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.

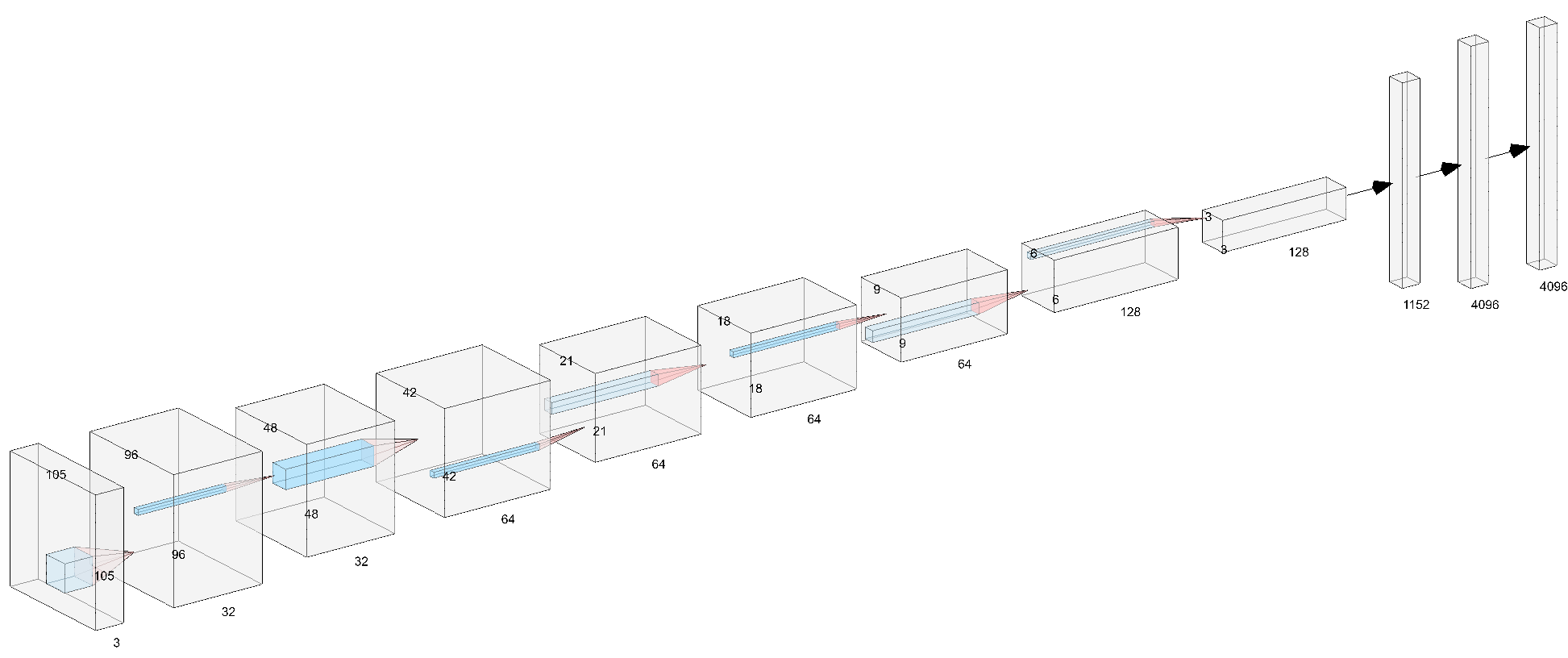


Figure 6 3D representation of our Siamese Network

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.

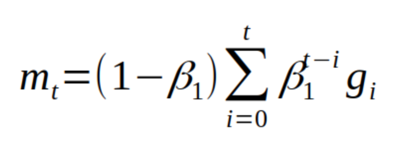


Figure 7 the moving average calculated with Adam formulation

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

****

Figure 8 The Binary Cross Entropy Formula

## Training and accuracy testing

In order to create batches for training and testing our Multi Learner 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 images 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.

## 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 ORA SONO PIGRO PER CERCARLA (NON ME LA RICORDO ORA)

Dovremmo lasciarlo????

# 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())

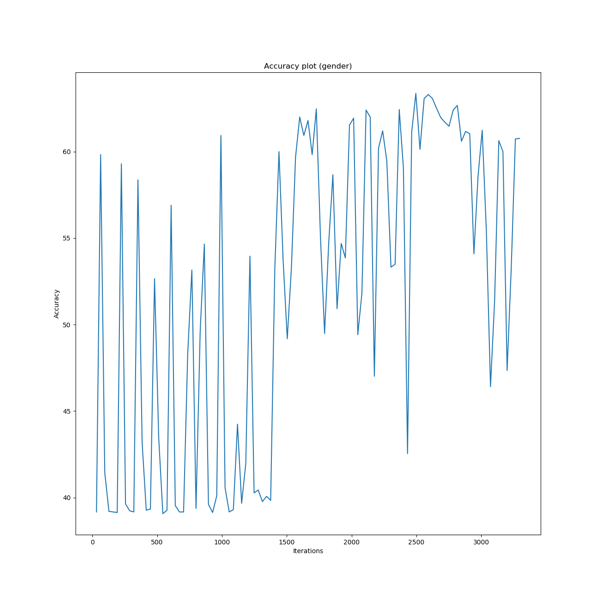
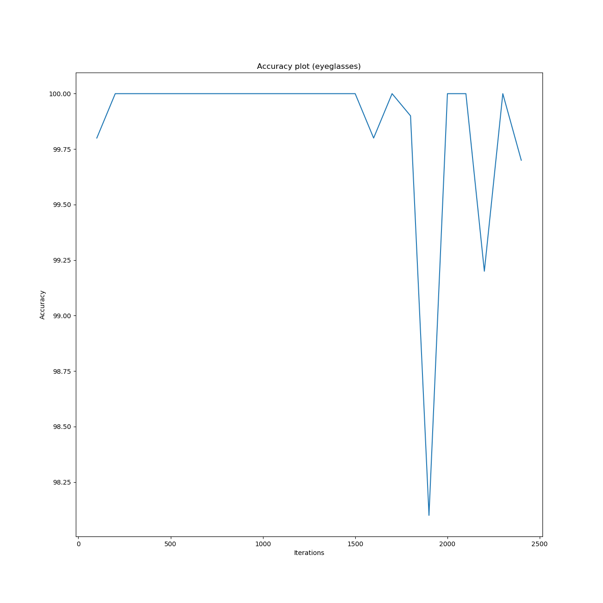
function. The specified input to .run() allows the Graph to have all its variables initialized, while the .run() method performs the specified action in input.

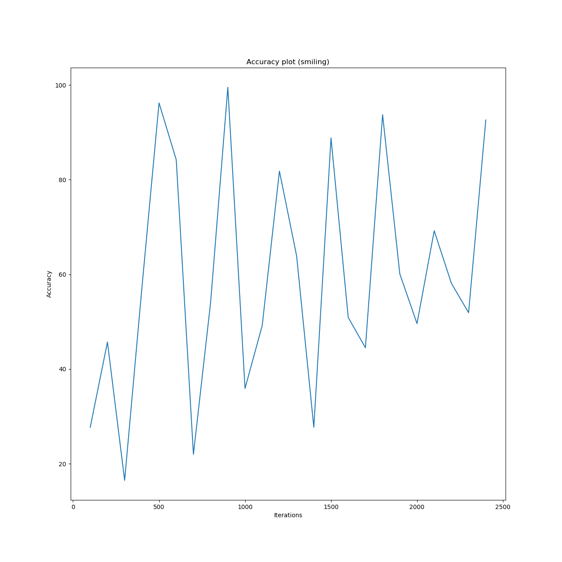
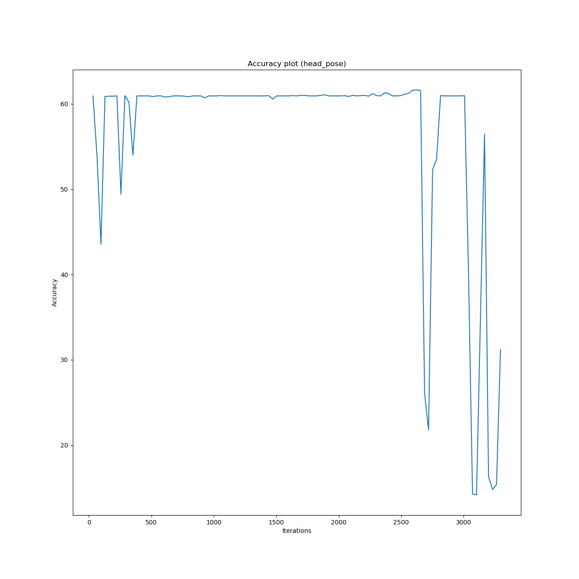
We then start the computation of the Network, by iterating to perform a good training. To get the dataset for training, we decided to generate a function (which can be read in [Appendix E](#_Appendix_E_–)) that will load, each time it receives a path, the image and will also augment it for the network.

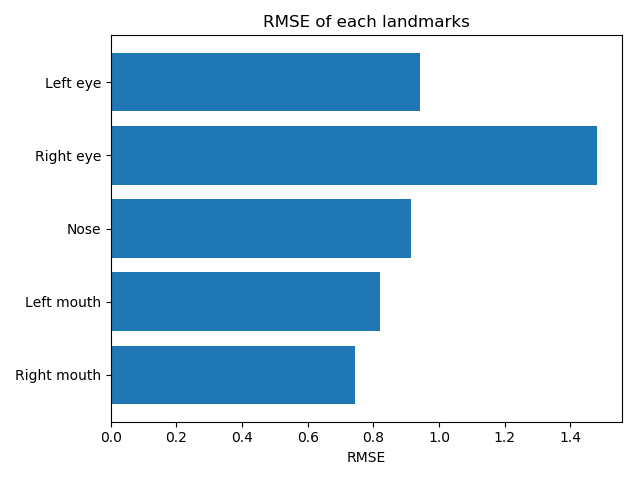
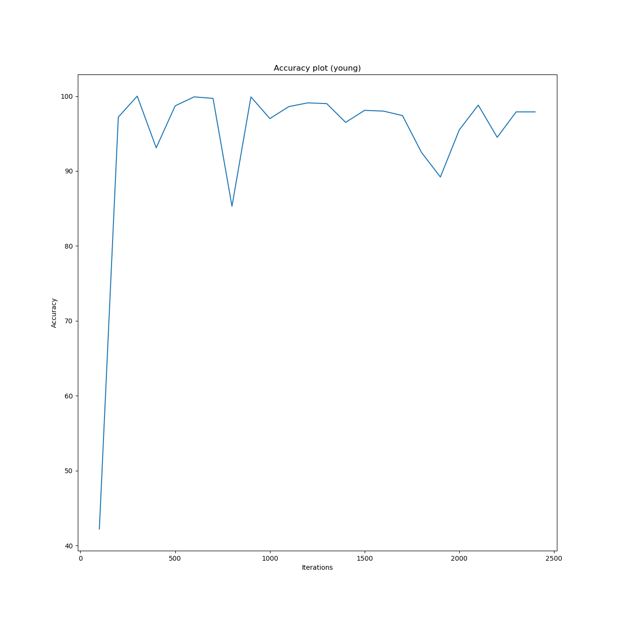
Every 100 iterations, there is also a “check step”: at that point, the algorithm will pick up a new portion of the dataset (Appendix F), so that it can perform some testing.

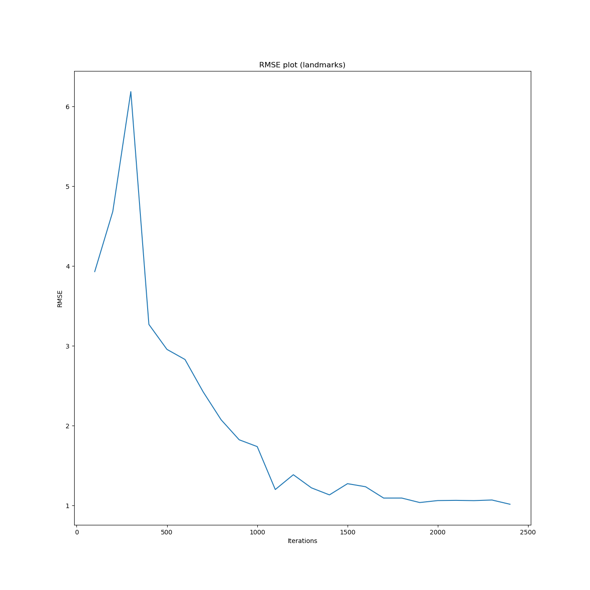
## Test Run

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





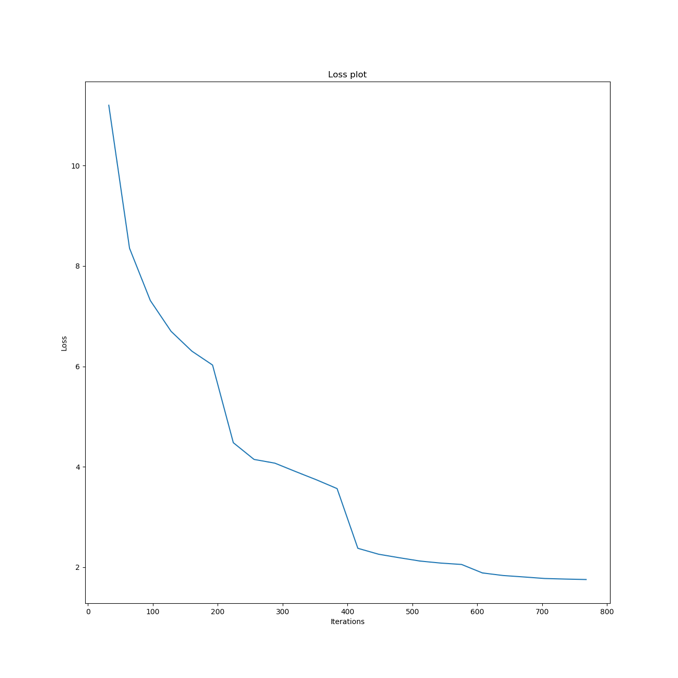
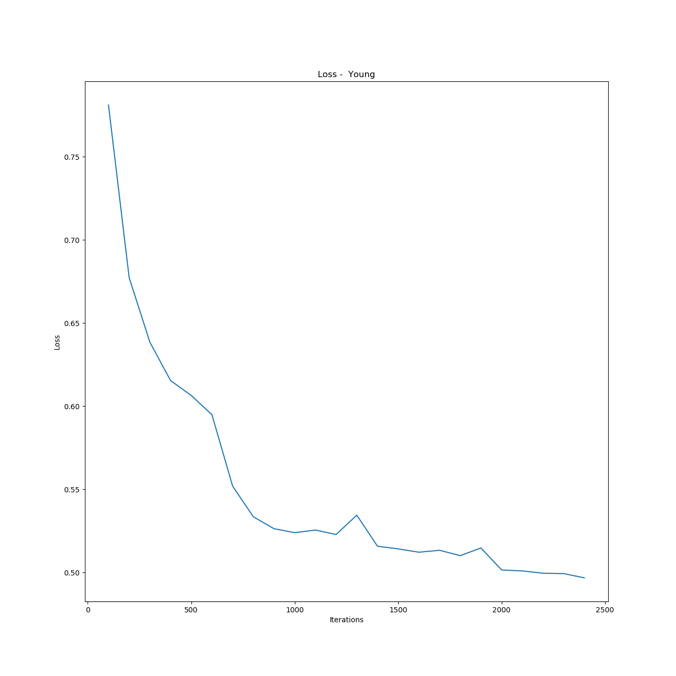
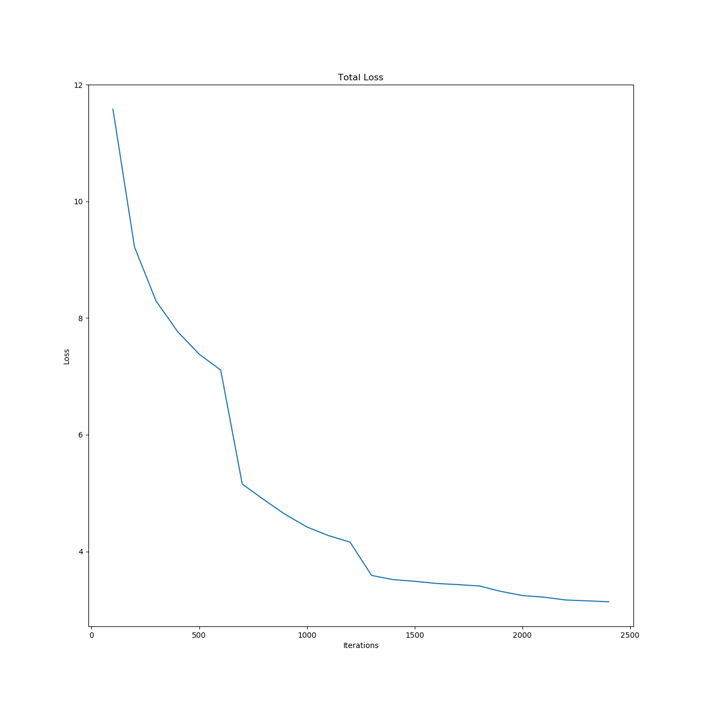
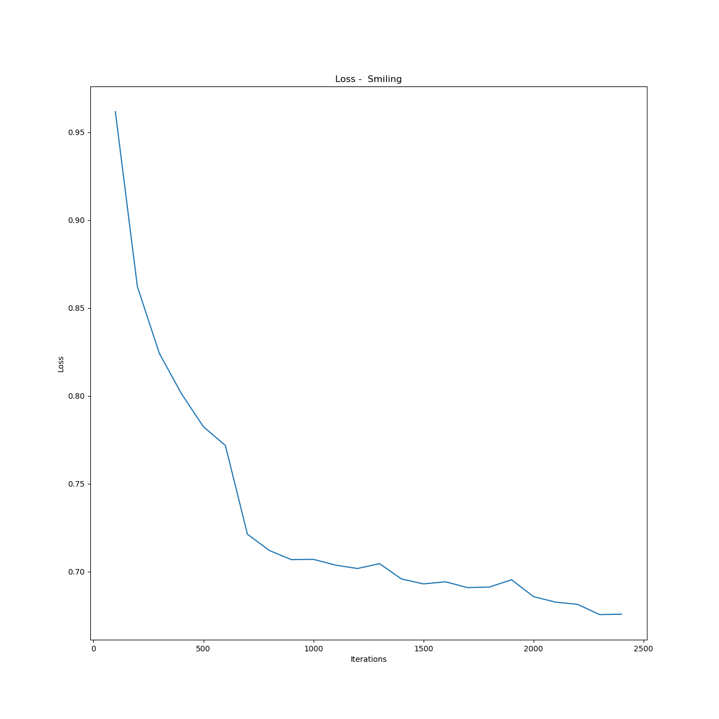
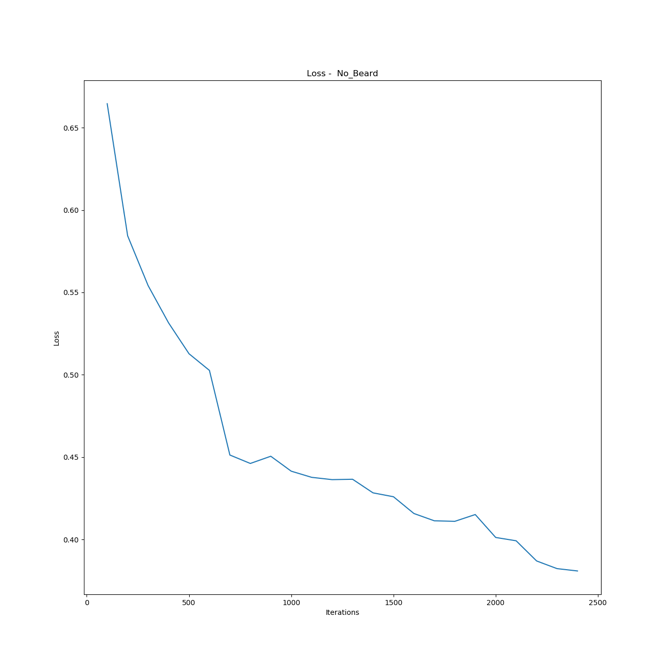
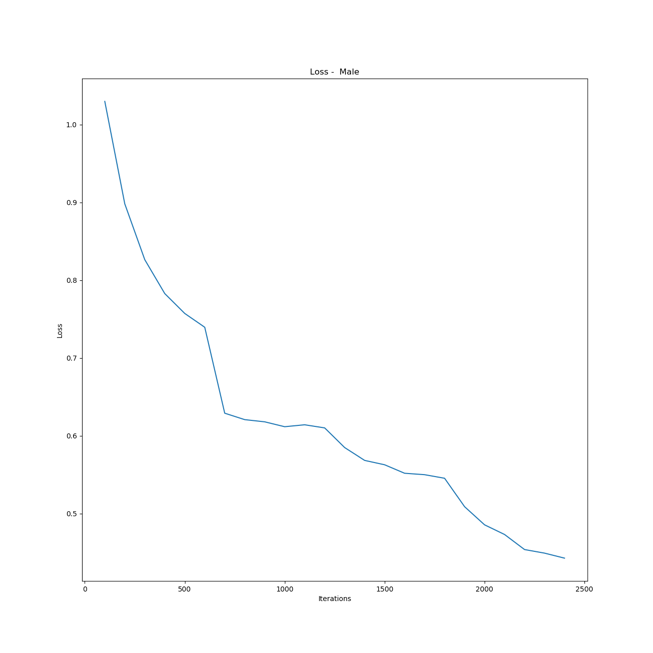
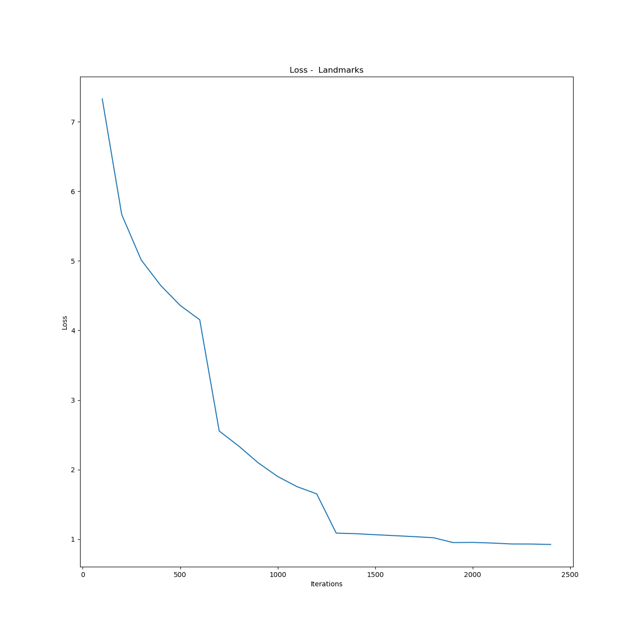
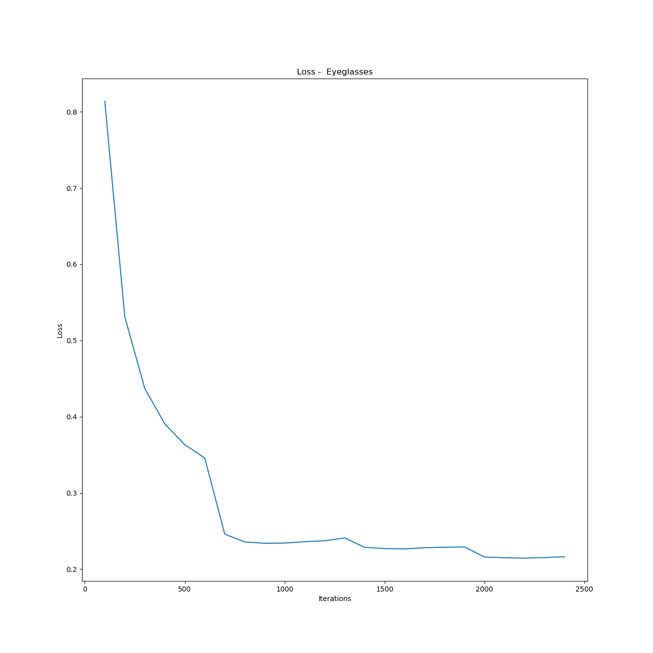




As it is possible to see from the figures, the accuracy that the MultiTask Learner Network can provide, in classification tasks, is immediately very high and it can reach in just a few iterations a value between the 85-95%. This may happen because of the difficulty of the task.

For the RMSE on landmarks, it is possible to see that the Network is immediately able to compute, at first, a very low RMSE (about 3.9) and, after 4 epochs (We remind that, in this case, an epoch consists of 594 iterations!), get to 0.98.

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



# 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 1.3, 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 0.4.

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 at least 8hrs to just complete.

# Appendix A – The Siamese Network class

1. **import** os
2. **import** numpy as np
3. **import** tqdm
4. **import** tensorflow as tf
5. **import** data\_preprocessing as dp
6. **from** cnn **import** siamese\_network
7. **import** plot\_generator as pg
9. **class** SiameseNetwork:
11. **def** \_\_init\_\_(self):
12. self.\_\_DATA\_DIR = 'cfp-dataset/Data/Images'
13. self.\_\_TMP\_DIR = 'tmp'
15. self.\_\_BATCH\_SIZE = 32
16. self.\_\_ITERATIONS = 3000
18. **if** **not** os.path.exists(self.\_\_TMP\_DIR):
19. os.makedirs(self.\_\_TMP\_DIR)
21. self.\_\_GLOBAL\_ITER = dp.global\_iteration(self.\_\_TMP\_DIR + '/iteration.txt')
23. **print**('Global iteration:', self.\_\_GLOBAL\_ITER)
25. self.\_\_train\_set = []
26. self.\_\_test\_set = []
28. self.\_\_shape = (105, 105, 3)
30. self.\_\_graph = tf.Graph()
32. with self.\_\_graph.as\_default():
33. self.\_\_img\_1 = tf.placeholder(tf.float32, shape=[None, self.\_\_shape[0], self.\_\_shape[1], self.\_\_shape[2]])
34. self.\_\_img\_2 = tf.placeholder(tf.float32, shape=[None, self.\_\_shape[0], self.\_\_shape[1], self.\_\_shape[2]])
35. self.\_\_flags = tf.placeholder(tf.float32, shape=[None])
37. self.\_\_embeddings\_1 = siamese\_network(self.\_\_img\_1, reuse\_variables=False)
38. self.\_\_embeddings\_2 = siamese\_network(self.\_\_img\_2, reuse\_variables=True)
40. self.\_\_distance = tf.abs(tf.subtract(self.\_\_embeddings\_1, self.\_\_embeddings\_2))
42. self.\_\_scores = tf.contrib.layers.fully\_connected(inputs=self.\_\_distance, num\_outputs=1, activation\_fn=tf.nn.sigmoid,
43. biases\_initializer=tf.truncated\_normal\_initializer(mean=0.5,
44. stddev=0.01))
46. self.\_\_losses = tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=self.\_\_flags,
47. logits=tf.reshape(self.\_\_scores, shape=[self.\_\_BATCH\_SIZE]))
48. self.\_\_loss = tf.reduce\_mean(self.\_\_losses)
50. self.\_\_optimizer = tf.train.AdamOptimizer(learning\_rate=0.00005)
51. # optimizer = tf.train.MomentumOptimizer(learning\_rate=0.0001, momentum=0.95, use\_nesterov=True)
52. self.\_\_train\_op = self.\_\_optimizer.minimize(self.\_\_loss)
54. self.\_\_prediction = tf.cast(tf.argmax(self.\_\_scores, axis=0), dtype=tf.int32)
56. self.\_\_saver = tf.train.Saver()
58. init = tf.global\_variables\_initializer()
60. ### SESSION ###
61. self.\_\_session = tf.Session(graph=self.\_\_graph)
63. # We must initialize all variables before we use them.
64. init.run(session=self.\_\_session)
66. # reload the model if it exists and continue to train
67. **try**:
68. self.\_\_saver.restore(self.\_\_session, os.path.join(self.\_\_TMP\_DIR, 'model.ckpt'))
69. **print**('Model restored')
70. **except**:
71. **print**('Model initialized')
73. **def** train(self, epochs=1):
74. **if** self.\_\_train\_set **and** self.\_\_test\_set:
75. **pass**
76. **else**:
77. self.\_\_train\_set, self.\_\_test\_set = dp.load\_dataset(self.\_\_TMP\_DIR, self.\_\_DATA\_DIR)
79. # Open a writer to write summaries.
80. self.\_\_writer = tf.summary.FileWriter(self.\_\_TMP\_DIR, self.\_\_session.graph)
82. average\_loss = 0
84. **for** step **in** tqdm.tqdm(range(self.\_\_ITERATIONS \* epochs), desc='Training Siamese Network'):
85. batch, label = dp.get\_batch(self.\_\_train\_set, self.\_\_BATCH\_SIZE)
87. pair\_1 = np.array([b[0] **for** b **in** batch])
88. pair\_2 = np.array([b[1] **for** b **in** batch])
90. # Define metadata variable.
91. run\_metadata = tf.RunMetadata()
93. \_, l = self.\_\_session.run([self.\_\_train\_op, self.\_\_loss], feed\_dict={self.\_\_img\_1: pair\_1, self.\_\_img\_2: pair\_2, self.\_\_flags: label},
94. run\_metadata=run\_metadata)
96. average\_loss += l
98. # print loss and accuracy on test set every 500 steps
99. **if** (step % 500 == 0 **and** step > 0) **or** (step == (self.\_\_ITERATIONS - 1)):
100. correct = 0
101. k = len(self.\_\_test\_set)
102. **for** \_ **in** range(k):
103. test, label = dp.get\_one\_shot\_test(self.\_\_test\_set)
104. pair\_1 = np.array([b[0] **for** b **in** test])
105. pair\_2 = np.array([b[1] **for** b **in** test])
107. run\_metadata = tf.RunMetadata()
109. pred = self.\_\_session.run(self.\_\_prediction, feed\_dict={self.\_\_img\_1: pair\_1, self.\_\_img\_2: pair\_2}, run\_metadata=run\_metadata)
110. **if** pred[0] == 0:
111. correct += 1
113. **print**('Loss:', str(average\_loss / step), '\tAccuracy:', correct / k)
115. with open(self.\_\_TMP\_DIR + '/log.txt', 'a', encoding='utf8') as f:
116. f.write(str(correct / k) + ' ' + str(average\_loss / step) + '\n')
118. **if** step == (self.\_\_ITERATIONS - 1):
119. self.\_\_writer.add\_run\_metadata(run\_metadata, 'step%d' % step, global\_step=self.\_\_GLOBAL\_ITER + step + 1)
121. self.\_\_saver.save(self.\_\_session, os.path.join(self.\_\_TMP\_DIR, 'model.ckpt'))
122. dp.global\_iteration(self.\_\_TMP\_DIR + '/iteration.txt', update=self.\_\_GLOBAL\_ITER + step + 1)
124. pg.generate\_accuracy\_plot(self.\_\_TMP\_DIR + '/')
125. pg.generate\_loss\_plot(self.\_\_TMP\_DIR + '/')
127. self.\_\_writer.close()
129. **def** predict(self, imgs1, imgs2):
130. run\_metadata = tf.RunMetadata()
131. similarity\_scores = self.\_\_session.run(self.\_\_scores, feed\_dict={self.\_\_img\_1: imgs1, self.\_\_img\_2: imgs2}, run\_metadata=run\_metadata)
132. **return** similarity\_scores

# Appendix B – Creation of the Model

1. **import** tensorflow as tf
3. # MODEL #
5. **def** siamese\_network(img, reuse\_variables=False):
7. with tf.name\_scope('siamese'):
9. with tf.variable\_scope('conv1') as scope:
10. layer = tf.contrib.layers.conv2d(inputs=img, num\_outputs=32, kernel\_size=[10, 10], padding='VALID', activation\_fn=tf.nn.relu,
11. biases\_initializer=tf.truncated\_normal\_initializer(mean=0.5, stddev=0.01), scope=scope,
12. reuse=reuse\_variables)
13. layer = tf.contrib.layers.max\_pool2d(layer, kernel\_size=[2, 2], stride=2, padding='VALID')
15. with tf.variable\_scope('conv2') as scope:
16. layer = tf.contrib.layers.conv2d(inputs=layer, num\_outputs=64, kernel\_size=[7, 7], padding='VALID', activation\_fn=tf.nn.relu,
17. biases\_initializer=tf.truncated\_normal\_initializer(mean=0.5, stddev=0.01), scope=scope,
18. reuse=reuse\_variables)
19. layer = tf.contrib.layers.max\_pool2d(layer, kernel\_size=[2, 2], stride=2, padding='VALID')
21. with tf.variable\_scope('conv3') as scope:
22. layer = tf.contrib.layers.conv2d(inputs=layer, num\_outputs=64, kernel\_size=[4, 4], padding='VALID', activation\_fn=tf.nn.relu,
23. biases\_initializer=tf.truncated\_normal\_initializer(mean=0.5, stddev=0.01), scope=scope,
24. reuse=reuse\_variables)
25. layer = tf.contrib.layers.max\_pool2d(layer, kernel\_size=[2, 2], stride=2, padding='VALID')
27. with tf.variable\_scope('conv4') as scope:
28. layer = tf.contrib.layers.conv2d(inputs=layer, num\_outputs=128, kernel\_size=[4, 4], padding='VALID', activation\_fn=tf.nn.relu,
29. biases\_initializer=tf.truncated\_normal\_initializer(mean=0.5, stddev=0.01), scope=scope,
30. reuse=reuse\_variables)
31. layer = tf.contrib.layers.max\_pool2d(layer, kernel\_size=[2, 2], stride=2, padding='VALID')
33. with tf.variable\_scope('flatten') as scope:
34. layer = tf.contrib.layers.flatten(inputs=layer)
36. with tf.variable\_scope('fc') as scope:
37. layer = tf.contrib.layers.fully\_connected(inputs=layer, num\_outputs=4096, activation\_fn=tf.nn.sigmoid,
38. biases\_initializer=tf.truncated\_normal\_initializer(mean=0.5, stddev=0.01),
39. scope=scope, reuse=reuse\_variables)
41. **return** layer

# Appendix C – Evaluation Process

1. with self.\_\_graph.as\_default():
2. self.\_\_img\_1 = tf.placeholder(tf.float32, shape=[None, self.\_\_shape[0], self.\_\_shape[1], self.\_\_shape[2]])
3. self.\_\_img\_2 = tf.placeholder(tf.float32, shape=[None, self.\_\_shape[0], self.\_\_shape[1], self.\_\_shape[2]])
4. self.\_\_flags = tf.placeholder(tf.float32, shape=[None])
6. self.\_\_embeddings\_1 = siamese\_network(self.\_\_img\_1, reuse\_variables=False)
7. self.\_\_embeddings\_2 = siamese\_network(self.\_\_img\_2, reuse\_variables=True)
9. self.\_\_distance = tf.abs(tf.subtract(self.\_\_embeddings\_1, self.\_\_embeddings\_2))
11. self.\_\_scores = tf.contrib.layers.fully\_connected(inputs=self.\_\_distance, num\_outputs=1, activation\_fn=tf.nn.sigmoid,
12. biases\_initializer=tf.truncated\_normal\_initializer(mean=0.5,
13. stddev=0.01))
15. self.\_\_losses = tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=self.\_\_flags,
16. logits=tf.reshape(self.\_\_scores, shape=[self.\_\_BATCH\_SIZE]))
17. self.\_\_loss = tf.reduce\_mean(self.\_\_losses)
19. self.\_\_optimizer = tf.train.AdamOptimizer(learning\_rate=0.00005)
20. # optimizer = tf.train.MomentumOptimizer(learning\_rate=0.0001, momentum=0.95, use\_nesterov=True)
21. self.\_\_train\_op = self.\_\_optimizer.minimize(self.\_\_loss)
23. self.\_\_prediction = tf.cast(tf.argmax(self.\_\_scores, axis=0), dtype=tf.int32)
25. self.\_\_saver = tf.train.Saver()

# Appendix D – Accuracy Calculation

1. **def** train(self, epochs=1):
2. **if** self.\_\_train\_set **and** self.\_\_test\_set:
3. **pass**
4. **else**:
5. self.\_\_train\_set, self.\_\_test\_set = dp.load\_dataset(self.\_\_TMP\_DIR, self.\_\_DATA\_DIR)
7. # Open a writer to write summaries.
8. self.\_\_writer = tf.summary.FileWriter(self.\_\_TMP\_DIR, self.\_\_session.graph)
10. average\_loss = 0
12. **for** step **in** tqdm.tqdm(range(self.\_\_ITERATIONS \* epochs), desc='Training Siamese Network'):
13. batch, label = dp.get\_batch(self.\_\_train\_set, self.\_\_BATCH\_SIZE)
15. pair\_1 = np.array([b[0] **for** b **in** batch])
16. pair\_2 = np.array([b[1] **for** b **in** batch])
18. # Define metadata variable.
19. run\_metadata = tf.RunMetadata()
21. \_, l = self.\_\_session.run([self.\_\_train\_op, self.\_\_loss], feed\_dict={self.\_\_img\_1: pair\_1, self.\_\_img\_2: pair\_2, self.\_\_flags: label},
22. run\_metadata=run\_metadata)
24. average\_loss += l
26. # print loss and accuracy on test set every 500 steps
27. **if** (step % 500 == 0 **and** step > 0) **or** (step == (self.\_\_ITERATIONS - 1)):
28. correct = 0
29. k = len(self.\_\_test\_set)
30. **for** \_ **in** range(k):
31. test, label = dp.get\_one\_shot\_test(self.\_\_test\_set)
32. pair\_1 = np.array([b[0] **for** b **in** test])
33. pair\_2 = np.array([b[1] **for** b **in** test])
35. run\_metadata = tf.RunMetadata()
37. pred = self.\_\_session.run(self.\_\_prediction, feed\_dict={self.\_\_img\_1: pair\_1, self.\_\_img\_2: pair\_2}, run\_metadata=run\_metadata)
38. **if** pred[0] == 0:
39. correct += 1
41. **print**('Loss:', str(average\_loss / step), '\tAccuracy:', correct / k)
43. with open(self.\_\_TMP\_DIR + '/log.txt', 'a', encoding='utf8') as f:
44. f.write(str(correct / k) + ' ' + str(average\_loss / step) + '\n')
46. **if** step == (self.\_\_ITERATIONS - 1):
47. self.\_\_writer.add\_run\_metadata(run\_metadata, 'step%d' % step, global\_step=self.\_\_GLOBAL\_ITER + step + 1)
49. self.\_\_saver.save(self.\_\_session, os.path.join(self.\_\_TMP\_DIR, 'model.ckpt'))
50. dp.global\_iteration(self.\_\_TMP\_DIR + '/iteration.txt', update=self.\_\_GLOBAL\_ITER + step + 1)
52. pg.generate\_accuracy\_plot(self.\_\_TMP\_DIR + '/')
53. pg.generate\_loss\_plot(self.\_\_TMP\_DIR + '/')
55. self.\_\_writer.close()

# Appendix E – Training Dataset Generation

1. **def** get\_batch(train\_set, batch\_size):
2. cat = np.random.choice(list(range(len(train\_set))), size=batch\_size, replace=False)
3. #print(cat)
4. label = np.zeros(batch\_size)
5. # If the inputs are from the same class, then the value of label is 1, otherwise label is 0
6. label[:batch\_size // 2] = 1
7. batch = []
8. **for** i **in** range(batch\_size // 2):
9. category = cat[i]
10. random\_index = np.random.randint(0, len(train\_set[category]))
11. img\_1 = train\_set[category][random\_index]
12. random\_index = np.random.randint(0, len(train\_set[category]))
13. img\_2 = train\_set[category][random\_index]
14. batch.append((img\_1, img\_2))
15. **for** i **in** range(batch\_size // 2, batch\_size):
16. category\_1 = cat[i]
17. random\_index = np.random.randint(0, len(train\_set[category\_1]))
18. img\_1 = train\_set[category\_1][random\_index]
19. category\_2 = (category\_1 + np.random.randint(1, len(train\_set))) % len(train\_set)
20. img\_2 = train\_set[category\_2][random\_index]
21. batch.append((img\_1, img\_2))
22. **return** batch, label

# Appendix F – One Shot Testing Dataset Generation

1. **def** get\_one\_shot\_test(test\_set, n\_examples=10):
2. # Questa funzione ritorna una lista di 10 coppie, dove la prima � con la medesima persona, le altre sono persone diverse
3. n\_classes = len(test\_set)
4. # n\_examples = len(test\_set[0])
5. cat = np.random.choice(list(range(n\_classes)), size=n\_classes, replace=False)
6. random\_indexes = np.random.randint(0, len(test\_set[0]), size=n\_examples)
7. true\_cat = cat[0]
8. ex1, ex2 = np.random.choice(len(test\_set[0]), replace=False, size=2)
9. test = []
10. label = np.zeros(n\_classes)
11. img\_1 = test\_set[true\_cat][ex1]
12. **for** k, random\_index **in** enumerate(random\_indexes):
13. **if** k == 0:
14. img\_2 = test\_set[cat[k]][ex2]
15. **else**:
16. img\_2 = test\_set[cat[k]][random\_index]
17. test.append((img\_1, img\_2))
18. label[0] = 1
19. **return** test, label

# Appendix G – Threshold values

1. 5e-05
2. 584 138617 916 609883
3. 0.3893333333333333
4. 0.1851930527722111
5. 0.6106666666666667
6. 0.8148069472277889
8. 7e-05
9. 558 129523 942 618977
10. 0.372
11. 0.17304342017368068
12. 0.628
13. 0.8269565798263193
15. 9e-05
16. 543 122758 957 625742
17. 0.362
18. 0.1640053440213761
19. 0.638
20. 0.8359946559786239
22. 6e-07
23. 936 278201 564 470299
24. 0.624
25. 0.3716780227120908
26. 0.376
27. 0.6283219772879092
29. 5e-08
30. 1081 361678 419 386822
31. 0.7206666666666667
32. 0.48320374081496326
33. 0.2793333333333333
34. 0.5167962591850367
36. 7e-08
37. 1066 350497 434 398003
38. 0.7106666666666667
39. 0.46826586506346024
40. 0.28933333333333333
41. 0.5317341349365398
43. 9e-08
44. 1053 342245 447 406255
45. 0.702
46. 0.45724114896459583
47. 0.298
48. 0.5427588510354041
50. 1e-07
51. 1045 338693 455 409807
52. 0.6966666666666667
53. 0.4524956579826319
54. 0.30333333333333334
55. 0.5475043420173681
57. 7e-06
58. 733 197443 767 551057
59. 0.4886666666666667
60. 0.26378490313961256
61. 0.5113333333333333
62. 0.7362150968603874
64. 9e-06
65. 718 189560 782 558940
66. 0.4786666666666667
67. 0.25325317301269207
68. 0.5213333333333333
69. 0.746746826987308
71. 1e-05
72. 712 186319 788 562181
73. 0.4746666666666667
74. 0.24892317969271877
75. 0.5253333333333333
76. 0.7510768203072812
78. 3e-05
79. 624 153278 876 595222
80. 0.416
81. 0.2047802271209085
82. 0.584
83. 0.7952197728790915
85. 9e-07
86. 909 264519 591 483981
87. 0.606
88. 0.3533987975951904
89. 0.394
90. 0.6466012024048097
92. 1e-06
93. 902 260935 598 487565
94. 0.6013333333333334
95. 0.3486105544422178
96. 0.39866666666666667
97. 0.6513894455577822
99. 3e-06
100. 803 224821 697 523679
101. 0.5353333333333333
102. 0.3003620574482298
103. 0.4646666666666667
104. 0.6996379425517703
106. 5e-06
107. 764 208208 736 540292
108. 0.5093333333333333
109. 0.27816700066800265
110. 0.49066666666666664
111. 0.7218329993319973
113. 7e-07
114. 925 272996 575 475504
115. 0.6166666666666667
116. 0.36472411489645956
117. 0.38333333333333336
118. 0.6352758851035404
120. 1e-08
121. 1163 413607 337 334893
122. 0.7753333333333333
123. 0.5525811623246493
124. 0.22466666666666665
125. 0.4474188376753507
127. 3e-08
128. 1113 378439 387 370061
129. 0.742
130. 0.5055965263861055
131. 0.258
132. 0.4944034736138945

# 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