Face Recognition through a Siamese Network

Emanuele Alessi (1486470)
Gianmarco Forcella (1725967)
Gabriel Radu Taranciuc (1693558)

Summary

1. Introduction *3*

2. Theoretical overview *4*

2.1 Convolutional Neural Networks 4

2.2 Siamese Networks 4

2.3 TensorFlow 6

3. Project setup *7*

3.1 The CFPW Dataset 7

4. Data Pre-processing *9*

5. Implementation of a Siamese Neural Network *11*

5.1 The Weight initialization problem 12

5.2 Minimizing loss 13

5.3 Training and accuracy testing 13

5.4 Prediction 14

6. Model Evaluation *15*

6.1 Test Run 15

6.2 Real Test Example 16

6.3 Real Test Example – Execution 18

7. Conclusions *20*

Appendix A – The Siamese Network class *21*

Appendix B – Creation of the Model *24*

Appendix C – Evaluation Process *25*

Appendix D – Accuracy Calculation *26*

Appendix E – Training Dataset Generation *27*

Appendix F – One Shot Testing Dataset Generation *27*

Appendix G – Threshold values *28*

References *31*

1. Introduction

Face Detection and Recognition are one of the most popular fields in Artificial Intelligence studies. But what's the difference between them? Many people still mistake them for one another.

- When somebody is talking about Face Detection, they are referring to a task in which the goal is to find faces in a given image (i.e.: is this a face / is there a face in this image?);
- On the other hand, Face Recognition is applied when it is necessary to identify a person in a given image (i.e.: who is this?).

But that is not where Face Recognition stops. In fact, it can be divided into sub-categories:

1. Identification: Given a face image, the objective is to match that file on a database – hence – identifying from this database whether that faces belongs to someone in it, and if so, who that face belongs to (most common mean of Face Recognition);
2. Verification: Given an image and an identity, confirm that the given face belongs to the given identity (typical authentication / authorization task).

In this paper, we faced the problem of performing a Face Recognition (via Verification) of a given image through a Siamese Network.
Why a Siamese Network? How did we get to it? And what is it?
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.

2. 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.

2.1 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.

2.2 Siamese Networks

Since sometimes image datasets can contain few training samples of the same photo, **one-shot learning** techniques may be the best option for the Face Recognition task: with this approach, as it may be easy to guess, the objective is to learn information about object categories from one or few training samples.

One of the most common "exploiters" of the one-shot learning technique is the **Siamese Network**, which was first reported in '90s research papers as a new model of Neural Network, in which there

are two or more twin nets that make predictions and then “merge” towards a guess: something similar to an ensemble method.

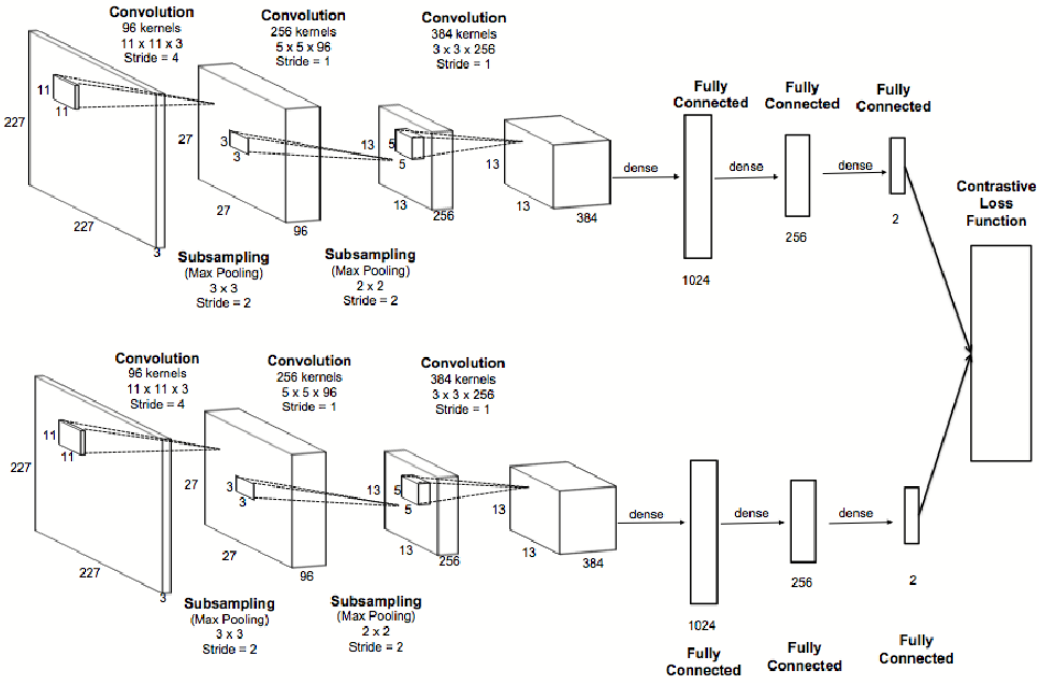


Figure 1 An example of a Siamese Network

Because the networks are twins, not only do they share the same architecture, they also share the very same weights.

The overall architecture of a Siamese Network, and its features, holds two key properties:

1. It ensures prediction consistency. Since the parameters between the networks are tied (and so are weights), there is a guarantee that two very similar images will have the same locations in the feature space, as per the activation function of the twin networks, since the calculations will always be the same;
2. Since the network is symmetric, no matter which network and image is first presented to, it will always calculate the same metrics (because of the above point).

But what about its predictive model? Essentially, the twin networks will compute n feature vectors from the input image (depending on how many pairs there are in the network) and then it will calculate a similarity value *d* amongst the two images. There are two possible outputs allowed:

- If *d* is small, then it is safe to say that the images are similar. That’s because the twin networks calculated very similar values through their neurons, so it is safe to assume that the images might be the same;
- On the contrary, if *d* is high, it is easily assumed that the images are not similar.

2.3 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.
Please refer to Chapter 5 to immediately see our implementation of the Convolutional Neural Network.

3. 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 CFPW dataset** (which is a collection of frontal and profile views of face images that belong to many celebrities).

To check out the full project, please refer to this GitHub repository, which also contains all the papers we took inspiration from for our work.

3.1 The CFPW Dataset

The **CFPW dataset**, as already stated, is a dataset that is a collection of frontal and profile views of face images that belong to many celebrities.
The dataset contains images of 500 celebrities, 14 images each (10 from a frontal point of view and 4 of their profiles): this means that, as a grand total, the dataset has 7000 samples.

The dataset is organized in the following way: there is a folder for each person, where each one has a unique integer ID (numbered from 001 to 500). Each one of these folders contains two separate folders for the frontal and the profile points of view.
For our purposes we decided to discard the part of the dataset that contains the profile images, so our working dataset contains 500 x 10 = 5000 samples.

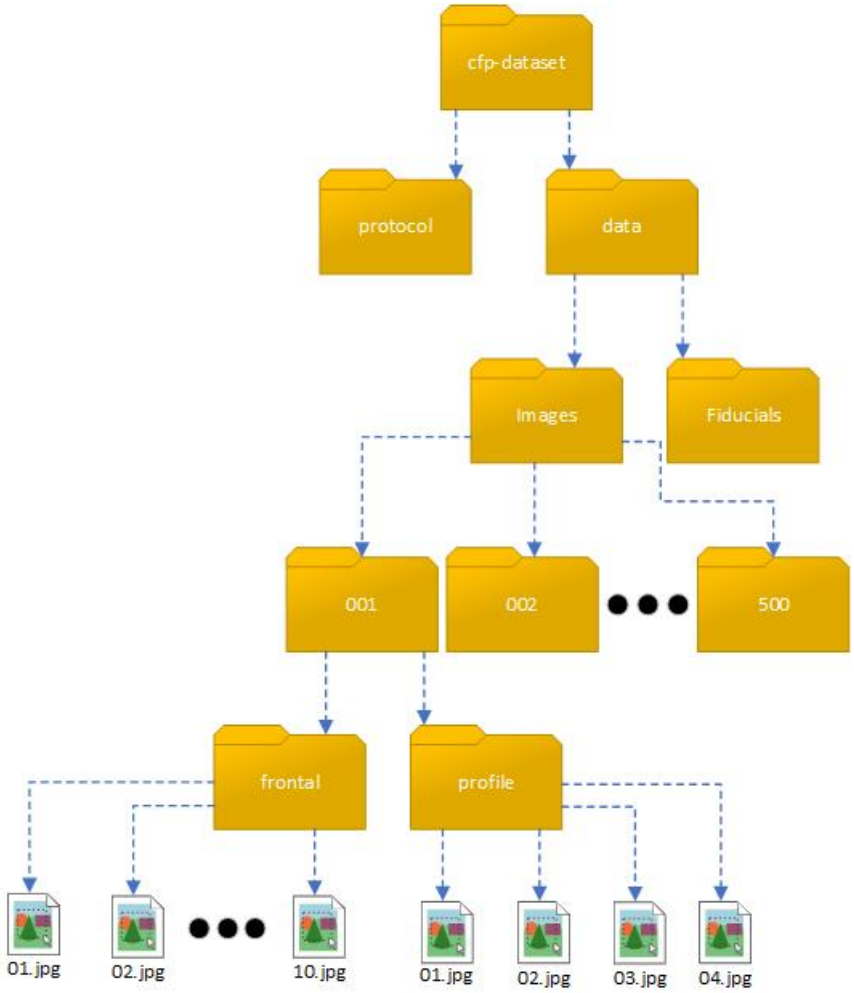


Figure 2 CFPW dataset structure

As stated above, we ignore the profile one. Finally, every frontal folder contains 10 photos of that person, taken in different lighting conditions, and with differing sized images.
As can be seen in the examples below, the images are already cropped.

Figure 3 An example of frontal images (Al Pacino, id 012)

4. 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 initializing two empty lists (one used for training the model, and the other one will be used for testing purposes) that will contain all the images in the dataset, grouped by person.
We decided to use 70% of the images of each person for training the model, while keeping the remaining 30% for the testing phase. Images used for training and testing are randomly picked from the 10 available for each person.

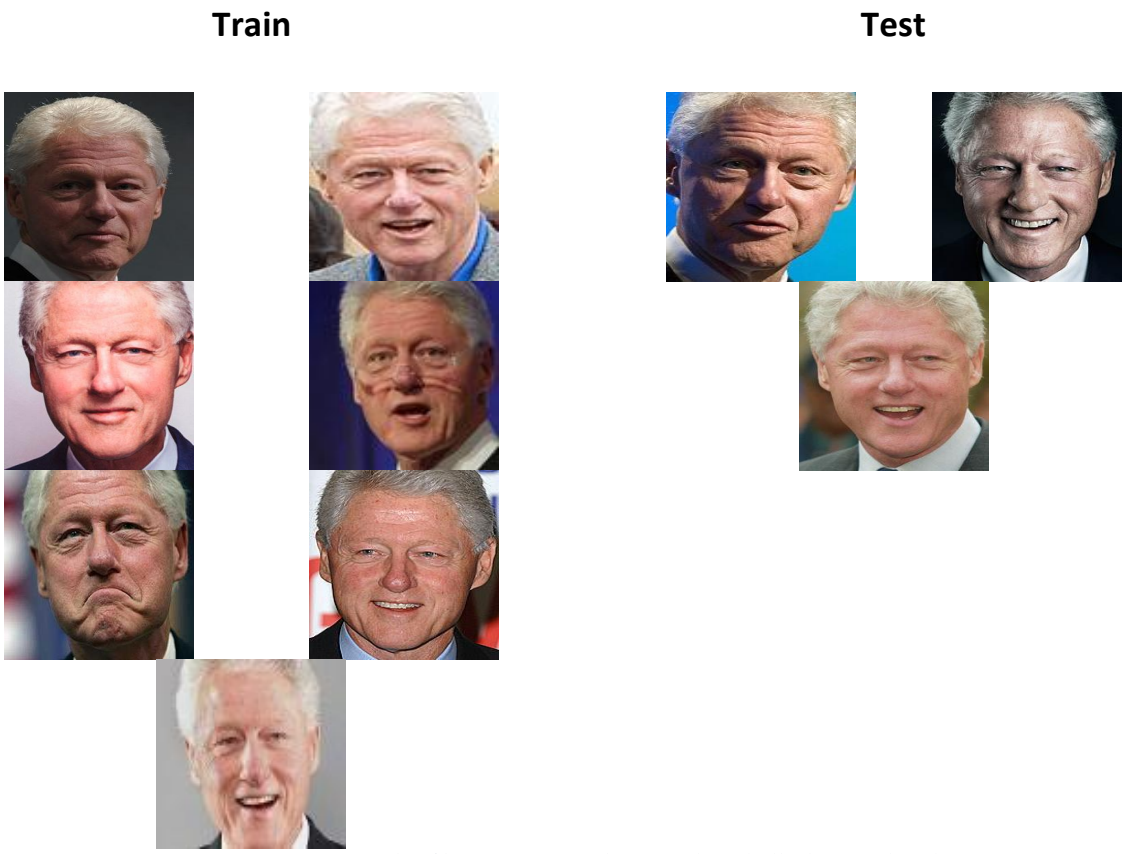


Figure 4 An example of how train-test splitting is done (Bill Clinton, id 067)

When loading each image, there are some steps done before saving it to our list:

- We load the image in RGB, through the PIL library;
- Each image is then resized to 105x105 using the LANCZOS resampling technique;
- Finally, it is converted into a NumPy array for use with TensorFlow.

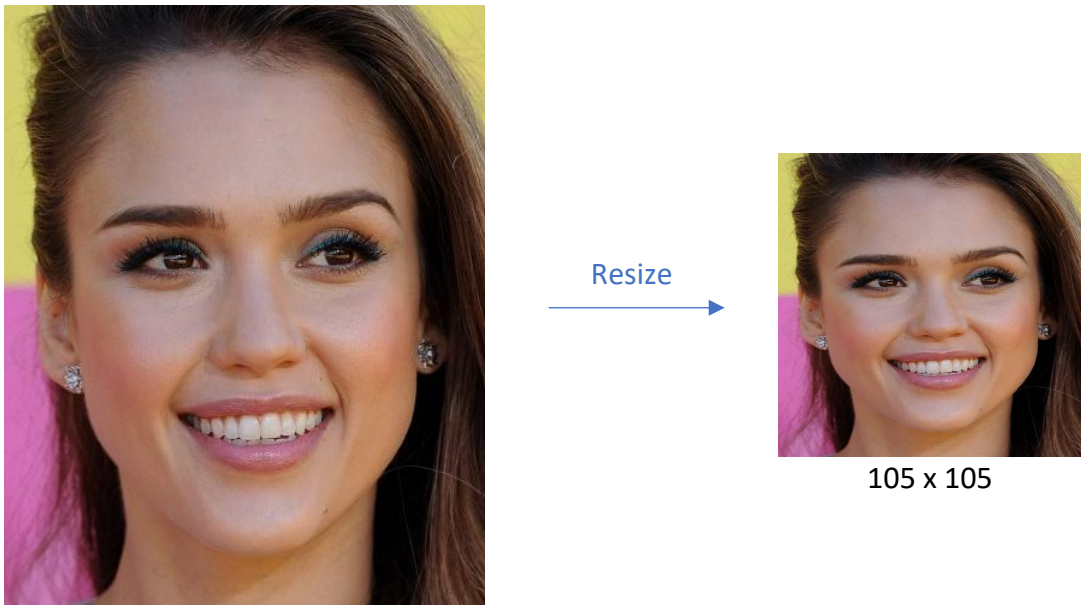


Figure 5 An example of how image resizing is done (Jessica Alba, id 225)

5. Implementation of a Siamese Neural Network

Our implementation takes free inspiration from this 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 A.
As can be seen, after loading the collection of images, we also make a small copy of the train and test set, so that it can be easily reloaded at the next run. Loading the collection means, to us, converting the image 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 then split it by **subject** (will be explained later why).
After that, we define the core of the Siamese Neural Network, as can be seen in Appendix B.
Through TensorFlow, we use a layer variable that may seem to get reinitialized with a new one, but it is not like that: by using the **with** statement, the Deep Learning framework allows us to create a new layer and give it a name. 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:

- At every **with** statement, we initialize a **Convolutional Layer** with **ReLu** as the activation function. The kernel size starts as a 10x10 array and it shrinks, at every new Conv Layer, by 3x3 units, and the filters are, at the beginning, 64: a number that gets halved every time a new Convolutional layer is 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 4096 units and uses **Sigmoid** 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.

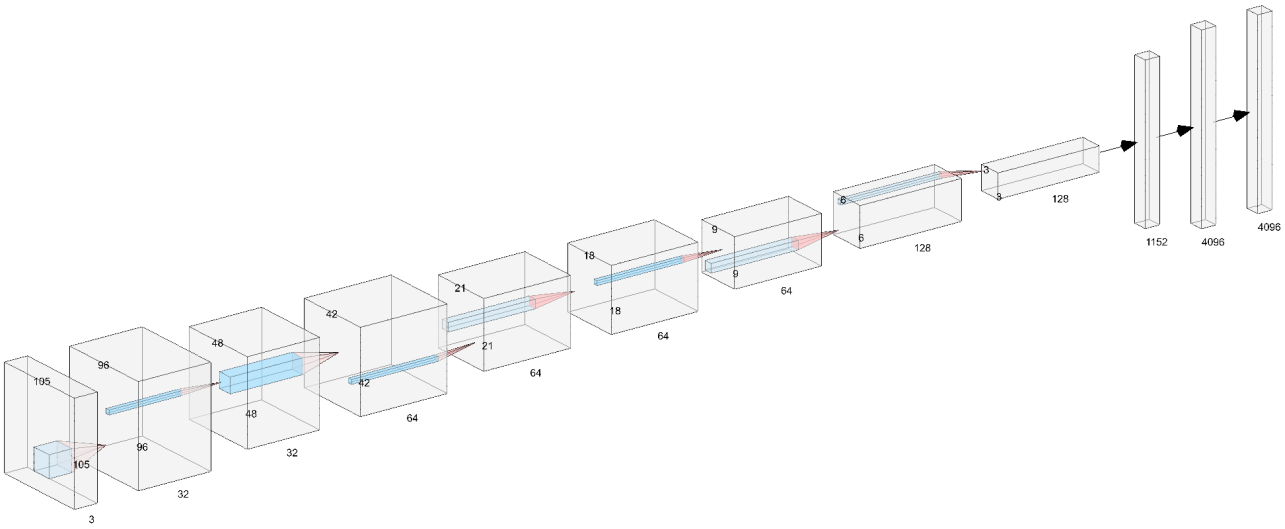


Figure 6 3D representation of our Siamese Network

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.

5.1 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**, 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 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.

$$W \sim U\left[\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right]\right.$$

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

5.2 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**. 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.

$$m_t = (1-\beta_1)\sum_{i=0}^{t} \beta_1^{t-i} g_i$$

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:

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1-y_i) \cdot log(1-p(y_i))$$

Figure 8 The Binary Cross Entropy Formula

5.3 Training and accuracy testing

In order to create batches for training and testing our Siamese Network, we go under two processes: first, we need to generate the batch and its labels, which will then be fed to the network. Our batch size is a hyper-parameter that we set to 32.
This is done by randomly initializing a list with its first half made of positive pairs and the second half with negative ones.
After that, every 500 iterative steps, we run a test to understand what the metrics of the Siamese 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 10 samples) that is made just by one correct pair and a remainder of wrong pairs: this is done to make sure that the network is able to recognize in a good way the person whose sample is being tested.

Given this support set, the model is tasked to score the similarity of each sample in it with the provided test sample, then we select the most similar one, using the argmax function (explained below).

Formally speaking, this is known as a **n-way one-shot learning** (with n = 10). Given a tiny labelled training set S, which will hold N examples, each vectors of the same dimension with a distinct label y.

$$S = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$$

And given x̂ (the test example it must classify), we need to classify the examples in the support set with the right class. But since exactly one example in the support set has the right class, the aim then becomes to correctly predict which y ∈ S is the same as x̂'s label, ŷ. Should we ignore this, then the task would become to try all the possible combinations for a single image, hence, a training epoch *would never meet its natural end*. Suppose, in fact, that we get to pass the full dataset to it: the full thing is composed of 500 classes C, each having 14 examples E (we will use 10 examples per person, since we use only the frontals). Then, we would have

$$N_{pairs} = \binom{C * E}{2} = \frac{(C * E)!}{2!\ (C * E - 2)!} = 12497500$$

In an epoch, the Siamese Network would then need to iterate over all the 12497500 N_{pairs}: this might be inefficient and time wasting. We will go, for this reason, under the assumption that an epoch of ours will last 3000 * 32 iterations: this means that we will check 96000 pairs per epoch, which is a more reasonable amount.

5.4 Prediction

After the optimization process, there is the prediction step, which gets calculated with the **argmax** function. By argmax, we mean the points of the domain of some function in which the function values get maximized. It is defined in the following way:

$$argmax_x\, f(x) := \{\, x \mid \forall y: f(y) \leq f(x)\}$$

We use this approach because it generates the value that, in our opinion, best represents the image that got the highest similarity score, by comparing it to the other template; that is, the person that is the most similar to the one we want to find, given the other persons used for comparison, and therefore, possibly the same person.
The code for this is available in Appendix C.

6. 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 Siamese 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) that will dynamically (and randomly) select a portion of the dataset and give it to the Network.
Every 500 iterations, there is also a "check step": at that point, the algorithm will pick up a new portion of the dataset (Appendix F), totally random, so that it can perform the **one-shot learning**.

6.1 Test Run

By making test runs of our Network, we reach the following results, expressed in terms of Accuracy and Loss

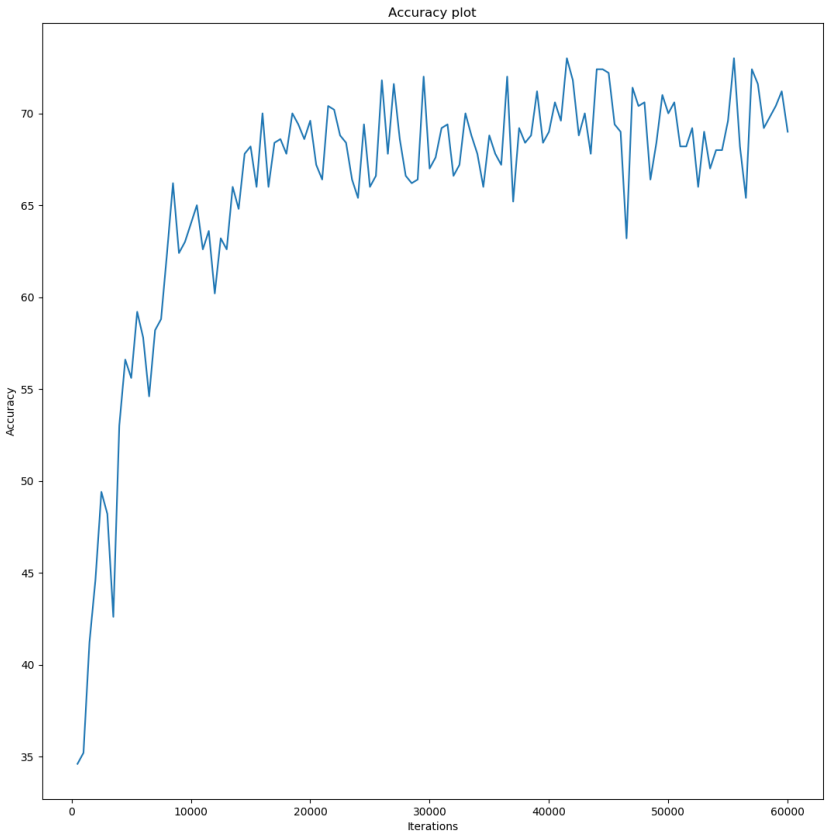


Figure 9 The Accuracy Plot after 60000 iterations of the network

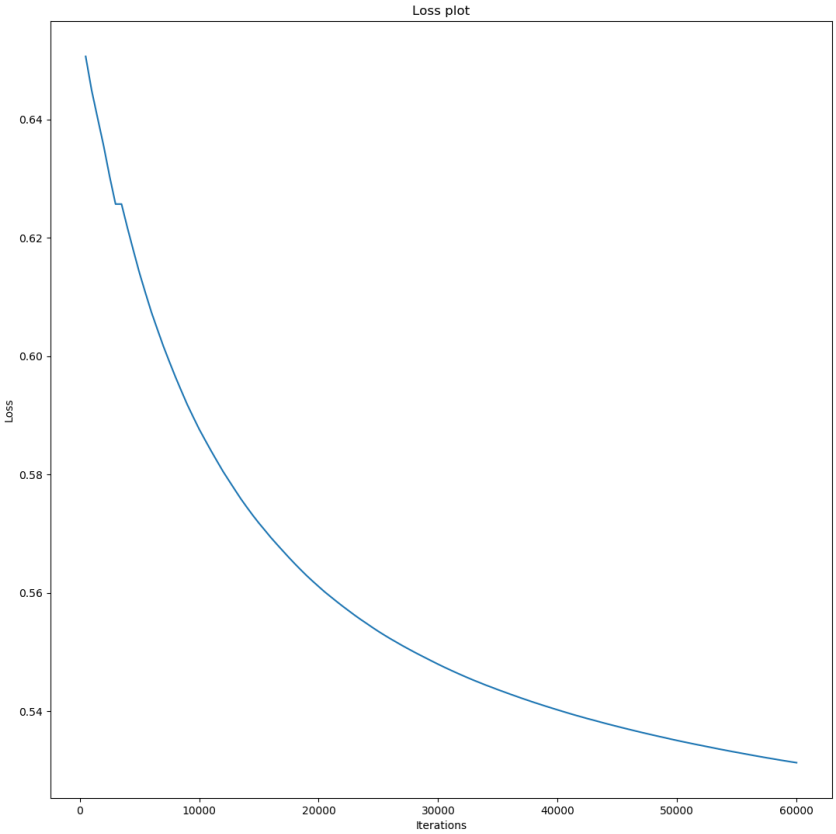


Figure 10 The Loss plot after 60000 iterations of the network

As it is possible to see from Figure 9, the accuracy that the Siamese Network can provide is immediately very high and it can reach in just a few iterations a value between the 65-70%. If we make it run, though, we reach a maximum value of **73%**, which appears to be a good compromise, considering that we are performing One-Shot Learning.
Even the loss has an interesting plot: we could say that it has a decreasing process similar to that of an exponential function.

6.2 Real Test Example

After testing the accuracy of our Siamese Network, it is now time for testing it in a real environment. Usually, there are two cases under which this occurs:

1. **Closed Set**, meaning that the Network will receive as an input images that are somehow known to the dataset (hence, to the network as well);
2. **Open Set**, meaning that our network could be tasked to also evaluate samples of users which are not registered in the dataset

For this project, we decided to stay focused on the **Verification with Multiple Templates:** this means that the network will have to make a match of a person against his/her stored templates on the

dataset (based on the person's identity claim) and, if the outcome is greater or equal than the threshold value, the system will consider that person *who it claims to be*.
The closed set operation will be implemented according to the following pseudocode (where TI = total number of impostor attempts, TG = total number of genuine attempts):

for each threshold t do
  for each row I do
    for each group M_{label} of cells $M_{i,j}$ with same label(j) excluding $M_{i,i}$
      select diff = min(M_{label})
      if diff $\leq$ t then
        if label(i)=label(M_{label}) then GA++
        else FA++
      else if label(i)=label(M_{label}) then FR++
          else GR++
  GAR(t)=GA/TG; FAR(t)=FA/TI;
  FRR(t)=FR/TG; GRR(t)=GR/TI

We now write our way of implementing the *One vs All* pseudocode, which can be seen in the next snippet. Instead of having a matrix, though, we will just iterate over our samples and have the network make its predictions.

```
# 1 VS ALL
os.path.exists(tmp_dir + '/rates.txt') and os.remove(tmp_dir + '/rates.txt')
threshold_list = [1e-8, 3e-8, 5e-8, 7e-8, 9e-8,
                  1e-7, 3e-7, 5e-7, 7e-7, 9e-7,
                  1e-6, 3e-6, 5e-6, 7e-6, 9e-6,
                  1e-5, 3e-5, 5e-5, 7e-5, 9e-5]
for threshold in threshold_list:
    print("Threshold:", threshold)
    TP, FP, FN, TN = 0, 0, 0, 0
    for i, x in tqdm.tqdm(enumerate(X), total=len(X)):
        if i != 0 and i % 250 == 0:
            print("Threshold:", threshold)
            print("TP:", TP, "FP:", FP, "FN:", FN, "TN:", TN)
            print("True Positive Rate:", TP / i)
            print("False Positive Rate:", FP / (i * (len(Y) // 3 - 1)))
            print("False Negative Rate:", FN / i)
            print("True Negative Rate:", TN / (i * (len(Y) // 3 - 1)))
        for category in range(0, len(Y), 3):
            if not Y[i] == category // 3:
                scores = s.predict([x * 3], X[category: category + 3])
            else:
                scores = s.predict([x * 2], X[category: i] + X[i + 1: category + 3])
            max_val = np.max(scores)
            if max_val >= threshold:
                if Y[i] == category // 3:
                    TP += 1
                else:
                    FP += 1
            else:
                if Y[i] == category // 3:
                    FN += 1
                else:
                    TN += 1
    GAR[threshold] = TP / len(X)
    FAR[threshold] = FP / (len(X) * (len(Y) // 3 - 1))
    FRR[threshold] = FN / len(X)
    GRR[threshold] = TN / (len(X) * (len(Y) // 3 - 1))
```

```
    with open(tmp_dir + '/rates.txt', 'a', encoding='utf-8') as w:
        w.write(str(threshold) + '\n')
        w.write(str(TP) + ' ' + str(FP) + ' ' + str(FN) + ' ' + str(TN) + '\n')
        w.write(str(GAR[threshold]) + '\n')
        w.write(str(FAR[threshold]) + '\n')
        w.write(str(FRR[threshold]) + '\n')
        w.write(str(GRR[threshold]) + '\n\n')
```

6.3 Real Test Example – Execution

This is how we proceed: after having loaded our dataset into memory, we initialize the Siamese Network, using the dump that was created once the training process was finished.
We then start running our implementation of the One vs All Algorithm, given the following thresholds:

```
threshold_list = [1e-8, 3e-8, 5e-8, 7e-8, 9e-8,
                  1e-7, 3e-7, 5e-7, 7e-7, 9e-7,
                  1e-6, 3e-6, 5e-6, 7e-6, 9e-6,
                  1e-5, 3e-5, 5e-5, 7e-5, 9e-5]
```

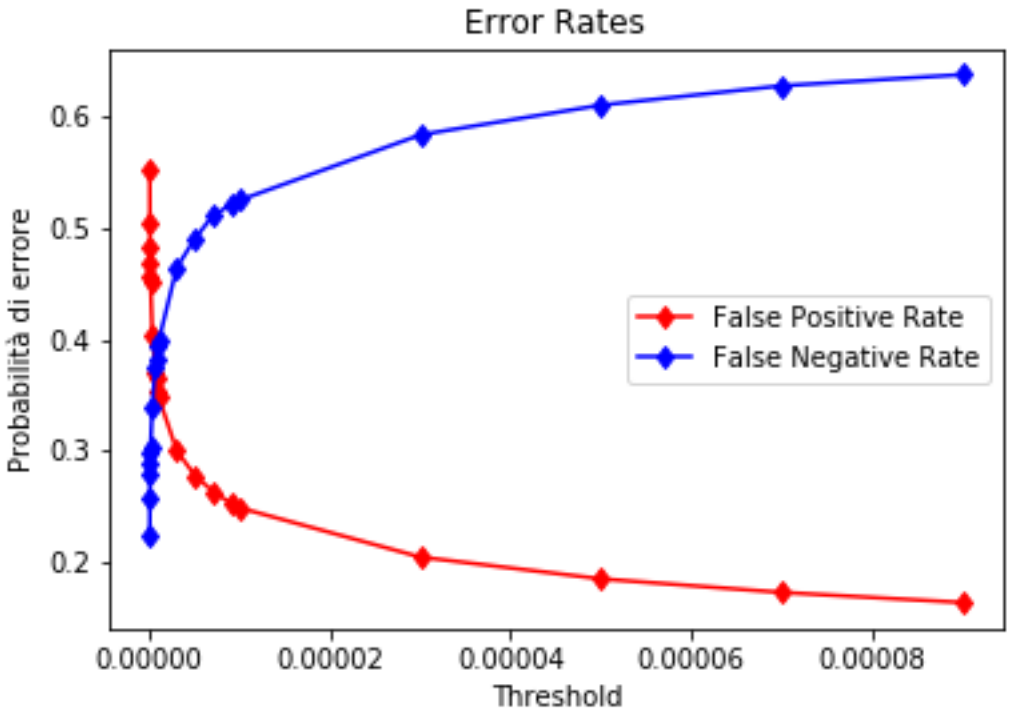
As the first iteration started, we saw that on a consumer grade computer a complete test of all those thresholds would have taken too much time. For this very reason, we decided to migrate our code on a **p2.xlarge** instance of **AWS**, which is configured in the following way:

1. 61GB of RAM
2. 4 vCPUs
3. 1 nVidia K80

By using AWS, we were able to reduce the calculation time from 20 days to just 9 days.

Our implementation of the *One vs All* algorithm works like this: by iterating on the test set, we take 3 images (3 of, say, a person X if X is not the person we are using to make the comparison; 2 otherwise) that will be compared to the one currently analyzed: the network will then perform its predictions. If one of these three predicted values are greater or equal to the current threshold, then it is safe to assume that we have a positive prediction. This will be either true or false, based on the matching of the compared identities. The same holds true if the prediction is negative.

After finishing our iteration over the list of thresholds, we then used all the values collected during the process to build the final graph which shows us how the FAR and FRR change against the threshold. Our objective is to find the Equal Error Rate (ERR) and, by analyzing the following image, it is possible to see that it is located between 0.37 and 0.38.



With the help of the plot we can see that smaller the threshold, the higher the false acceptance rate. To enforce this thought, please consult Appendix G, where it is possible to find all the values that were calculated for each threshold (in order of appearance: GAR, FAR, FRR, GRR).

7. Conclusions

With our work, we managed to adapt a network that was initially created for handwriting recognition to image recognition.
As for future work, it might be interesting to try to tweak the network in order to reach an accuracy value of 75%-80% or more.
An interesting method that we can apply to our project, in order to achieve better results, is to retrain the Siamese Network using the data augmentation technique, like random shift, rotation, zoom, etc.
Very often this trick is used to allow a model to get 5%-10% more accuracy and, at the same time, have better generalization.

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
8.
9.  class SiameseNetwork:
10.
11.     def __init__(self):
12.         self.__DATA_DIR = 'cfp-dataset/Data/Images'
13.         self.__TMP_DIR = 'tmp'
14.
15.         self.__BATCH_SIZE = 32
16.         self.__ITERATIONS = 3000
17.
18.         if not os.path.exists(self.__TMP_DIR):
19.             os.makedirs(self.__TMP_DIR)
20.
21.         self.__GLOBAL_ITER = dp.global_iteration(self.__TMP_DIR + '/iteration.txt')
22.
23.         print('Global iteration:', self.__GLOBAL_ITER)
24.
25.         self.__train_set = []
26.         self.__test_set = []
27.
28.         self.__shape = (105, 105, 3)
29.
30.         self.__graph = tf.Graph()
31.
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])
36.
37.             self.__embeddings_1 = siamese_network(self.__img_1, reuse_variables=False)
38.             self.__embeddings_2 = siamese_network(self.__img_2, reuse_variables=True)
39.
40.             self.__distance = tf.abs(tf.subtract(self.__embeddings_1, self.__embeddings_2)
    )
41.
42.             self.__scores = tf.contrib.layers.fully_connected(inputs=self.__distance, num_
    outputs=1, activation_fn=tf.nn.sigmoid,
43.                                                               biases_initializer=tf.truncated_nor
    mal_initializer(mean=0.5,
44.
                    stddev=0.01))
45.
46.             self.__losses = tf.nn.sigmoid_cross_entropy_with_logits(labels=self.__flags,
47.                                                                     logits=tf.reshape(self.__scor
    es, shape=[self.__BATCH_SIZE]))
48.             self.__loss = tf.reduce_mean(self.__losses)
49.
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)
53.
54.             self.__prediction = tf.cast(tf.argmax(self.__scores, axis=0), dtype=tf.int32)
```

```
55.
56.             self.__saver = tf.train.Saver()
57.
58.             init = tf.global_variables_initializer()
59.
60.             ### SESSION ###
61.             self.__session = tf.Session(graph=self.__graph)
62.
63.             # We must initialize all variables before we use them.
64.             init.run(session=self.__session)
65.
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.c
    kpt'))
69.                 print('Model restored')
70.             except:
71.                 print('Model initialized')
72.
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.__DAT
    A_DIR)
78.
79.         # Open a writer to write summaries.
80.         self.__writer = tf.summary.FileWriter(self.__TMP_DIR, self.__session.graph)
81.
82.         average_loss = 0
83.
84.         for step in tqdm.tqdm(range(self.__ITERATIONS * epochs), desc='Training Siamese Ne
    twork'):
85.             batch, label = dp.get_batch(self.__train_set, self.__BATCH_SIZE)
86.
87.             pair_1 = np.array([b[0] for b in batch])
88.             pair_2 = np.array([b[1] for b in batch])
89.
90.             # Define metadata variable.
91.             run_metadata = tf.RunMetadata()
92.
93.             _, l = self.__session.run([self.__train_op, self.__loss], feed_dict={self.__im
    g_1: pair_1, self.__img_2: pair_2, self.__flags: label},
94.                                       run_metadata=run_metadata)
95.
96.             average_loss += l
97.
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])
106.
107.                     run_metadata = tf.RunMetadata()
108.
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
112.
113.                 print('Loss:', str(average_loss / step), '\tAccuracy:', correct / k
    )
114.
```

```
                        with open(self.__TMP_DIR + '/log.txt', 'a', encoding='utf8') as f:
                            f.write(str(correct / k) + ' ' + str(average_loss / step) + '\n')

                    if step == (self.__ITERATIONS - 1):
                        self.__writer.add_run_metadata(run_metadata, 'step%d' % step, global_step=self.__GLOBAL_ITER + step + 1)

                self.__saver.save(self.__session, os.path.join(self.__TMP_DIR, 'model.ckpt'))
                dp.global_iteration(self.__TMP_DIR + '/iteration.txt', update=self.__GLOBAL_ITER + step + 1)

                pg.generate_accuracy_plot(self.__TMP_DIR + '/')
                pg.generate_loss_plot(self.__TMP_DIR + '/')

                self.__writer.close()

            def predict(self, imgs1, imgs2):
                run_metadata = tf.RunMetadata()
                similarity_scores = self.__session.run(self.__scores, feed_dict={self.__img_1: imgs1, self.__img_2: imgs2}, run_metadata=run_metadata)
                return similarity_scores
```

Appendix B – Creation of the Model

```
1.  import tensorflow as tf
2.
3.  # MODEL #
4.
5.  def siamese_network(img, reuse_variables=False):
6.
7.      with tf.name_scope('siamese'):
8.
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_initia
    lizer(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, padd
    ing='VALID')
14.
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_initia
    lizer(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, padd
    ing='VALID')
20.
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_initia
    lizer(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, padd
    ing='VALID')
26.
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_initia
    lizer(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, padd
    ing='VALID')
32.
33.         with tf.variable_scope('flatten') as scope:
34.             layer = tf.contrib.layers.flatten(inputs=layer)
35.
36.         with tf.variable_scope('fc') as scope:
37.             layer = tf.contrib.layers.fully_connected(inputs=layer, num_outputs=4096, acti
    vation_fn=tf.nn.sigmoid,
38.                                                       biases_initializer=tf.truncated_norm
    al_initializer(mean=0.5, stddev=0.01),
39.                                                       scope=scope, reuse=reuse_variables)
40.
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])
5.
6.      self.__embeddings_1 = siamese_network(self.__img_1, reuse_variables=False)
7.      self.__embeddings_2 = siamese_network(self.__img_2, reuse_variables=True)
8.
9.      self.__distance = tf.abs(tf.subtract(self.__embeddings_1, self.__embeddings_2))
10.
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_init
    ializer(mean=0.5,
13.
            stddev=0.01))
14.
15.     self.__losses = tf.nn.sigmoid_cross_entropy_with_logits(labels=self.__flags,
16.                                                             logits=tf.reshape(self.__scores, shap
    e=[self.__BATCH_SIZE]))
17.     self.__loss = tf.reduce_mean(self.__losses)
18.
19.     self.__optimizer = tf.train.AdamOptimizer(learning_rate=0.00005)
20.     # optimizer = tf.train.MomentumOptimizer(learning_rate=0.0001, momentum=0.95, use_nest
    erov=True)
21.     self.__train_op = self.__optimizer.minimize(self.__loss)
22.
23.     self.__prediction = tf.cast(tf.argmax(self.__scores, axis=0), dtype=tf.int32)
24.
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_DI
    R)
6.
7.      # Open a writer to write summaries.
8.      self.__writer = tf.summary.FileWriter(self.__TMP_DIR, self.__session.graph)
9.
10.     average_loss = 0
11.
12.     for step in tqdm.tqdm(range(self.__ITERATIONS * epochs), desc='Training Siamese Networ
    k'):
13.         batch, label = dp.get_batch(self.__train_set, self.__BATCH_SIZE)
14.
15.         pair_1 = np.array([b[0] for b in batch])
16.         pair_2 = np.array([b[1] for b in batch])
17.
18.         # Define metadata variable.
19.         run_metadata = tf.RunMetadata()
20.
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)
23.
24.         average_loss += l
25.
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])
34.
35.                 run_metadata = tf.RunMetadata()
36.
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
40.
41.             print('Loss:', str(average_loss / step), '\tAccuracy:', correct / k)
42.
43.             with open(self.__TMP_DIR + '/log.txt', 'a', encoding='utf8') as f:
44.                 f.write(str(correct / k) + ' ' + str(average_loss / step) + '\n')
45.
46.         if step == (self.__ITERATIONS - 1):
47.             self.__writer.add_run_metadata(run_metadata, 'step%d' % step, global_step=self
    .__GLOBAL_ITER + step + 1)
48.
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 + ste
    p + 1)
51.
52.     pg.generate_accuracy_plot(self.__TMP_DIR + '/')
53.     pg.generate_loss_plot(self.__TMP_DIR + '/')
54.
55.     self.__writer.close()
```

Appendix E – Training Dataset Generation

```
def get_batch(train_set, batch_size):
    cat = np.random.choice(list(range(len(train_set))), size=batch_size, replace=False)
    #print(cat)
    label = np.zeros(batch_size)
    # If the inputs are from the same class, then the value of label is 1, otherwise label is 0
    label[:batch_size // 2] = 1
    batch = []
    for i in range(batch_size // 2):
        category = cat[i]
        random_index = np.random.randint(0, len(train_set[category]))
        img_1 = train_set[category][random_index]
        random_index = np.random.randint(0, len(train_set[category]))
        img_2 = train_set[category][random_index]
        batch.append((img_1, img_2))
    for i in range(batch_size // 2, batch_size):
        category_1 = cat[i]
        random_index = np.random.randint(0, len(train_set[category_1]))
        img_1 = train_set[category_1][random_index]
        category_2 = (category_1 + np.random.randint(1, len(train_set))) % len(train_set)

        img_2 = train_set[category_2][random_index]
        batch.append((img_1, img_2))
    return batch, label
```

Appendix F – One Shot Testing Dataset Generation

```
def get_one_shot_test(test_set, n_examples=10):
    # Questa funzione ritorna una lista di 10 coppie, dove la prima  con la medesima persona, le altre sono persone diverse
    n_classes = len(test_set)
    # n_examples = len(test_set[0])
    cat = np.random.choice(list(range(n_classes)), size=n_classes, replace=False)
    random_indexes = np.random.randint(0, len(test_set[0]), size=n_examples)
    true_cat = cat[0]
    ex1, ex2 = np.random.choice(len(test_set[0]), replace=False, size=2)
    test = []
    label = np.zeros(n_classes)
    img_1 = test_set[true_cat][ex1]
    for k, random_index in enumerate(random_indexes):
        if k == 0:
            img_2 = test_set[cat[k]][ex2]
        else:
            img_2 = test_set[cat[k]][random_index]
        test.append((img_1, img_2))
    label[0] = 1
    return test, label
```

Appendix G – Threshold values

```
5e-05
584 138617 916 609883
0.3893333333333333
0.1851930527722111
0.6106666666666667
0.8148069472277889

7e-05
558 129523 942 618977
0.372
0.17304342017368068
0.628
0.8269565798263193

9e-05
543 122758 957 625742
0.362
0.1640053440213761
0.638
0.8359946559786239

6e-07
936 278201 564 470299
0.624
0.3716780227120908
0.376
0.6283219772879092

5e-08
1081 361678 419 386822
0.7206666666666667
0.48320374081496326
0.2793333333333333
0.5167962591850367

7e-08
1066 350497 434 398003
0.7106666666666667
0.46826586506346024
0.28933333333333333
0.5317341349365398

9e-08
1053 342245 447 406255
0.702
0.45724114896459583
0.298
0.5427588510354041

1e-07
1045 338693 455 409807
0.6966666666666667
0.4524956579826319
0.30333333333333334
0.5475043420173681

7e-06
733 197443 767 551057
0.4886666666666667
0.26378490313961256
```

0.5113333333333333
0.7362150968603874

9e-06
718 189560 782 558940
0.4786666666666667
0.25325317301269207
0.5213333333333333
0.746746826987308

1e-05
712 186319 788 562181
0.4746666666666667
0.24892317969271877
0.5253333333333333
0.7510768203072812

3e-05
624 153278 876 595222
0.416
0.2047802271209085
0.584
0.7952197728790915

9e-07
909 264519 591 483981
0.606
0.3533987975951904
0.394
0.6466012024048097

1e-06
902 260935 598 487565
0.6013333333333334
0.3486105544422178
0.39866666666666667
0.6513894455577822

3e-06
	803 224821 697 523679
	0.5353333333333333
	0.3003620574482298
	0.4646666666666667
	0.6996379425517703

	5e-06
	764 208208 736 540292
	0.5093333333333333
	0.27816700066800265
	0.49066666666666664
	0.7218329993319973

	7e-07
	925 272996 575 475504
	0.6166666666666667
	0.36472411489645956
	0.38333333333333336
	0.6352758851035404

	1e-08
	1163 413607 337 334893
	0.7753333333333333
	0.5525811623246493
	0.22466666666666665
	0.4474188376753507

	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