Multi Task Learner Convolutional Neural Network

Emanuele Alessi (1486470)

Gianmarco Forcella (1725967)

Summary

[1. Introduction 3](#_Toc10579641)

[2. Theoretical overview 4](#_Toc10579642)

[2.1 Convolutional Neural Networks 4](#_Toc10579643)

[2.2 Siamese Networks 4](#_Toc10579644)

[2.3 TensorFlow 6](#_Toc10579645)

[3. Project setup 7](#_Toc10579646)

[3.1 The CFPW Dataset 7](#_Toc10579647)

[4. Data Pre-processing 9](#_Toc10579648)

[5. Implementation of a Siamese Neural Network 11](#_Toc10579649)

[5.1 The Weight initialization problem 12](#_Toc10579650)

[5.2 Minimizing loss 13](#_Toc10579651)

[5.3 Training and accuracy testing 13](#_Toc10579652)

[5.4 Prediction 14](#_Toc10579653)

[6. Model Evaluation 15](#_Toc10579654)

[6.1 Test Run 15](#_Toc10579655)

[6.2 Real Test Example 16](#_Toc10579656)

[6.3 Real Test Example – Execution 18](#_Toc10579657)

[7. Conclusions 20](#_Toc10579658)

[Appendix A – The Siamese Network class 21](#_Toc10579659)

[Appendix B – Creation of the Model 24](#_Toc10579660)

[Appendix C – Evaluation Process 25](#_Toc10579661)

[Appendix D – Accuracy Calculation 26](#_Toc10579662)

[Appendix E – Training Dataset Generation 27](#_Toc10579663)

[Appendix F – One Shot Testing Dataset Generation 27](#_Toc10579664)

[Appendix G – Threshold values 28](#_Toc10579665)

[References 31](#_Toc10579666)

# 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. Otherwhise, 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 MultiTask Convolutional Neural Network Learner, applied to Computer Vision by recognizing the following values:

* Whether the subject wears glasses or not
* Whether he/she has facial hairs
* 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.

Why is all this important? As an example, let’s assume that we want to process a 48x48x3 (48x48 with RGB color scheme) image with a Multilayer Neural Network, with Sigmoid as its activation function: this would mean having, just for a single neuron in the first hidden layer, about 6912 weights (not to mention the problem of the Vanishing / Exploding Gradient)! 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](https://en.wikipedia.org/wiki/Regularization_(mathematics)) induced by requiring an algorithm to perform well on a related task can be superior to regularization that prevents [overfitting](https://en.wikipedia.org/wiki/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.

# Project setup

Let us know enlist the technologies that we used while working on this project.

* **Python 3.x**
* **PyCharm as our IDE**
* **NumPy**
* **PIL**
* **Tqdm for pretty printing progress bars**
* **TensorFlow**
* **The** [**MAFL**](https://github.com/zhzhanp/TCDCN-face-alignment/tree/master/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).

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

EMANUELE!

# 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 594 images for the training process
* 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 RGB, through the PIL library;
* Each image is then resized to 105x105 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.

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

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

|  |  |  |
| --- | --- | --- |
| 400 x 499 | Resize | 105 x 105 |

Figure 5 An example of how image resizing is done (Jessica Alba, id 225)

# Implementation of a Siamese Neural Network

Our implementation takes free inspiration from [this](https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf) paper.

The first thing we do is, of course, initialize the class: we decided to build it in a not really pythonic way to have a better reuse in Jupyter Notebooks. The full class can be seen in [Appendix](#_Appendix_A_–) A.

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 105x105x3 NumPy Array (because we load it with a 3 color channels), cast it to a simple array and returning it, alongside with its size.

After that, we define the core of our MultiTask Learner Neural Network, as can be seen in [Appendix](#_Appendix_B_–) B.

Through TensorFlow (and with the contribution of Keras),

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.

The list of the layers used 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 units 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.

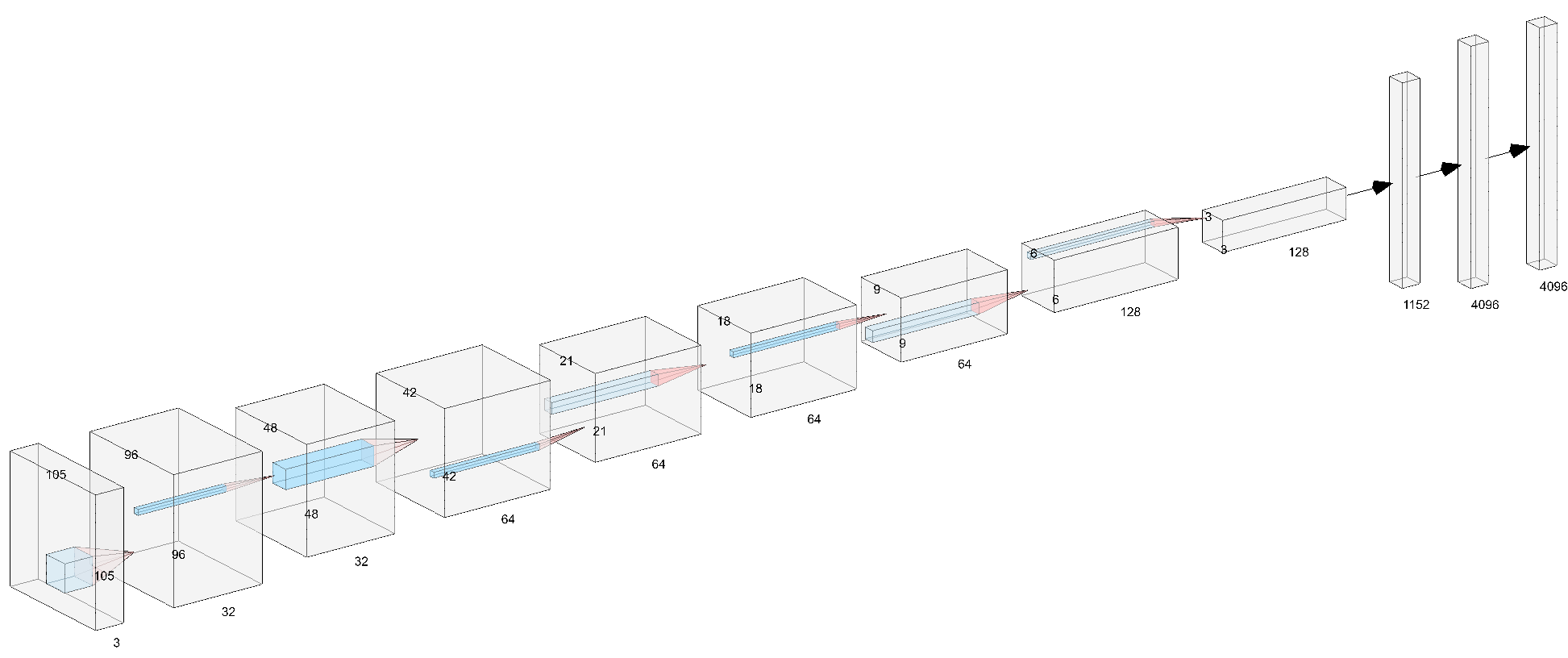


Figure 6 3D representation of our Siamese Network

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’ biases 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.

## Minimizing loss

In order to minimize the loss that each layer produces at any epoch, we decided to not apply a standard Optimizer but to use [**Adam’s**](https://arxiv.org/pdf/1412.6980.pdf). Adam is an optimization algorithm that can be used instead of the stochastic gradient descent procedure to update network weights.

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? At its core, Adam takes inspiration from two other extensions of the stochastic gradient descent, that are, **AdaGrad** and **RMSProp** (both maintain a per-parameter learning rate, while RMSProp also adapts that value to the average of the magnitude of the gradients for the weights) but, in its calculations, it also considers the uncentered variance (i.e. meaning it doesn’t subtract the mean during variance calculation): this means that the algorithm will calculate an exponential *moving average* of the gradient and the squared gradient, while keeping two other parameters (namely, beta1 and beta2) that control the decay rates of these moving averages.

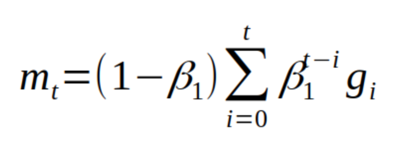


Figure 7 the moving average calculated with Adam formulation

The loss function, instead, is defined through the **binary cross entropy** formula, which is the following:

****

Figure 8 The Binary Cross Entropy Formula

# References

Koch G., Zemel R., Salakhutdinov R. – Siamese Neural network for One-Shot Image Recognition

Sun Y., Wang X., Tang X. – Deep Learning Face Representation by Joint Identification-Verification

Vinyals O., Blundell C., Lillicrap T., Kavukcuoglu K., Wierstra D. – Matching Networks for One Shot Learning

Schroff F., Kalenichenko D., Philbin J. – Facenet: A Unified Embedding for Face Recognition and Clustering

Ahmed E., Jones M., Marks T.K. – An Improved Deep Learning Architecture for Person Re-Identification

Chopra S., Hadsell R., LeCun Y. – Learning a Similarity Metric Discriminatively, with Application to Face Verification

Qi Y., Song Y., Zhang H., Liu J. – Sketch-Based Image Retrieval via Siamese Convolutional Neural Network