Computer vision project

Coins classification

M. Colasuonno, F. Monelli and M. Mosconi

Summary

Goal…

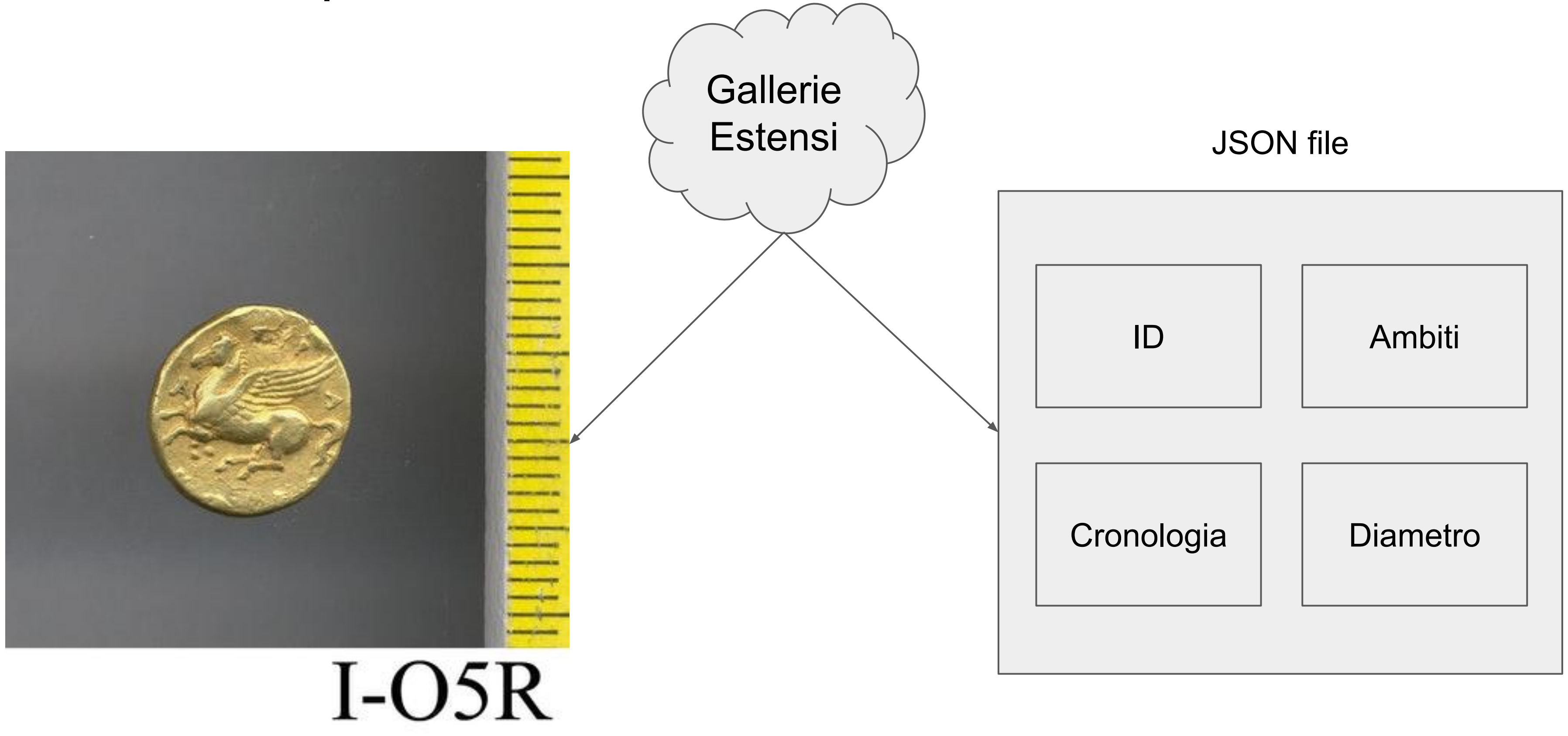
There are many coins not classified in Gallerie Estensi museum. To solve this problem, our goal is to giving informations about chronology and civilization(so the class) of coins given in input. Starting from the dataset of the museum we collect data and use them to achieve our goal, we do so by the development of a pipeline of operations through which each coin goes in the training and classification process.

Source:

https://www.gallerie-estensi.beniculturali.it/collezioni-digitali?oggetto=Moneta&perpage=30&page=1

GALLERIE ESTENSI

Dataset acquisition

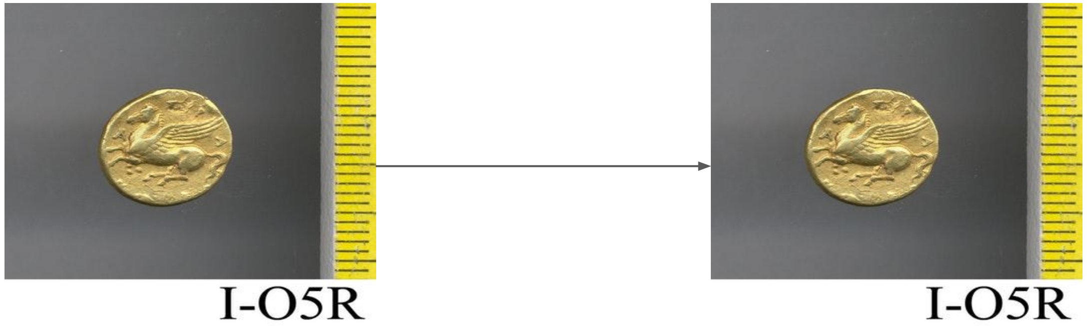


Labeling

After downloading the information from www.gallerie-estensi.beniculturali.it we started the data manipulation to create our dataset. We decided to rename the files with the information obtained from the json file.
More precisely:

- **ID**: represents the coin identification code in gallerie Estensi
- **Ambiti**: represents which population made the coin
- **Cronologia**: represents the period in which the coin was made
- **Diametro**: represents the diameter of the coin.

At this point we have renamed all images.



s-34868-8919.jpg 34868_ greca_(280 aC - 280 aC)_15,5.jpg

Image scraping and labeling: results

The numbers of images downloaded from www.gallerie-estensi.beniculturali.it is **113**. At the first moment, after the renaming phase, we noticed that there were a lot of images that represent the same classes but this wasn't correct because the attribute diameter of the coins represented is different.

Therefore in labeling phase we added the attribute "diameter", obtaining **100 different classes** from 113 images.

The libraries used to create our base dataset:

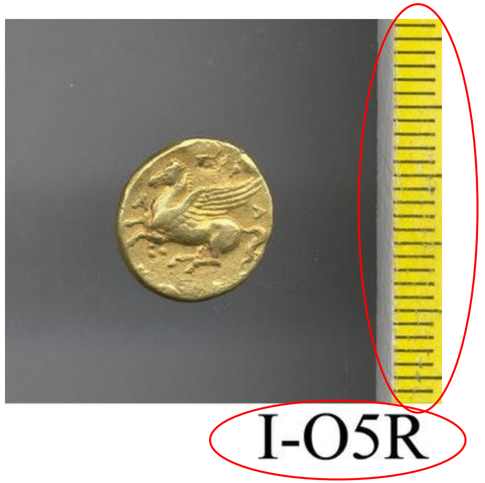
- **selenium**: used to be able to use a web server in python code
- **beautiful soup**: is a library that makes it easy to scrape information from web pages
- **urllib.request**: is a module that defines functions and classes which help in opening URLs in a complex world
- **json**: is a module that helps us to manage json file

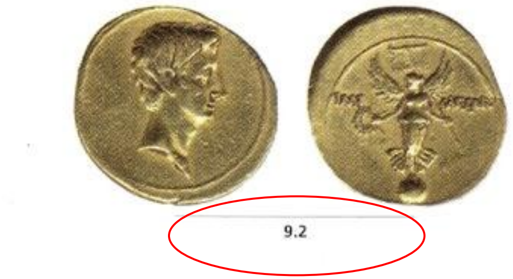
Beautiful Soup

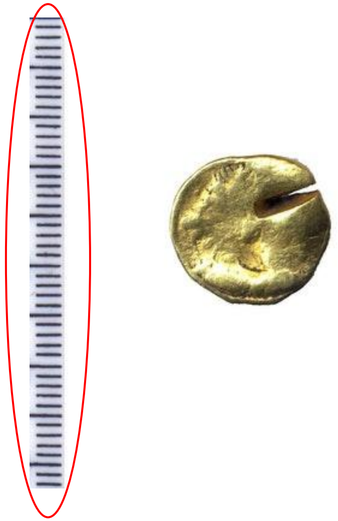
Selenium

Images analysis

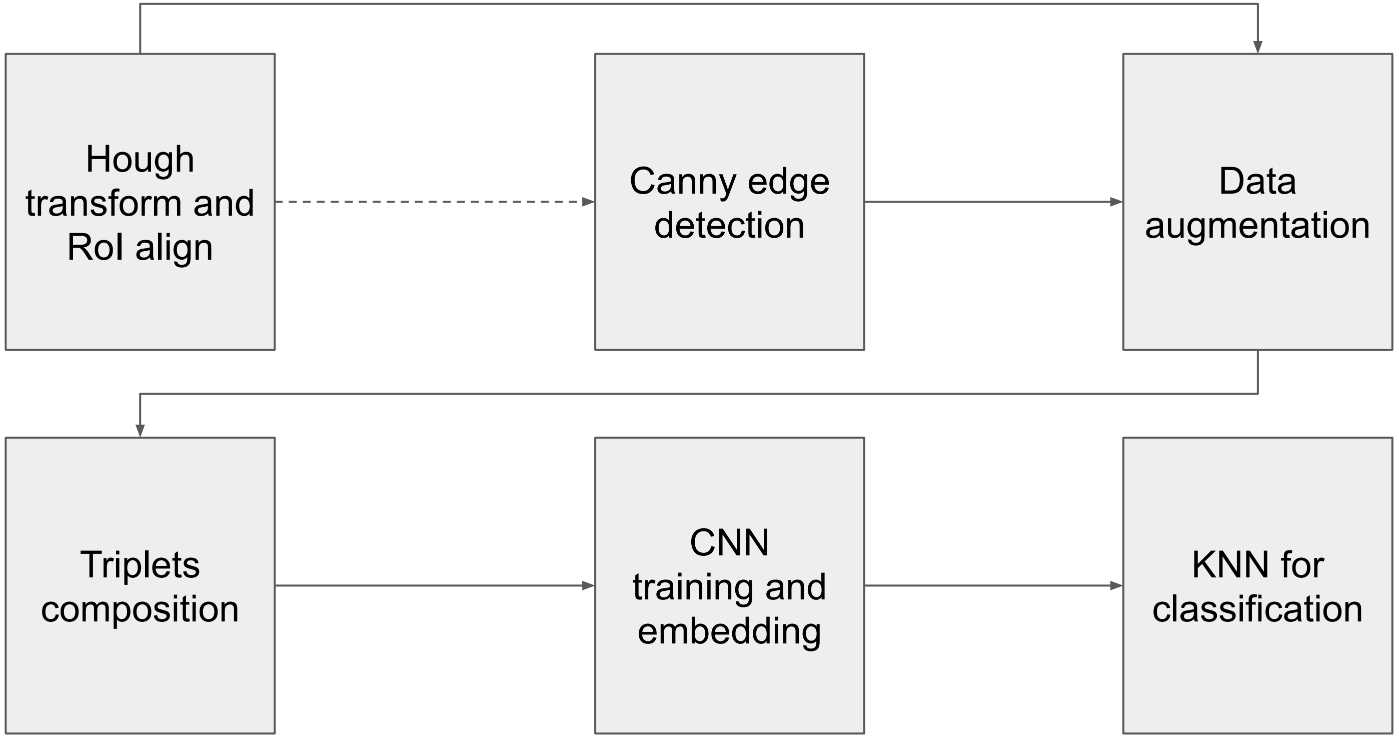
After a first visual analysis of the images of our dataset we notice that many of them have a lot of elements that for us are considerable perceptual noise(like some “id” labels, or even some measuring columns). We want to crop those elements out, obtaining just the coin for each image.







An ensemble vision of the pipeline



Hough transform and RoI align

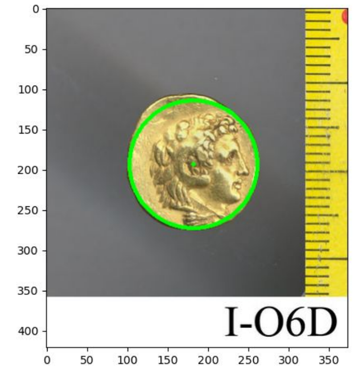
By performing some analysis on the position and the radii of the circles we discover that the maximum radius is 144 and the minimum is 24, so a considerable variation.

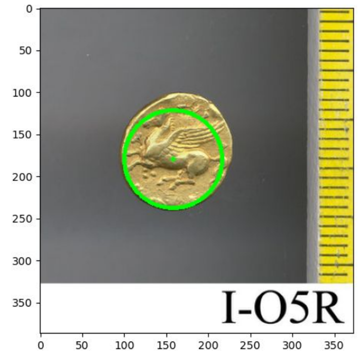
We want to have just the coin image for feeding the CNN and we also need a mini-batch of images, so they have to be of the same shape(possibly reduced, for computational reason), to achieve both of these goals we use **RoI align** in combination with the **Hough Transform**.

In particular we decide to use Hough Transform, not for its ability to draw circles but because it will tell us the center and the radius of each coin, with this information we can proceed feeding the RoI align with a perfectly located bounding box for each one.

We’ve decided to use RoI align to reduce the input size of the images exploiting the fact that it can act independently from the input size of the images, we use one single box for each image with a dimension that is calculated after the HT results, we choose 100x100 because it is an approximation of the average of the coins dimensions.

Some examples...

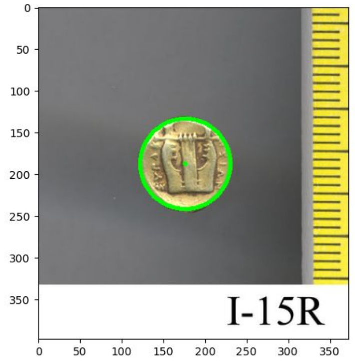


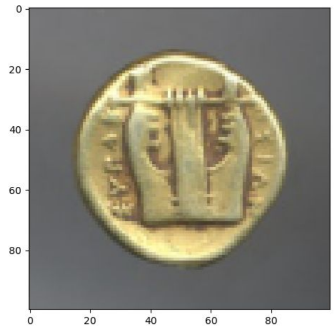


Hough Transform

RoI align

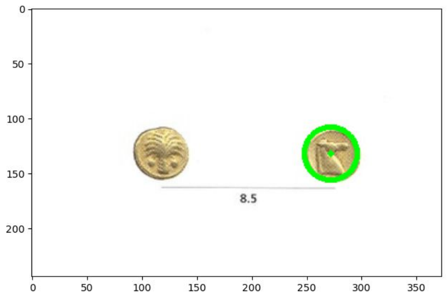
...and others

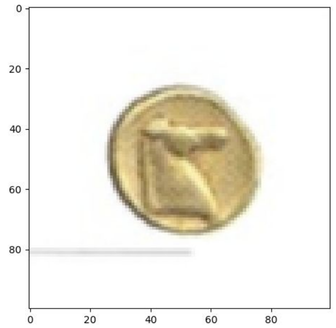




Hough Transform

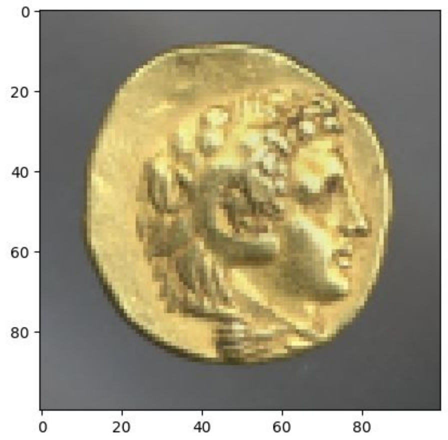
RoI align



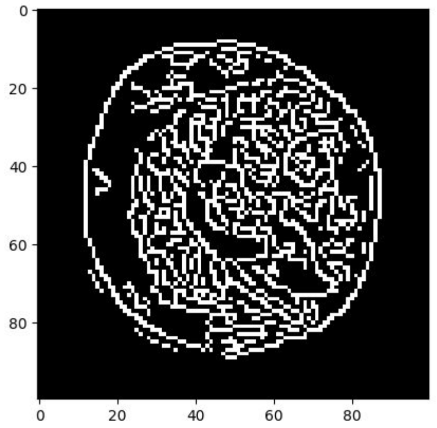


… and Canny edge detection

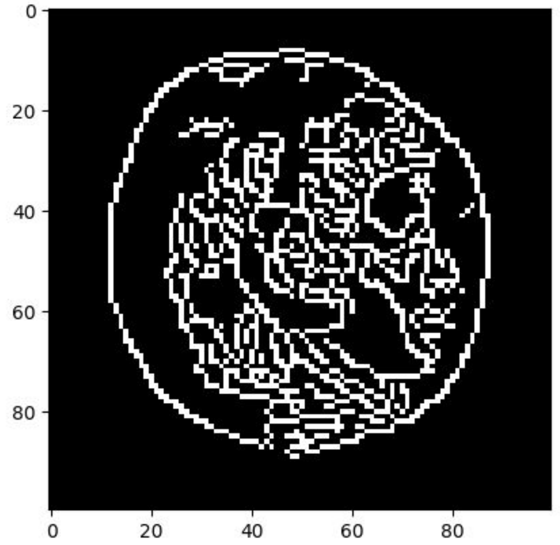
Input
image



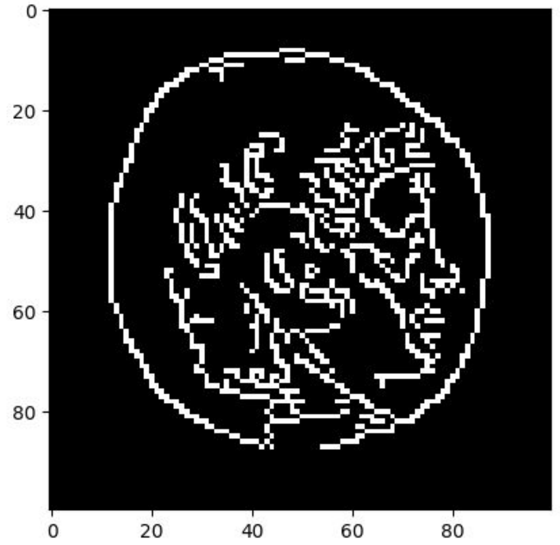
Canny
with a
50-150 as
threshold



Canny
with a
100-250
as
threshold



Canny
with a
200-350
as
threshold



Some considerations on Canny

In our first test Canny works in an unexpected good way, introducing first a great improvement in the accuracy performance, we are arrived to 99%, but there is a big problem that at a first glance we have not noticed.

In the following we say that we want a dataset which is augmented with some linear operators, the problem is that Canny produces an image which is black with the edges that are white and if you apply some linear operator you are not modifying a black and white map of this type, or at most you're modifying it in an inconsistent manner.

So it is not feasible to introduce it, because it would be like cheating due to the fact that even if you are taking images away for the test set, the network will still see some of them(or a version which is too similar to them) because of the inconsistency that the linear operator has on canny.

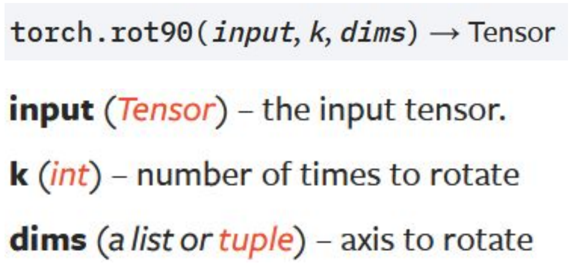
Of course if one want to use Canny, the thing can be done by taking in consideration the fact that a linear operator is dangerous because it can produce results in accuracy that are extremely good but at the end not reliable.

Image manipulation

The number of images was too low, it was necessary to perform a data augmentation.

The **first step** was to convert all 113 images in a tensor with shape [113,3,100,100] using the python code, that implements the hough transform and RoI align, that returns the tensor.

To perform data augmentation we have decided to apply a rotation on the tensor using the function torch.rot90().



`torch.rot90(`*input*, *k*, *dims*`)` → Tensor

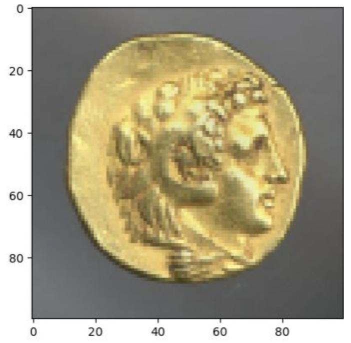
input (*Tensor*) – the input tensor.

k (*int*) – number of times to rotate

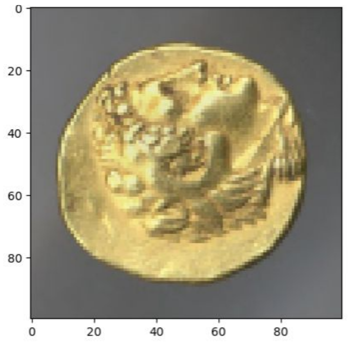
dims (*a list or tuple*) – axis to rotate

We apply this function 3 times (0° is the original image already present in dataset) to obtain a result tensor with shape [452,3,100,100] where 452 = 113 * 4.

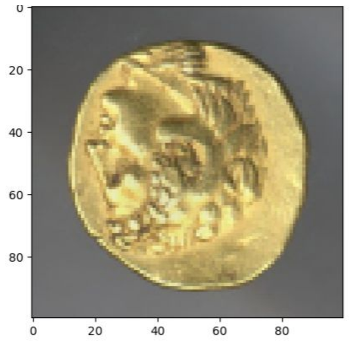
Tensor rotation: some examples



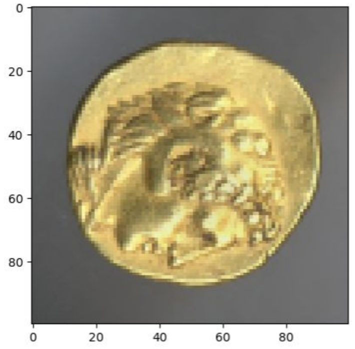
0°



90°



180°



270°

Images manipulation: linear operator

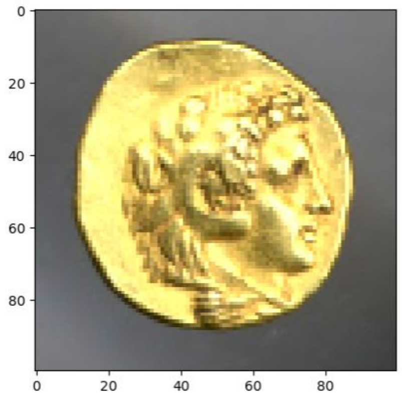
The second operator that we applied to increase the number of images is a linear operator.

$$I'(\boldsymbol{x}) = h(I(\boldsymbol{x})) = s\, I(\boldsymbol{x}) + k$$

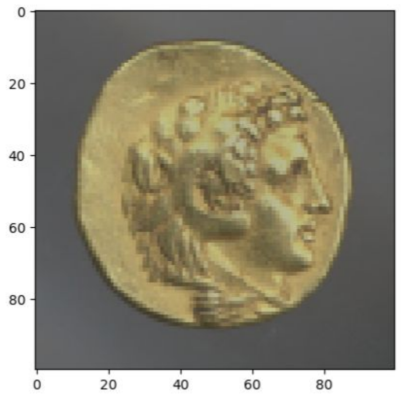
The shape of input tensor is [452,3,100,100], we apply linear operator twice to obtain a tensor that has a shape of [1356,3,100,100].
At this point our final dataset is composed by 1356 images arranged on the 3 colors channel with 100x100 dimension.

Original

s = 1.5 , k = -50

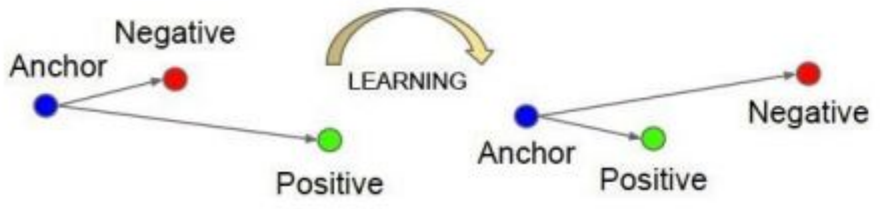


s = 0.7 , k = 20



Triplets Generation(1)

As can be seen from the “Ensemble vision” we have decide to use a triplet loss for training the CNN to become good as an embedding network for retrieval(even if at the end we use a KNN for the classification to achieve our final goal). In order to do training with a triplet loss we need triplets.



$$\|x_i^a - x_i^p\|_2^2 + \alpha < \|x_i^a - x_i^n\|_2^2, \ \forall \left(x_i^a, x_i^p, x_i^n\right) \in \mathcal{T} .$$

We start by considering the fact that we don’t care about the repetition of some triplets(even if very unlikely). So we start by selecting an anchor in a random way, then we continue with a positive of the same class but with another constraint, the fact the we don’t want the same instance of the same class, then we pick a random negative belonging to a random class(different from the anchor and positive class of course).

Triplets Generation(2)

Due to the fact that we have taken into consideration that we don’t want the same element in a tuple anchor-positive, we want to take as a maximum number of triplets that we can generate the number of the partial permutations of 12 in 2(n = 12, k = 2), because we have at least 12(11, if we don’t consider the instances for test cases) instances of objects of the same class. This number is equal to 132(110).

If we multiply the number obtained by the number of classes that we have(100) we obtain 13200(11000) which is the maximum number of different triplets that we can obtain considering all the things that we have just said(to be precise this would be the minimum of the maximum number of triplets, because some classes have more than one instance for each class in the original dataset composed of 113 images, in fact we have 100 classes).

We choose to implement a function that ask the number of triplets that you want, in this way we can perform many experiments with different numbers of triplets.

We notice that the improvements stop after a number of **10800 triplets**(this number is totally empiric and derives from our experiments).

Embedder and KNN

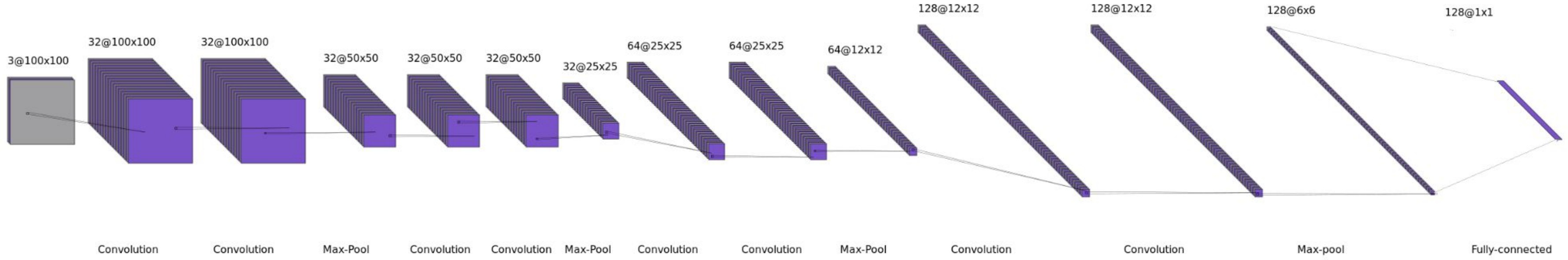
We have choose to use a deep learning approach for the part of the features extraction, in particular we have used a CNN inspired from **VGG**, however the problem is concerning our dataset which is quite small, consequently the network will be absolutely shallow both in term of layers and in term of filter learned for each layer.

The details of the network will be showed in the Technical implementation section.

Starting from a 100x100(3 channels, RGB) image we will arrive at the end of the network having an embedded vector 128-dimensional, with this vector(and many others) we will train **K Nearest Neighbor**; When trained this will map in a 128-dimensional space each vector(with their labels associated) which will be feed into it.

At the end the KNN will tell us, after the feeding of a test vector, which is the nearest class for it, in particular with a majority voting of the K nearest points, we choose a K of 5.

Network architecture

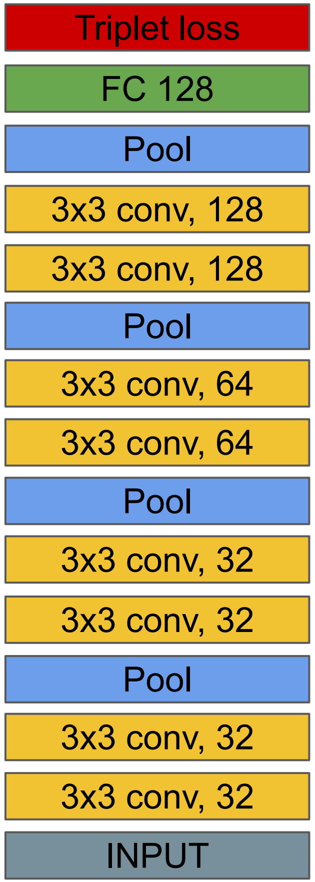


Model and parameters

Input: [100x100x3]	memory: 100*100*3 = 30K	params: 0
Conv3-32: [100x100x32]	memory: 100*100*32 = 320K	params: (3*3*3)*32 = 864
Conv3-32: [100x100x32]	memory: 100*100*32 = 320K	params: (3*3*32)*32 = 9216
Pool2: [50x50x32]	memory: 50*50*32 = 80K	params: 0
Conv3-32: [50x50x32]	memory: 50*50*32 = 80K	params: (3*3*32)*32 = 9216
Conv3-32: [50x50x32]	memory: 50*50*32 = 80K	params: (3*3*32)*32 = 9216
Pool2: [25x25x32]	memory: 25*25*32 = 20K	params: 0
Conv3-64: [25x25x64]	memory: 25*25*64 = 40K	params: (3*3*32)*64 = 18432
Conv3-64: [25x25x64]	memory: 25*25*64 = 40K	params: (3*3*64)*64 = 36864
Pool2: [12x12x64]	memory: 12*12*64 = 9216	params: 0
Conv3-128: [12x12x128]	memory: 12*12*128 = 18432	params: (3*3*64)*128 = 73728
Conv3-128: [12x12x128]	memory: 12*12*128 = 18432	params: (3*3*128)*128 = 147456
Pool2: [6x6x128]	memory: 6*6*128 = 4608	params: 0
FC: [1x1x128]	memory: 128	params: 6*6*128*128 = 589824

Total memory = 1.06 M * 4 bytes = 4.2 MB/image

Total number of params: 894816



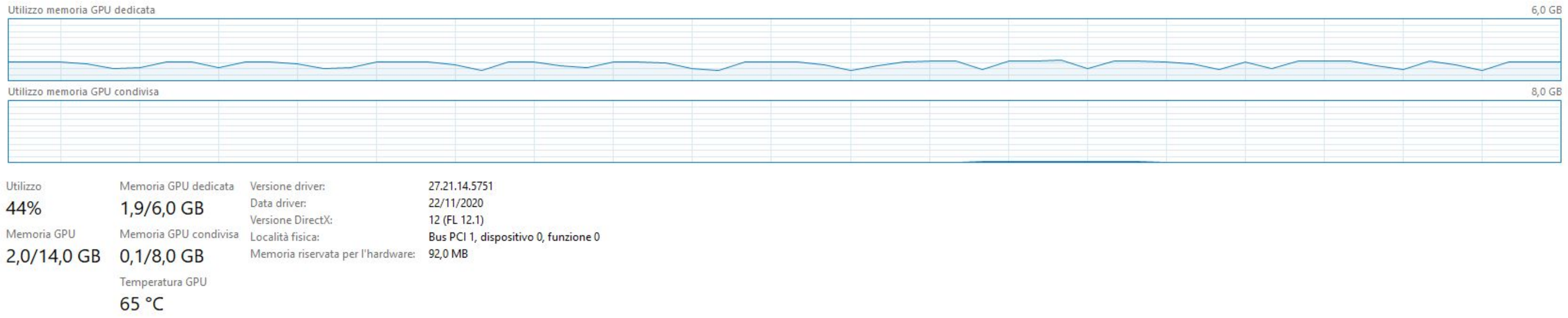
Training

Due to the shallowness and compactness of the network we decide to train it on a NVIDIA GTX 1060, which has 6GB of VRAM, clearly not a GPU born for deep training or for scientific purpose but it fits our case of use.

The main thing that we have to set for what concerns hyperparameters is the margin of the **Triplet loss**. We found out, also using some empirical strategy found online(stackoverflow.com, discuss.pytorch.org), that a good strategy is to set the margin in a way in which half of the triplets are already solved(so the loss gives 0) and the other half is not.

The other important thing to take into account is the number of epochs that we want to perform in our loop training cycle and according to this the learning rate, more on this in the evaluation section.

For what concern the memory consumption: on the GPU we are talking about 2 GB of total memory without particular optimization, deleting input tensors and freeing the cache at each iteration bring us at a maximum of 1.9 GB. For the KNN instead we have used the CPU, so we are talking about classical RAM, we use a total of 5 GB.



Utilizzo
44%

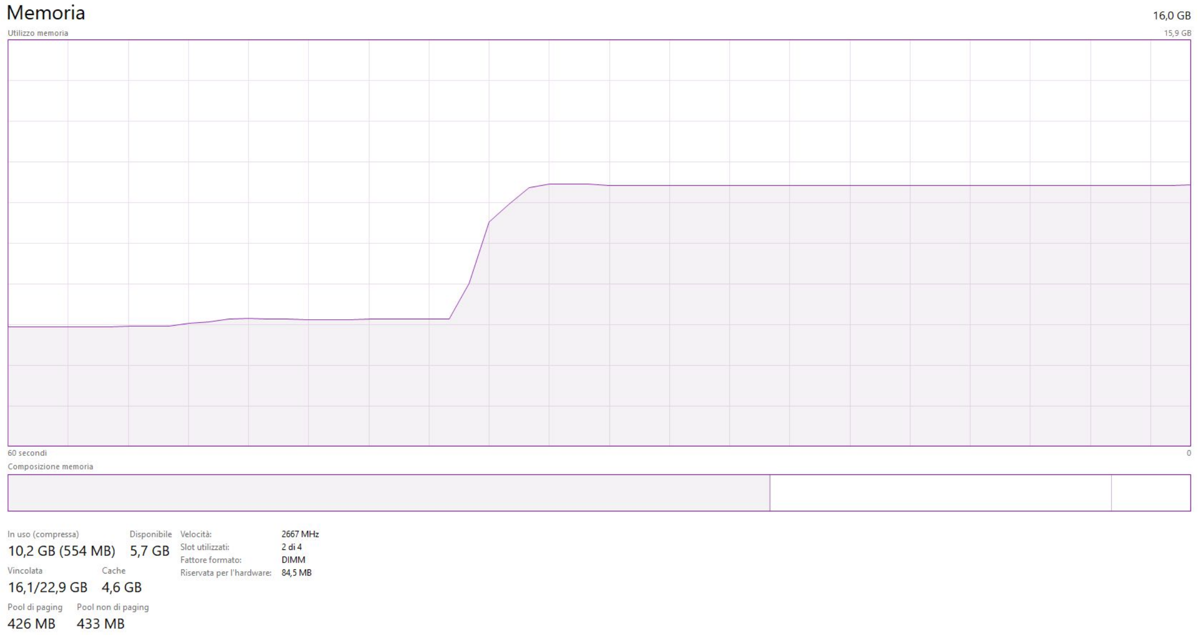
Memoria GPU
2,0/14,0 GB

Memoria GPU dedicata
1,9/6,0 GB

Memoria GPU condivisa
0,1/8,0 GB

Temperatura GPU
65 °C

Versione driver:	27.21.14.5751
Data driver:	22/11/2020
Versione DirectX:	12 (FL 12.1)
Località fisica:	Bus PCI 1, dispositivo 0, funzione 0
Memoria riservata per l'hardware:	92,0 MB



In uso (compressa)
10,2 GB (554 MB)

Disponibile
5,7 GB

Vincolata
16,1/22,9 GB

Cache
4,6 GB

Pool di paging
426 MB

Pool non di paging
433 MB

Velocità:	2667 MHz
Slot utilizzati:	2 di 4
Fattore formato:	DIMM
Riservata per l'hardware:	84,5 MB

Evaluation and results

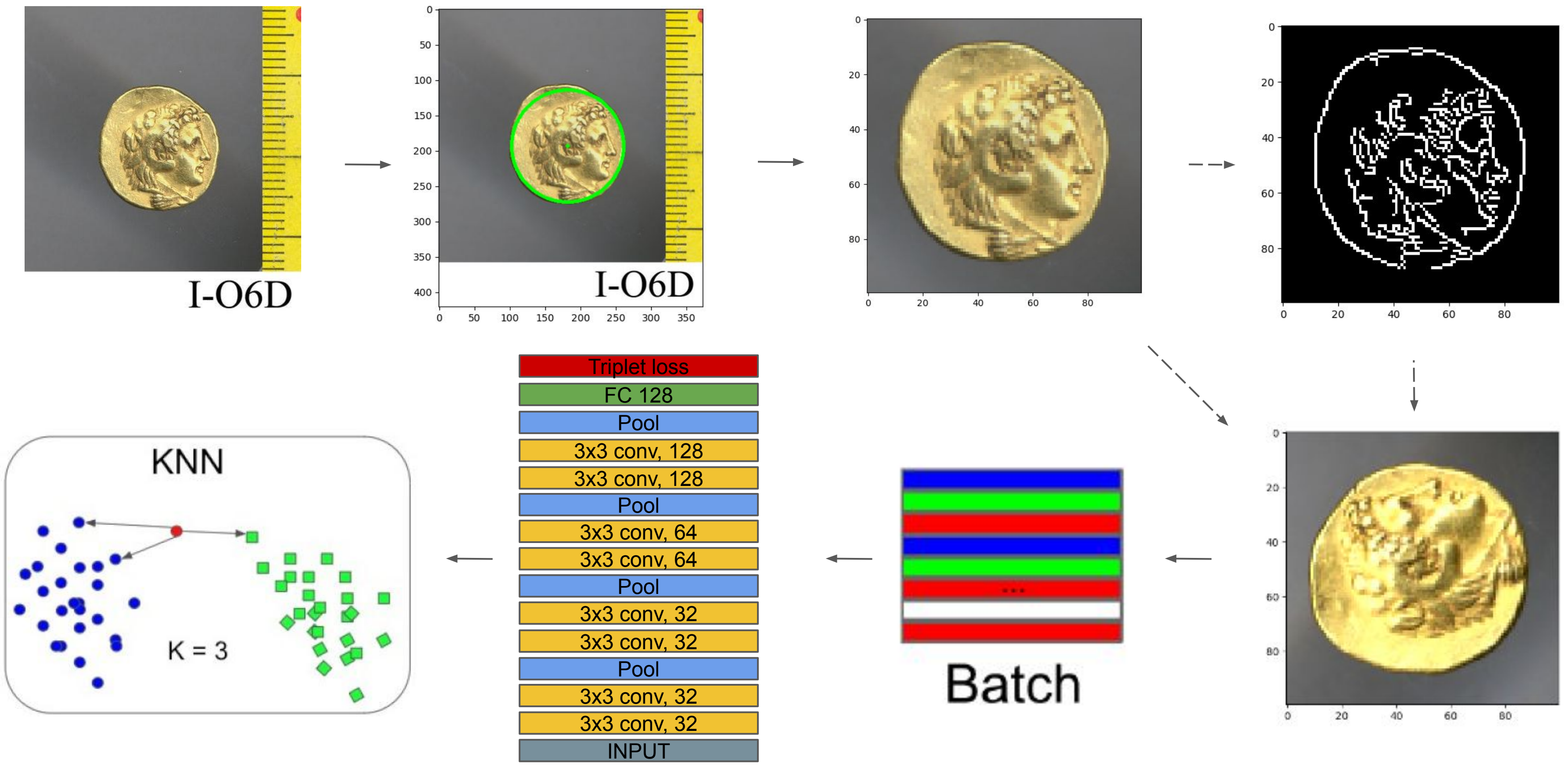
For the evaluation part our plan is to generate another batch of triplets(different to the one used in the training of the CNN part), this for increasing the randomness and the correlation between data, in other words to be less prone to overfitting.

We proceed then by feeding them to the already trained CNN that has just to perform one forward pass, this is the embedding part in which we end up with **10800**(this is the number of triplets that we have chosen to generate) vectors of 128 dimension.

As the last step we train, and test immediately after, the **KNN** and evaluate the accuracy given by different iterations of this same procedure, this because, due to the fact that we are sampling random triplets each time, the accuracy can vary accordingly.

We were able to achieve, with a number of 9600 epochs and a learning rate of 0.001, a maximum accuracy of **92.9%**, with an average of **91.3%** measured on 5 iterations.

All the story



Some documentations used

- Vassileios Balntas, Edgar Riba, Daniel Ponsa and Krystian Mikolajczyk. Learning local feature descriptors with triplets and shallow convolutional neural networks. September, 2016.
- Florian Schroff, Dmitry Kalenichenko, James Philbin. FaceNet: A Unified Embedding for Face Recognition and Clustering. 17 June 2015.
- https://pytorch.org/docs/stable/index.html
- https://pytorch.org/vision/stable/index.html
- https://discuss.pytorch.org/
- https://stackoverflow.com/

Thank you for the attention!