.

# Data Pre-processing

Before we can work on the dataset, we must load it in memory and apply some pre-processing techniques.

Our procedure consists of iterating over two files that contains file names that has to be loaded either for training or testing purposes: being that, as stated before, MAFL is a subset of CelebA, the original authors decided just to prepare two .txt files for the previously cited purposes.

For our purposes, we decided to load the following samples:

* About 19000 images for the training process, each epoch
* 1000 for the testing one

When loading each image, there are some steps done before giving it to our network:

* We load the image in gray-scale, through the PIL library;
* Each image is then resized to 48x48 using the LANCZOS resampling technique;
* We then apply an augmentation process (the **shift**) to the image (moving it with a random number in the range [-10,10])
* Finally, it is converted into a NumPy array for use with TensorFlow.

SOSTITUIRE IMMAGINE!!!!

|  |  |
| --- | --- |
| **Train** | **Test** |
| |  |  | | --- | --- | |  |  | |  |  | |  |  | |  | | | |  |  | | --- | --- | |  |  | |  | | |

Figure 4 An example of how train-test splitting is done (Bill Clinton, id 067)

|  |  |  |
| --- | --- | --- |
| 400 x 499 | Resize | 48 x 48 |

Figure 5 An example of how image resizing is done

# Data Augmentation

The performance of deep learning neural networks often improves with the amount of data available.

Data augmentation is a technique to artificially create new training data from existing training data. This is done by applying domain-specific techniques to examples from the training data that create new and different training examples.

Image data augmentation is perhaps the most well-known type of data augmentation and involves creating transformed versions of images in the training dataset that belong to the same class as the original image.

Transforms include a range of operations from the field of image manipulation, such as shifts, flips, zooms, and much more.

The intent is to expand the training dataset with new, plausible examples. This means, variations of the training set images that are likely to be seen by the model.

As such, it is clear that the choice of the specific data augmentation techniques used for a training dataset must be chosen carefully and within the context of the training dataset and knowledge of the problem domain. In addition, it can be useful to experiment with data augmentation methods in isolation and in concert to see if they result in a measurable improvement to model performance, perhaps with a small prototype dataset, model, and training run.

Modern deep learning algorithms, such as the convolutional neural network, or CNN, can learn features that are invariant to their location in the image. Nevertheless, augmentation can further aid in this transform invariant approach to learning and can aid the model in learning features that are also invariant to transforms such as left-to-right to top-to-bottom ordering, light levels in photographs, and more.

We applied data augmentation, through openCV’s function *warpAffine*, only to the training dataset, and not to the test dataset.

For our purposes, we decided to augment the train set with height-shift or width-shift (thanks to OpenCV).



Figure 6 - An example of how data augmentation is applied on a given image

With the data augmentation technique, our final model improved its predictive ability in the landmarks, reducing our RMSE from 1.95 to 0.91.

# Implementation of a the Multi Task Learning Convolutional Neural Network

As already stated, our implementation takes free inspiration from this paper and this one too, where two models of Task Constrained Deep Convolutional Network (TCDCN) were proposed: although the implementation from paper B was better than the first, we were forced to pursue the proposed implementation of paper A because of hardware restrictions. We reached this conclusion after several attempts of training the network on the computer that we were using, equipped with an **nVidia** **GeForce 840M Graphic Board** **Card**.

The first thing we do is, of course, initialize the class: we decided to build it in this way in order to have a better reuse of the code and, eventually, to use it in the future as a module.

As can be seen, we never get to load the full dataset in one shot because we’ve seen that it can be a really slow operation: we will get back to this later on.

Loading each image in memory means, to us, converting it to an Image object in Python, convert it as a 48x48x1 NumPy Array (because we load it with a 1 color channel), cast it to a simple float array and returning it, alongside with its size.

After that, we define the core of our Multi Task Learning Neural Network.

Through TensorFlow (and with the contribution of Keras which, starting from the recent versions, has been integrated inside Google’s library with the tf.keras module), we use a layer variable that may seem to get reinitialized with a new one, but it is not like that: the Deep Learning framework allows us to create a new layer and add it like if it was on a stack. When a layer is assigned, it **also** gets executed: this means that the user doesn’t have to use something like a .build() method and have everything executed all at once. TensorFlow allows to inspect the progress of the calculations while the input flows from layer to layer.

As stated in a previous paragraph, we decided to use the following techniques to ensure better results:

* **Regularization**: it allows to apply penalties on layer parameters during optimization. These penalties are incorporated in the loss function that the network optimizes. Argument in convolution layer is nothing but L2 regularization of the weights. This penalizes peaky weights and makes sure that all the inputs are considered.

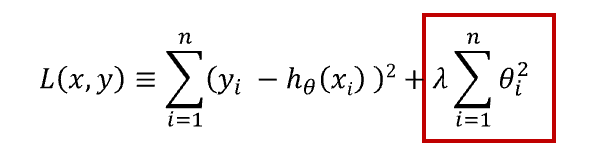


Figure 7 - Regularization term (in red box) with lambda fixed to 0.01

* **Batch normalization**: it normalizes the activation of the previous layer at each batch, i.e. applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1. It addresses the problem of internal covariate shift. It also acts as a regularizer, in some cases eliminating the need for Dropout. It helps in speeding up the training process. This function trains two parameters, and , in order to use them in the calculation of the normalization on each layer. In the below formula,

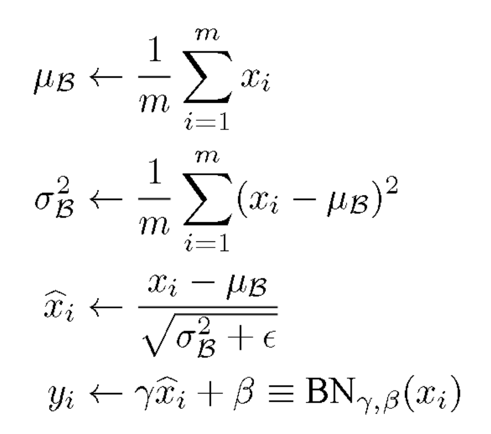


Figure 8 - Batch normalization formula

* **ReLU (Rectified Linear Unit)** activation function: it is the most used activation function in the world right now. Since, it is used in almost all the convolutional neural networks or deep learning, thanks also to its derivative that is effective for the vanishing gradient problem

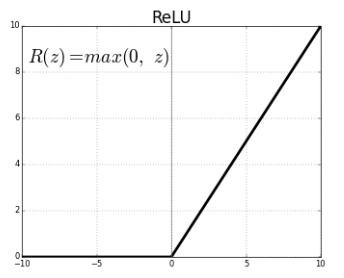


Figure 9 - Graphic representation of the ReLU activation function

* **Max Pooling:** it is a **discretization process**. The objective is to down-sample an input representation (image, hidden-layer output matrix, etc.), reducing its dimensionality and allowing for assumptions to be made about features contained in the sub-regions binned.

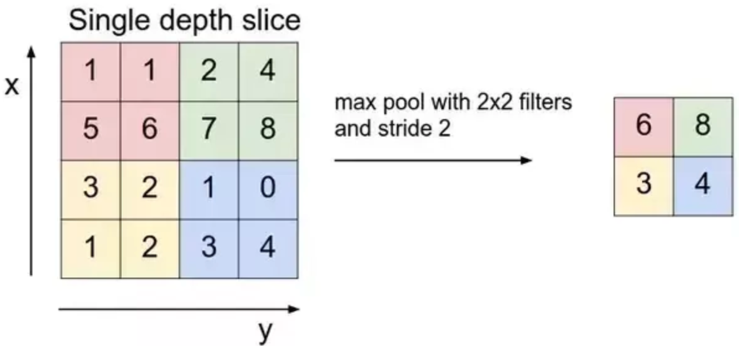


Figure 10 - An example of how Max Pooling is performed

Regarding the list of the layers used, it is as follows:

* We initialize a **Convolutional Layer** with **ReLu** as the activation function. The kernel size starts as a 5x5 array and it gets shrinked to 2x2 and the filters are, at the beginning, 16: the number increases every time a new Convolutional layer gets initialized;
* After every Convolutional layer, we have a **Max Pooling** layer whose purpose is to halve its input. We also add a stride of 2, so that the algorithm can shift over the input matrix by a factor of 2 pixel at a time.
* Right before the fully connected layer, there is a **Flatten Layer** that “collapses” its input to just one dimension.
* The final layer is a **Fully Connected** one, which symbolizes the fully connected Neural Network; it is made of 100 neurons and uses **ReLu** as its activation function. As for the input, it takes what the layer variable holds on that point of the program, which is now a one-dimensional vector of values.

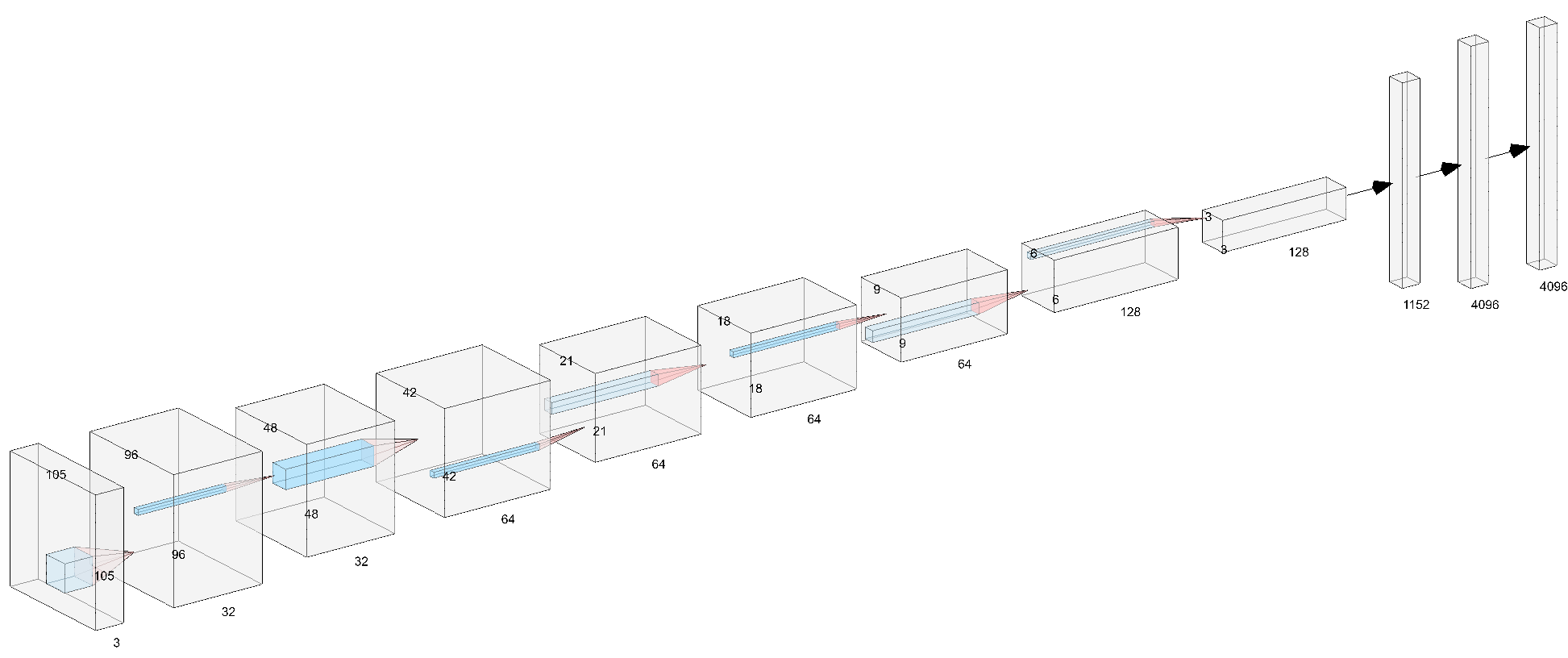


Figure 11 3D representation of our Siamese Network

The Siamese Network architecture follows the pattern:

INPUT -> [[CONV -> RELU] -> POOL] \* 4 -> FLATTEN -> [FC -> SIGMOID]

Notice that after all this there is another operation performed on the outputs of the two twin conv nets, that is the absolute value subtraction between the first image and the second image outputs, which will be our similarity value.

## The Weight initialization problem

In the beginning, we initialized all the Convolutional Layers’ weights with the following value

1. weights\_initializer=tf.truncated\_normal\_initializer(mean=0.0, stddev=0.01)

which didn’t allow us to produce any satisfying predictions as output. We then decided to change this part of our implementation and opted for the [**Glorot-Bengio weight initialization technique**](http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf)**,** which is the standard way TensorFlow initializes the weights of a Convolutional Layer: as specified both in the paper (formula #16) and [in the official TensorFlow’s documentation](https://www.tensorflow.org/api_docs/python/tf/contrib/layers/xavier_initializer) for this technique, weights are going to be initialized with a value that ranges from minus the square root of 6 divided by the square root of the sum between the input units and the output units up to the positive value of the very same formula.

By doing so, we achieved better results in terms of accuracy.

## Prediction

After the optimization process, there is the prediction step, which its formula all depends on the task the network is facing:

* If the Network is trying to predict the landmarks, then it will use the **multiple linear regression** formula, which is
* Else, if it is facing a classification task, it will use the formula for classifying in Neural Networks, that is the **Softmax function** which returns a probability Tensor (that is, a Tensorflow array) that is associated with each instance in the batch and, for every instance in the batch, this Tensor is given in input to the **argmax** function (), which will return the Tensor’s index with the best probability value that will become and compared with a truth value for the loss’s calculation.

## Minimizing loss

In order to minimize the loss that each layer produces at any epoch, we decided to not apply a standard optimizer (like stochastic gradient descent) but to use [**Adam’s**](https://arxiv.org/pdf/1412.6980.pdf). Adam is an adaptive learning rate optimization algorithm that’s been designed specifically for training deep neural networks and a lot of research has been done to address the problems of Adam. It can be used instead of the stochastic gradient descent procedure to update network weights. Adam is an optimization algorithm.

The main reasons on why anybody should use this procedure are:

* It is **easy** to implement;
* It is **computationally** efficient;
* It requires a small amount of memory;
* It is invariant to the diagonal rescale of gradients.

But how does it work? Adam can be looked at as a combination of RMSprop and Stochastic Gradient Descent with momentum. It uses the squared gradients to scale the learning rate like RMSprop and it takes advantage of momentum by using moving average of the gradient instead of gradient itself like SGD with momentum. In this optimization algorithm, running averages of both the gradients and the second moments of the gradients are used. Given parameters *w(t)* and a loss function *L(t)*, where *t* (initially set to 0)is the current training iteration, Adam’s parameter update is given by:

|  |  |
| --- | --- |
| 1. |  |
| 2. |  |
| 3. |  |
| 4. |  |
| 5. |  |

In our case, , , and .

As our network is trained to do both classification and regression, we decided to use the **Root Mean Square Error** (RMSE) as loss function for predicting landmarks.

Regarding the labels that are predicted by classification, defined in a set S = {Eyeglasses, Gender, Beard, Smile, Young}, the loss function, instead, is defined through the **softmax cross entropy** (also known as categorical cross entropy)formula, which is the following:

Once the losses for each label are calculated, the weights of the neural network will be updated according to the sum of all losses Ltotal.

## Training and accuracy testing

In order to create batches for training and testing our Multi Task Learning Neural Network, we go under two processes: first, we need to generate the batch that will be sent to the Neural Network for the training phase. As stated before, this is done by loading approximately 594 batches (that is, because we load 19000 training instances divided by the batch size, in our case 32) per epoch and sending them to the network, which will then try to predict its tasks.

After that, every 100 iterative steps, we run a test to understand what the metrics of the Convolutional Neural Network at that stage are. We do this by feeding the model trained up to that point a sample of the test set (in our case a support set of 1000 samples) that is, of course, always made of different images.

Given this support set, the model is tasked to perform regression for the landmarks and classification for all the other tasks.

# Model Evaluation

The evaluation process starts by initializing a TensorFlow Session and assigning to it the graph variable that was previously created.

After that, we make all the Graph’s variables initialize through the

session.run(tf.global\_variables\_initializer())

function. The specified input to .run() allows the Graph to have all its variables initialized, while the .run() method performs the specified action in input.

We then start the computation of the Network, by iterating to perform a good training. To get the dataset for training, we decided to generate a function (which can be read in [Appendix E](#_Appendix_E_–)) that will load, each time it receives a path, the image and will also augment it for the network.

Every 100 iterations, there is also a “check step”: at that point, the algorithm will pick up a new portion of the dataset (Appendix F), so that it can perform some testing.

## Test Run

By making test runs of our Network, we reach the following results, expressed in terms of Accuracy and Loss (or RMSE for landmarks).

Notice that the accuracies are calculated on the test set, while the losses are calculated on the training set.

|  |  |
| --- | --- |
| Figure 12 - Eyeglasses accuracy | Figure 13 - Gender accuracy |
| Figure 14 - Facial hairs accuracy | Figure 15 - Smile accuracy |
| Figure 16 - Age accuracy | |

As it is possible to see from the figures, the accuracy that the Multi Task Learning Network can provide, in classification tasks, is immediately very high and it can reach in just a few iterations a value between the 85-95%. This may happen because of the difficulty of the task.

For the RMSE on landmarks, it is possible to see that the Network is immediately able to compute, at first, a slightly low RMSE (about 4.5) and, after 4 epochs (We remind that, in this case, an epoch consists of 594 iterations!), get to 0.91.

Even the loss of the tasks has an interesting plot: we could say that it has a decreasing process that recalls the plot of an exponential function.

|  |  |
| --- | --- |
| Figure 17 - Eyeglasses loss | Figure 18 - Landmarks loss |
| Figure 19 - Gender loss | Figure 20 - Facial hairs loss |
| Figure 21 - Smile loss | Figure 22 - Age loss |
| Figure 23 - Total loss | |

Regarding the landmarks localization task, we obtained the following results on the test set:

|  |  |
| --- | --- |
| Figure 24 - RMSE landmarks | Figure 25 - RMSE of each landmark |

## Landmarks results on test images

After completing the model train and test phase we tested our trained neural network on some image taken from the test set in order to get visual feedbacks on network performance, in terms of landmarks localization, and these are the results:

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

# Conclusions

With our work, we managed to get a great result, in terms of accuracy and RMSE, on both the classification and the regression task without modifying the image on runtime: we had an accuracy that almost every time got over the 80% , while the RMSE reaches a value of 0.9, which is good considering that the objective was to calculate the landmarks.

To reach higher results, though, it was required to do some augmentation on the image that was destined to the training process: by choosing to shift the image, not only we managed to gather at least 5-7% more in terms of accuracy, but we were also able to decrease the RMSE by 1.0.

One concern that might be worthy to look at is the dataset: right now, a batch of 32 items per iteration is loaded in memory. Another option would be to *load the full thing* in memory and then randomize the accesses to the dataset, but that is simply just not possible: having tried it, we could saw how this process required a lot of time to get completed, so we decided to go for the best option in terms of time saving.

Finally, we have noticed that the neural network is not able to locate the facial landmarks of some subjects, especially those whose face is in profile, but the cases are very rare. In any case we feel satisfied with the results obtained considering above all the problem of overfitting mentioned in the literature and that we too we found in our project.

# References

|  |  |
| --- | --- |
| [1] | *Facial Landmark Detection by Deep Multi-task Learning* byZhanpeng Zhang, Ping Luo, Chen Change Loy, and Xiaoou Tang  <http://personal.ie.cuhk.edu.hk/~ccloy/files/eccv_2014_deepfacealign.pdf> |
| [2] | *Learning Deep Representation for Face Alignment with Auxiliary Attributes* by Zhanpeng Zhang, Ping Luo, Chen Change Loy, and Xiaoou Tang  <https://arxiv.org/pdf/1408.3967.pdf> |
| [3] | *Multi Attribute Facial Landmarks dataset*  <https://github.com/zhzhanp/TCDCN-face-alignment> |
| [4] | *CelebA dataset*  <https://drive.google.com/drive/folders/0B7EVK8r0v71pWEZsZE9oNnFzTm8> |
| [5] | *Understanding the difficulty of training deep feedforward neural networks* by Xavier Glorot and Yoshua Bengio  <http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf> |
| [6] | *ADAM: A method for stochastic optimization* byDiederik P. Kingma and Jimmy Lei Ba  [*https://arxiv.org/pdf/1412.6980.pdf*](https://arxiv.org/pdf/1412.6980.pdf) |
| [7] | *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift* by Sergey Ioffe and Christian Szegedy  <https://arxiv.org/pdf/1502.03167.pdf> |