Face Recognition

through a Siamese Network

Emanuele Alessi (1486470)

Gianmarco Forcella (1725967)

Gabriel Radu Taranciuc (1693558)

Sommario

[1. Introduction 3](#_Toc4582579)

[2. Theoretical overview 4](#_Toc4582580)

[2.1 Convolutional Neural Networks 4](#_Toc4582581)

[2.2 Siamese Networks 4](#_Toc4582582)

[2.3 Keras 6](#_Toc4582583)

[2.4 Code Example 6](#_Toc4582584)

[2.5 TensorFlow 6](#_Toc4582585)

[3. Project setup 8](#_Toc4582586)

[4. Implementation of a Siamese Neural Network 9](#_Toc4582587)

[4.1 The Weight inizialization problem 9](#_Toc4582588)

[4.2 Minimizing loss 10](#_Toc4582589)

[4.3 Prediction 10](#_Toc4582590)

[5. Model Evaluation 11](#_Toc4582591)

[5.1 Test Run 11](#_Toc4582592)

[5.2 Real Test Example 13](#_Toc4582593)

[Appendix A – Dataset loading 14](#_Toc4582594)

[Appendix B – Creation of the Model 15](#_Toc4582595)

[Appendix C – Evaluation Process 16](#_Toc4582596)

[Appendix D – Accuracy Calculation 16](#_Toc4582597)

[Appendix E – Training Dataset Generation 17](#_Toc4582598)

[Appendix F – One Shot Testing Dataset Generation 18](#_Toc4582599)

# Introduction

Face Detection and Recognition are one of the most popular fields in the study of Artificial Intelligence studies. But what’s the difference between them? Many people still make a mistake in differencing the two approaches.

* When somebody is talking about Face Detection, it is referring to a task in which the goal is to find some faces in a given image (ie: is this a face / is there a face in this image?)
* On the contrary, Face Recognition is applied when there is the necessity to identify the person on a given image (ie: 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 to 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 Identification) of a given image through a Siamese Network.

Why a Siamese Network? How did we reach it? And what is it?

In the following chapters, we will going to describe our architectural choices also by providing the source code and we will see 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

Before going deeper with describing the problem, let us first describe what a Convolutional Neural Network is.

This type of Neural Network 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 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 of this important? As an example, let’s assume that we want to process a 48x48x3 image with a Multilayer Neural Network with Sigmoid as its activation function: this would mean to have, just for a single neuron in the first hidden layer, about 6912 weights (not to mention also the problem of the Vanishing / Exploding Gradient)! By exploiting the strong spacially 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 perform the dot product between the entry of the filter and the postions that are close to the input computed; the output is influenced by some hyperparameters
* Pooling: usually inserted between one convulational layer and another, they are used to reduce the amount of parameters and computation in the network, in order to avoid overfitting
* Fully-connected layer: usually placed at the end of the Convolutation Neural Network. Since the neurons are fully connected, the layer is goingn to be threated as a normal neural network.

## 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: in this approach, as it may be easy to guess, the objective is to learn information about object categories from one or just a few training samples.

One of the most common “exploiter” of the one-shot learning is the **Siamese Network**, which was first reported in the ‘90s researchs’ 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 like an ensemble method.

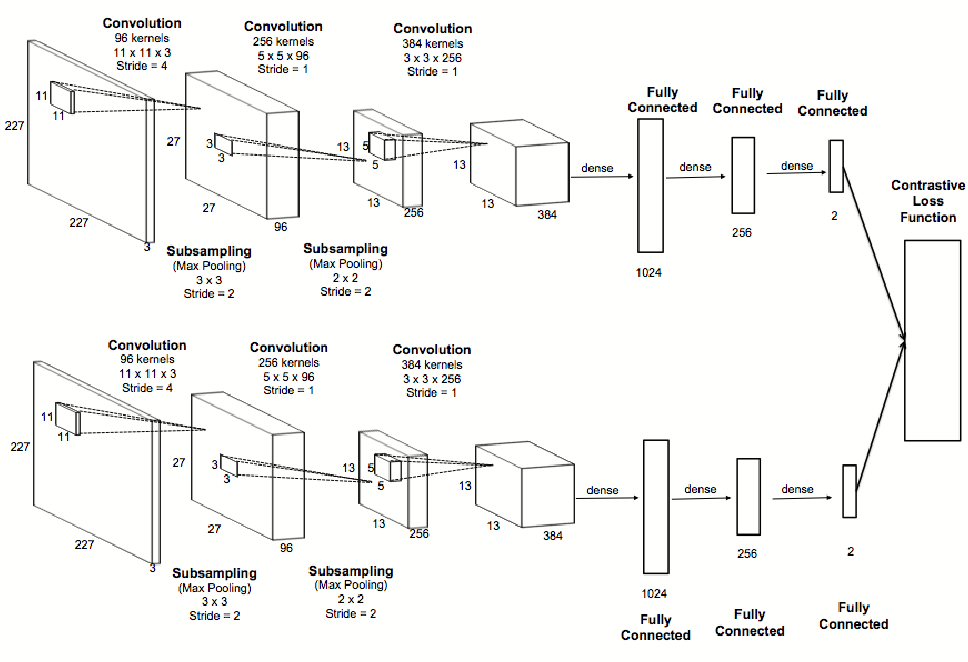


Figure 1 An example of a Siamese Network

Being that the networks are twins with each other, not only 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 predictions’ 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 to which network first an image is presented, it will always calculate the same metric (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 are in the network) and then it will calculate the distance d between each of the points in the final n vectors. There are two possible outputs allowed:

* If d is small, then it is possible to say that the images are similar between them. That’s because the twin networks calculated the same values through their neurons, so it is safe to assume that the images might be the same
* On the contrary, if d holds an high value, it is safe to assume that the the images are not similar.

## Keras

In order to implement our Convolutional Neurl Network we can’t use scikit: this is due to the fact that the famous Machine Learning library doesn’t support any of the Deep Learning algorithms.

For this reason, we chose to use Keras, which is a high-level API deisgned just for handling neural networks.

But why Keras is popular? There are, mainly, 2 reasons:

* Its user “friendliness”: it offers consistent and simle APIs, which minimizes the number of actions by an user for common cases and provides a readable stack trace, in case of some-kind of error;
* Its modularity, which if exploited well, allows the user to create new model as he/she wants (ie.: neural layers, cost functions, optimizers and so on are all modules)

But that is not where this framework stops: Keras can, in fact, have a custom back-end proessor that can do all the calculations. The most used are Tensorflow or Theano but there many others.

## Code Example

Implementing a Convolutional Neural Network in Keras is as simply as

1. **from** keras.models **import** Sequential
2. **from** keras.layers **import** Dense, Conv2D, Flatten
3. #create model
4. model = Sequential()
5. #add model layers
6. model.add(Conv2D(64, kernel\_size=3, activation=’relu’, input\_shape=(28,28,1)))
7. model.add(Conv2D(32, kernel\_size=3, activation=’relu’))
8. model.add(Flatten())
9. model.add(Dense(10, activation=’softmax’))

As it is possible to see, we just need to initialize the model as in line 4 and then add as many layers as we want: in this short example, we just use two convolutional layers, a “connection” between these and the fully-connected neural network (respectively, line 8 and 9).

Note that the parameters specified in parenthesis are all arbitrary and they can hold any value that is accepted as per the documentation.

## 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, starting from the digit classification and arriving to the image recognition.

But how does it work?

Tensorflow allows to create the 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 much more speed in terms of calculations, in fact, the library executes these operations in C++, so that they can work at a low-level context.

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

As of 2019, Tensorflow is accreditated as one of the most used libraries for Deep Learning and it keeps growing, even with a 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 favour 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 X to immediately see our implementation of the Convolutional Neural Network.

# Project setup

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

* **Python 3.x**
* **PyCharm as our IDE**
* **Numpy**
* **Pandas**
* **Tensorflow**
* **Keras**
* **The** [**CFP dataset**](http://www.cfpw.io/) (which is a collection of front and profile face images that belongs to many celebrities)

To check out the full project, please refer to [this](https://github.com/TarazGr/BSProj) GitHub repository, which also contains all the papers from which we were inspired for this work.

# Implementation of a Siamese Neural Network

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

The first thing we do is, of course, loading the dataset: this can be seen in [Appendix](#_Appendix_A_–) A.

As it 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 easier to load it at the next run: loading the collection means, to us, converting the image to an Image object in Python, convert it as a 3x3 Numpy Array (because we load it with a 3 color channels), cast it to a simple array and then split it depending on the **subject** (will be explained later why).

After that, we define the core of the Siamese Neural Network, as it is possible to see in [Appendix](#_Appendix_B_–) B.

Through Tensorflow, we use a layer variable that, everytime, it may seem that it is getting reinitialized with a new layer but it is, actually, not like that: by using the **with** statement, the Deep Learning framework allows us to create a new layer and give to it a name. When the layer is assigned, it **also** gets executed: this means that the user don’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 a layer to another.

Following are the list of the used layers:

* 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, of 3x3 units less and the filters are, at the beginning, 64: a number that decreases by its half everytime it is initialized a new Convolutional layer.
* Immediately after the Convolutional layer, we have a **Max Pooling** layer which purpose is to decrease by its half the 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 after the fully connected layer, there is a **Flatten Layer** that allows to have the input “collapse” in just one dimension
* The final layer is a **Fully Connected**, which actually symbolize the fully connected Neural Network, is made of 4096 units and holds **Sigmoid** as the activation function. As per the input, it takes what the layer variable holds on that point of the program, which should by now represent an image that has now just one dimension.

## The Weight inizialization problem

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

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

which didn’t allow us to produce any satisfying predictions as output. We then decided to change this part of our implementation and opted for the [**Glorot-Bengio weight initialization technique**](http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf)**,** which is the standard way with which Tensorflow initialize 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 hold a value that has a range that starts from minus the square root of 6 divided by the input units + the output units until the positive of this very formula.

By doing so, we achieve greater results, also in term 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 the [**Adam’s** one](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 might be:

* It is **easy** to implement;
* It is **computationally** efficient
* It requires just a little bit of memory
* Invariant to diagonal rescale of the gradients

But how does it work? At its core, Adam takes inspiration from two other extensions of the stochastic gradient descent, that are, the **AdaGrad** and the **RMSProp** (both maintains 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 (meaning we don’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 having two other parameters (namely, beta1 and beta2) that will control the decay rates of these moving averages.

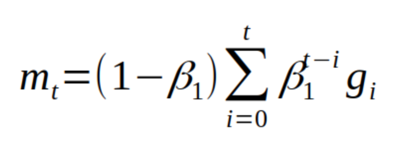


Figure 2 the moving average calculated with Adam formulation

## 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 gets maximized.

The code for this is available in [Appendix C.](#_Appendix_C_–)

# Model Evaluation

The evaluation process starts by initializing a Tensorflow’s 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](#_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](#_Appendix_F_–)), totally randomic, so that it can perform the **one shot testing**.

## Test Run

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

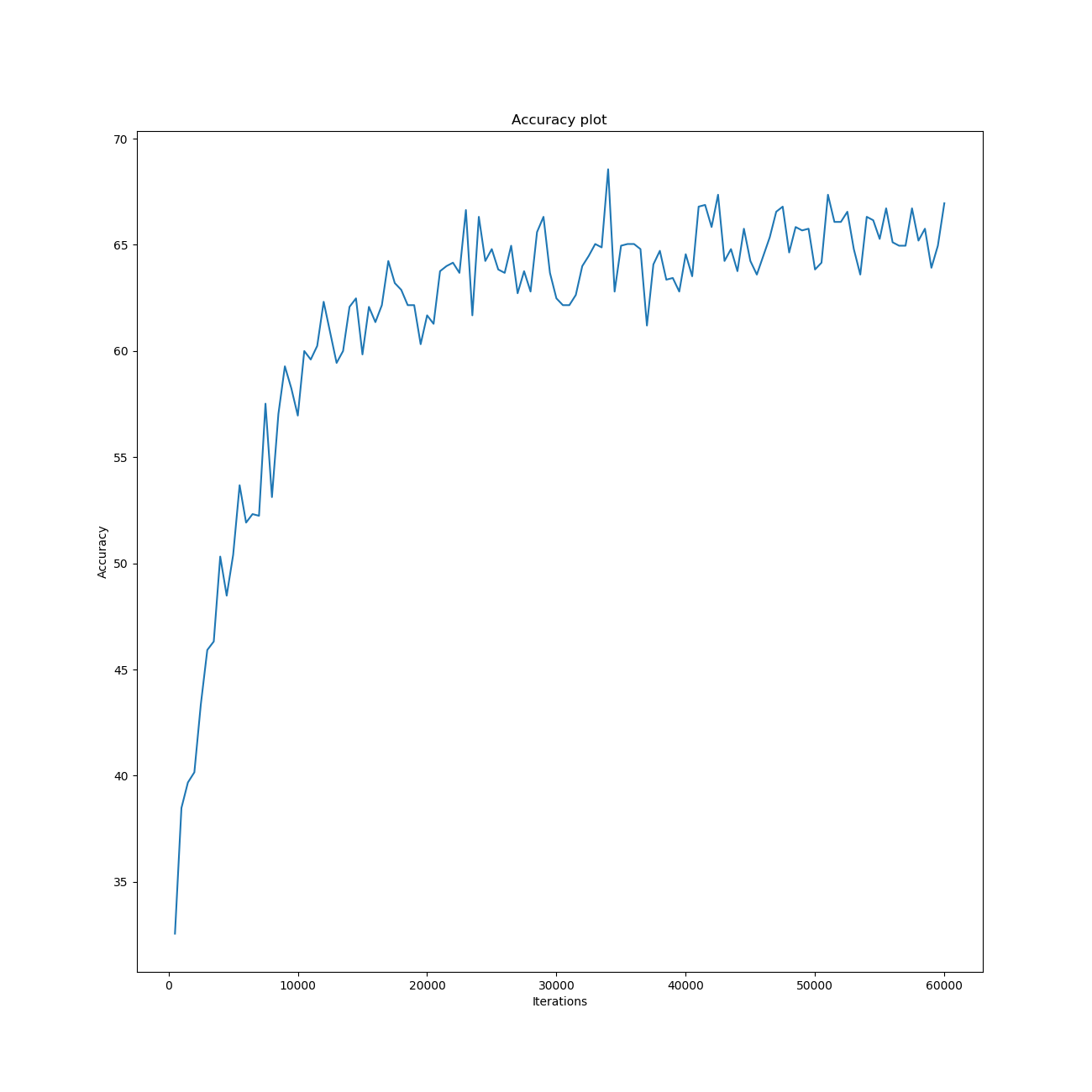


Figure 3 The Accuracy Plot after 60000 iterations of the network

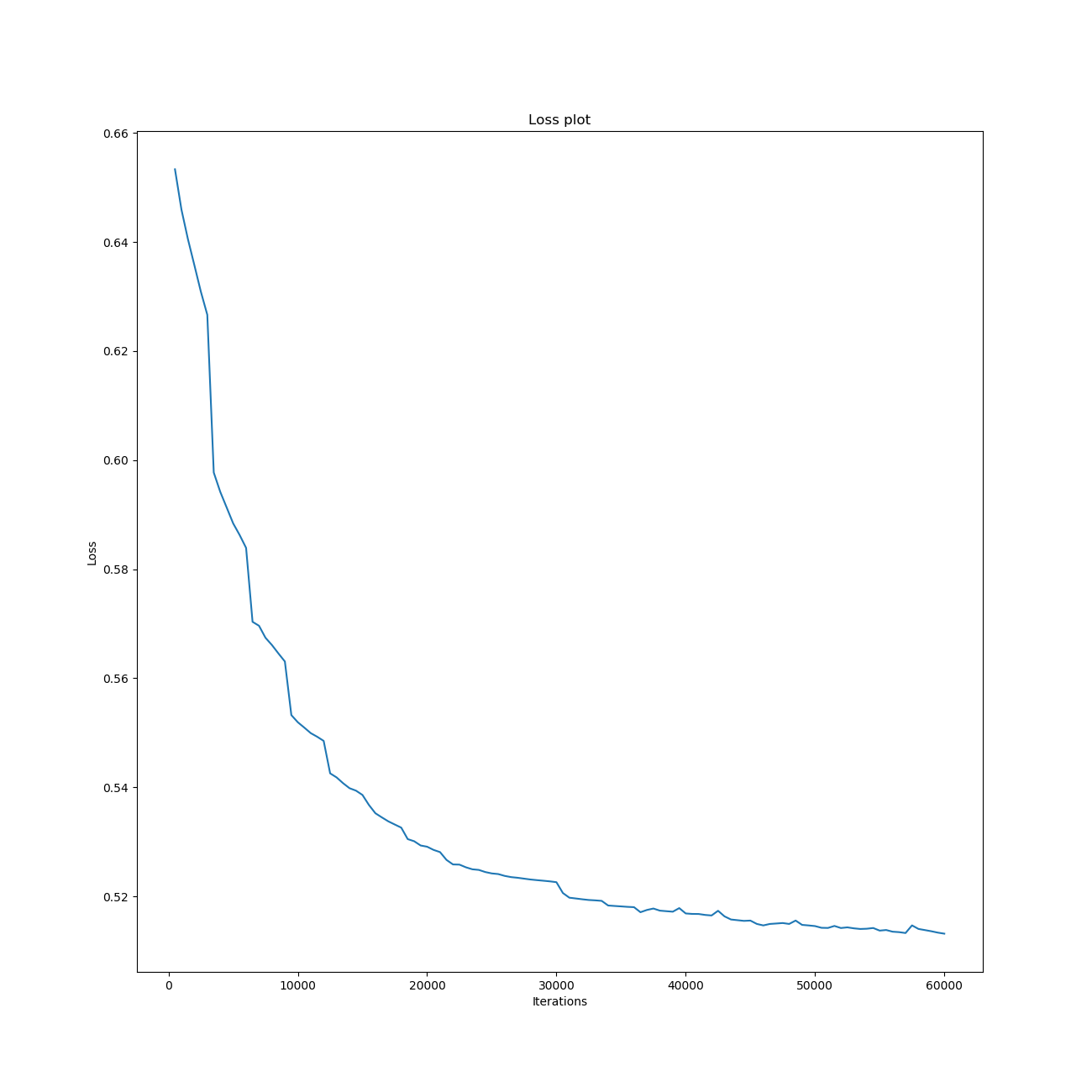


Figure 4 The Loss plot after 60000 iterations of the network

As it is possible to see from Figure 3, 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 55-60%. If we make it run, though, we reach a maximum value of **67%**, which seems a good compromise, considering also the fact that we are performing a One Shot Testing.

Even the loss has an interesting plot: we could say that it has a decrease process like an exponential function.

## Real Test Example

# Appendix A – Dataset loading

1. DATA\_DIR = 'cfp-dataset/Data/Images'
2. TMP\_DIR = 'tmp'
4. BATCH\_SIZE = 32
5. ITERACTIONS = 3000
7. **if** **not** os.path.exists(TMP\_DIR):
8. os.makedirs(TMP\_DIR)
10. GLOBAL\_ITER = dp.global\_iteraction(TMP\_DIR + '/iteraction.txt')
12. **print**('Global iteraction:', GLOBAL\_ITER)

15. **if** **not** os.path.exists(TMP\_DIR + '/train\_set.pkl') **or** **not** os.path.exists(TMP\_DIR + '/test\_set.pkl'):
16. **print**('Loading dataset...')
17. dataset = []
18. category2index = {}
19. index = 0
20. **for** folder **in** os.listdir(DATA\_DIR):
21. **if** folder **not** **in** category2index:
22. category2index[folder] = index
23. index += 1
24. imgs = []
25. directory = os.path.join(DATA\_DIR, folder)
26. dir = os.path.join(directory, 'frontal')
27. **for** img **in** os.listdir(dir):
28. img = Image.open(os.path.join(dir, img)).convert('RGB').resize((105, 105), Image.LANCZOS)
29. img = np.asarray(img)
30. imgs.append(img)
31. dataset.append(imgs)
32. train\_set, test\_set = dp.split\_dataset(dataset, category2index, train\_size=0.75)
33. **del** dataset
34. pickle.dump(train\_set, open(TMP\_DIR + '/train\_set.pkl', 'wb'))
35. pickle.dump(test\_set, open(TMP\_DIR + '/test\_set.pkl', 'wb'))
36. **else**:
37. **print**('Reloading train set and test set...')
38. train\_set = pickle.load(open(TMP\_DIR + '/train\_set.pkl', 'rb'))
39. test\_set = pickle.load(open(TMP\_DIR + '/test\_set.pkl', 'rb'))

42. shape = train\_set[0][0].shape

# Appendix B – Creation of the Model

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

# Appendix C – Evaluation Process

1. with graph.as\_default():
2. img\_1 = tf.placeholder(tf.float32, shape=[None, shape[0], shape[1], shape[2]])
3. img\_2 = tf.placeholder(tf.float32, shape=[None, shape[0], shape[1], shape[2]])
4. flags = tf.placeholder(tf.float32, shape=[None])
6. embeddings\_1 = siamese\_network(img\_1, reuse\_variables=False)
7. embeddings\_2 = siamese\_network(img\_2, reuse\_variables=True)
9. distance = tf.abs(tf.subtract(embeddings\_1, embeddings\_2))
11. scores = tf.contrib.layers.fully\_connected(inputs=distance, num\_outputs=1, activation\_fn=tf.nn.sigmoid,
12. biases\_initializer=tf.truncated\_normal\_initializer(mean=0.5, stddev=0.01))
14. losses = tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=flags, logits=tf.reshape(scores, shape=[BATCH\_SIZE]))
15. loss = tf.reduce\_mean(losses)
17. optimizer = tf.train.AdamOptimizer(learning\_rate=0.00005)
18. #optimizer = tf.train.MomentumOptimizer(learning\_rate=0.0001, momentum=0.95, use\_nesterov=True)
19. train\_op = optimizer.minimize(loss)
21. prediction = tf.cast(tf.argmax(scores, axis=0), dtype=tf.int32)
23. saver = tf.train.Saver()

# Appendix D – Accuracy Calculation

1. with tf.Session(graph=graph) as session:
2. session.run(tf.global\_variables\_initializer())
4. writer = tf.summary.FileWriter(TMP\_DIR, session.graph)
6. # reload the model if it exists and continue to train it
7. **try**:
8. saver.restore(session, os.path.join(TMP\_DIR, 'model.ckpt'))
9. **print**('Model restored')
10. **except**:
11. **print**('Model initialized')
13. average\_loss = 0
15. **for** step **in** tqdm.tqdm(range(ITERACTIONS), desc='Training Siamese Network'):
16. batch, label = dp.get\_batch(train\_set, BATCH\_SIZE)
18. pair\_1 = np.array([b[0] **for** b **in** batch])
19. pair\_2 = np.array([b[1] **for** b **in** batch])
21. # Define metadata variable.
22. run\_metadata = tf.RunMetadata()
24. \_, l = session.run([train\_op, loss], feed\_dict={img\_1: pair\_1, img\_2: pair\_2, flags: label},
25. run\_metadata=run\_metadata)
27. average\_loss += l
29. # print loss and accuracy on test set every 500 steps
30. **if** (step % 500 == 0 **and** step > 0) **or** (step == (ITERACTIONS - 1)):
31. correct = 0
32. k = len(test\_set) \* len(test\_set[0])
33. **for** \_ **in** range(k):
34. test, label = dp.get\_one\_shot\_test(test\_set)
35. pair\_1 = np.array([b[0] **for** b **in** test])
36. pair\_2 = np.array([b[1] **for** b **in** test])
38. run\_metadata = tf.RunMetadata()
40. pred = session.run(prediction, feed\_dict={img\_1: pair\_1, img\_2: pair\_2}, run\_metadata=run\_metadata)
41. **if** pred[0] == 0:
42. correct += 1
44. **print**('Loss:', str(average\_loss / step), '\tAccuracy:', correct / k)
46. with open(TMP\_DIR + '/log.txt', 'a', encoding='utf8') as f:
47. f.write(str(correct / k) + ' ' + str(average\_loss / step) + '\n')
49. **if** step == (ITERACTIONS - 1):
50. writer.add\_run\_metadata(run\_metadata, 'step%d' % step, global\_step=GLOBAL\_ITER + step + 1)
52. saver.save(session, os.path.join(TMP\_DIR, 'model.ckpt'))
53. dp.global\_iteraction(TMP\_DIR + '/iteraction.txt', update=GLOBAL\_ITER + step + 1)

# 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):
2. n\_classes = len(test\_set)
3. n\_examples = len(test\_set[0])
4. cat = np.random.choice(list(range(n\_classes)), size=n\_classes, replace=False)
5. random\_indexes = np.random.randint(0, n\_examples, size=n\_examples)
6. true\_cat = cat[0]
7. ex1, ex2 = np.random.choice(n\_examples, replace=False, size=2)
8. test = []
9. label = np.zeros(n\_classes)
10. img\_1 = test\_set[true\_cat][ex1]
11. k = 0
12. **for** random\_index **in** 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. k += 1
19. label[0] = 1
20. **return** test, label