VGG network for re-identification

Alessandra Alfani
alessandra.alfani@stud.unifi.it

Giulia Pellegrini
giulia.pellegrini1@stud.unifi.it

Abstract

In the context of video analysis, face re-identification aims to verify if certain detected faces have already been observed. In this paper we present some different implementations of an algorithm that performs face re-identification using the VGG network.

1. Introduction

Person re-identification is the problem of identifying faces across images that have been taken using different cameras, or across time using a single camera. Re-identification is an important capability for surveillance systems as well as human-computer interaction systems. It is a particularly difficult problem, because large variations in lighting or different views can cause two images of the same face to look quite different and images of different people to look very similar.
In this project we develop a face re-identification system capable of working on a video stream captured by a webcam or a camera. The system processes the RGB stream so as to detect the presence of one or more faces in the current frame and extracts a descriptor of the face using the *VGG* convolutional neural network. Every detected face has to be associated to a person ID. The system at first determines by overlap if the current person is the same person as in the previous frame, otherwise checks if the descriptor matches one of the persons previously observed, if not, creates a new person ID.
The aim of this project is, once extracted face's informations from *VGG*, to find an appropriate and efficient way to keep track of these descriptors to be used for re-identification purpose.

2. Implementation

This project is developed in *Python 3* using the *TensorFlow*[1]'s implementation of the *Keras*[2] *API* specification. Initially we create the *VGG model* and load the *Cascade classifier*.

VGG Model

VGG is a convolutional neural network with 16 layers, tipically used to perform image localization and classification. The network takes as input an image of size 224x224x3(RGB) and gives a feature vector as output.
In particular we build the *VGG-Face model* as explained in "`https://aboveintelligent.com/face-recognition-with-keras-and-opencv-2baf2a83b799`" employing the pre-trained weights from *MatConvNet*[3]. These weights are obtained by training the network on the *OXFORD VGG Face* dataset of 2622 celebrities.

Cascade Classifier

Cascade classifiers are particular structures based on the concatenation of several classifiers, in which the output of a classifier is given as an additional input to the next one. A simple and rapid way to perform object detection is using Haar Feature-based Cascade Classifiers. This is a machine learning based approach where a cascade function is trained from several positive and negative images and then is used to detect objects in other images. Since we need to work with faces we exploit the pre-trained classifier contained in *OpenCV* that detects frontal faces [4].

Our project can either work on a RGB stream captured by a webcam or a video. For every frame considered we decide to skip the five subsequent frames with the aim of speeding up the execution.
Once selected the source of the stream, each considered frame is resized to the fixed dimension 240x320x3 if it comes from a webcam. To apply our face detector we need a grayscale image, so we duplicate the RGB frame and convert it to grayscale, then we use adaptive histogram equalization to improve the constrast. In particular we

[1] https://www.tensorflow.org/
[2] https://www.tensorflow.org/guide/keras
[3] http://www.vlfeat.org/matconvnet/models/vgg-face.mat
[4] https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml

adopt the *CLAHE*[5] (Constrast Limited Adaptive Histogram Equalization) method from *OpenCV*, which divides the image in blocks of 16x16 and each block is histogram equalized, then, to avoid the noise enhancing, a contrast limiting is applied. After equalization, to remove artifacts in blocks borders, it makes use of bilinear interpolation.
At this point the algorithm calls the function *detectMultiScale* from our Cascade Classifier on this image. If faces are found, it returns their positions as Rect(x, y, width, heigth) where (x, y) is the upper-left corner of the bounding box. With the aim of minimizing false positives, common in Haar features based classifiers, we choose to set some parameters: *minNeighbors* (how many neighbors each candidate rectangle should have to retain it) with value 10 and *minSize* (minimum possible object size) of (50, 50). Despite this, sometimes the method *detectMultiScale* still finds faces where there are none.
Afterwords, using each bounding box, the algorithm crops the original RGB frame obtaining a squared centered image of the corresponding face. Then we need to resize the face-image to 224x224x3 in order to extract the feature vector, using the *VGG* model created, that is the most expensive operation in the execution. When dealing with *VGG* is it possible to consider the lasts three layers as output, see Table 1.

Table 1. Last three VGG model's layers

Layer	Type	Output Shape
fc6	Convolutional2D	4096
fc7	Convolutional2D	4096
fc8	Convolutional2D	2622

We develop three versions of the same algorithm. For each of them the feature vector corresponds to a different output layer.
Now that we have a descriptor of the current face we want to estabilish if we have already seen the corresponding person.
At the end of the execution on every frame, we predispose a structure to store the bounding boxes and the corresponding ID of the faces processed. So the first thing we have to do is to check if there is overlap between the current bounding box and one of the bounding boxes of the previous frame through the *intersection over union* method. In particular, if the overlap is higher than 85% we assume that the person is one of the persons observed in the previous frame, so we proceed with *Structure Updating* of the corresponding ID with this new descriptor.
If overlap is not successfull or in the previous frame there are no faces, we have to check if the current descriptor corresponds to a person representation stored doing *ID selection* or, if not, instantiate a new person ID and then proceed with *Structure Updating*.
To compute re-identification, the algorithm makes use of a set of essential parameters that has to be updated after every iteration with *Tolerance Updating*.

[5]https://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html

Structure Representation

For each ID we associate some descriptors to keep track of faces with different expressions and under different light conditions, because we want the algorithm to be able to recognize already seen faces in the future. Whit this intent, we predispose a *python dictionary*, where the IDs represent the keys and the descriptors their values. We decide to set a maximum number of descriptors to each person ID. After some experiments we fixed this number to 15, as a compromise between representation's level of detail and realtime execution.

ID Selection

For each ID the algorithm computes and stores the distances between the current feature vector and the ID's stored descriptors. If one of these distances is higher than a maximum distance value (*dist_limit*), the function skips to the next ID, otherwise it saves the average between all the distances.
Once all the IDs have been considered, the algorithm selects the smallest average and the corrisponding ID. If the average is less than the ID's tolerance, we conclude that the current descriptor is referred to that ID. If the descriptor does not match with any person ID, the algorithm creates a new ID and associates to it this feature vector.

Structure Updating

Once the ID has been selected we proceed to update the structure. If the minimum distance between the current feature vector and the ID's descriptors is less than a minimum distance value(*min_dist*) associated to this ID, the descriptor does not add information to the current representation, and is discarded. Else the feature vector has significant information to add; in case the structure has not exceeded the fixed capacity, it is simply inserted, otherwise the algorithm computes a weighted average between the feature vector and the descriptor that has minimal distance from it. We decide to give little more importance to the old descriptor.

Tolerance Updating

As said we need tolerance parameters to verify if the feature vector under analysis is related to one of the persons' representations. Through a set of experiments, we observed that a fixed tolerance parameter does not work, neither works a solution that reduces by a factor the specific ID's parameter

each time a new descriptor referred to it is inserted.
We instead propose an adaptive updating method. This method needs some auxiliary parameters: *max_toll*, *min_distance* and some auxiliary variables for each ID: *toll*, *min_dist*, *mean_dist*.
Every time a new ID is created the algorithm initializes its *toll* to *max_toll*, *min_dist* to *min_distance* and put the average of these two in *mean_dist*. When a representation of an ID is updated, the method initially needs to compute the distances, taken pairwise, between the descriptors of that ID and saves their minimum(*m*), maximum(*M*) and mean value(*avg*). Then the function modifies the variables as follows:

$$min\_dist = m * w1 + min\_distance * w2$$

After some tests, we notice that it is necessary to update *toll* under the hypothesis that *M* is higher than *mean_dist* to avoid errors due to too few descriptors. In this weighted average we give importance to *max_toll* to prevent a fast decreasing of toll that could lead to errors.

$$toll = toll * w3 + M * w4 + max\_toll * w5$$

$$mean\_dist = avg$$

The result of the execution on each frame is displayed on video. Each face is enclosed in its bounding box with the estabilished ID.

3. Experiments

In this section we analyse the three version proposed, focusing attention on the auxiliary parameters related to *Tolerance Updating* and possible distances to adopt.

3.1. fc8 output layer

We first implemented the fc8 version. At the beginning we opted for the common *Euclidean distance* and *Cosine distance* to compute the similarity between descriptors. Then we noticed that the feature vector returned through fc8 output layer has a dimension of 2622 in which every element is a value between 0 and 1 and represents the probability that the considered person looks like a certain celebrity. So the whole vector can be interpreted as a probability distibution. For this reason we tried to use measures that work on probability distribution as *KullbackLeibler divergence* and *Chi-Square Distance*, but in the simplest cases these distances fail, in particular they assign the same ID to completely different persons.
Using the fact that the values of a decriptor are between 0 and 1, we then propose to apply the square root to them before computing the *Euclidean distance* to obtain an increment of the distances' values and an improvement on the re-identification operation.
We therefore work with *Euclidean distance*, *Cosine distance* and *Euclidean sqrt distance*. For each of them we set the auxiliary parameters and weights, established through a set of experiments, see Table 2 and Table 3.

Table 2. fc8 parameters

Distance	*dist_limit*	*max_toll*	*min_distance*
Euclidean distance	1	0.8	0.2
Cosine distance	0.4	0.317	0.07
Euclidean sqrt distance	4.9	3.8	1

Table 3. fc8 weights

Distance	w1	w2	w3	w4	w5
Euclidean distance	0.2	0.8	0.3	0.15	0.55
Cosine distance	0.2	0.8	0.25	0.15	0.6
Euclidean sqrt distance	0.2	0.8	0.3	0.15	0.55

Since the values of fc8 represent the probability to look like to 2622 celebrities, it is reasonable to think that these values are not significant to characterize a generic face . So we decide to consider two alternative outputs.

3.2. fc7 output layer

The fc7 output layer is the penultimate layer of the net. We implement this version with *Euclidean distance* and *Cosine distance*. Depending on the distance, we set the auxiliary parameters and weights, established through a set of experiments, see Table 4 and Table 5.

Table 4. fc7 parameters

Distance	*dist_limit*	*max_toll*	*min_distance*
Euclidean distance	1.3	1.25	0.3
Cosine distance	0.92	0.8	0.2

Table 5. fc7 weights

Distance	w1	w2	w3	w4	w5
Euclidean distance	0.2	0.8	0.3	0.15	0.55
Cosine distance	0.2	0.8	0.3	0.2	0.5

3.3. fc6 output layer

The last version proposed extracts the feature vectors from fc6 output layer and employs *Euclidean distance* and *Cosine distance* to do comparisons. For both metrics we set the auxiliary parameters and weights, established through a set of experiments, see Table 6 and Table 7.

Table 6. fc6 parameters

Distance	*dist_limit*	*max_toll*	*min_distance*
Euclidean distance	1.15	1	0.3
Cosine distance	0.7	0.55	0.15

Table 7. fc6 weights

Distance	w1	w2	w3	w4	w5
Euclidean distance	0.2	0.8	0.3	0.15	0.55
Cosine distance	0.2	0.8	0.3	0.2	0.5

4. Results

We test the project on specifically created videos. These videos are made in such a way that different people appear in front of the camera to be firstly identified and then reappear, under different light conditions or different distances, to be re-identified. To evaluate the best option, we consider the behaviour of the implementations in critical cases. We define two critical cases, first a situation in which a person, appearing under different or adverse light conditions, compared with its stored representation, can not be re-identified. The other critical case occurs when the descriptor of a person, appearing in the scene for the first time, is too similar to the representation of another ID and so the re-identification could fail.

Observing the behaviour of the three versions implemented and their variants, paying attention to critical cases, we can derive the following conclusions. First of all, we notice that the fc7 can not discriminate between faces of two different persons that have a high color similarity or look very similar.

As regards fc8, the best variant is the one that uses the *Euclidean sqrt distance*, since it allows to set the parameters in a such way as to avoid errors due to outliers. In the examined videos, the variant using *Euclidean distance* works but could fail on other examples, since its parameters are calibrated to solve the considered critical cases and there is low margin. The *Cosine distance* variant, instead, in certain videos misses some critical cases, so it is discarded.

No significant differences are found between fc6 variants, so both implementations are considered to be efficient for the re-identification purpose.

5. Conclusions

In this project we develop an algorithm to perform re-identification on videos, using information extracted by *VGG* convolutional neural network. We propose three different versions and their variants, focusing the attention on the way to do *Structure Representation* and *Structure Updating*. For each ID the system stores a maximum of 15 descriptors that have the task to characterize the ID's face with different pose, illumination and expression, adding new feature vectors or updating any of them, according to specific rules, during the execution.

We test all the variants on specifically created videos, observing that better results were given by the fc8 with *Euclidean sqrt distance* and both variants of fc6.

The three versions (*fc8, fc7, fc6*) of the system differ in the *VGG*'s output. Since the fc8 uses descriptors representing probabilities to look like celebrities and the fc7 is discarded, we select the fc6 as the best option. However, given that *Euclidean distance* is better suited in all versions, we establish the fc6 with *Euclidean distance* as the best variant.