

## Deep Learning for Content-Based Image Retrieval in Biomedical applications

**Auteur :** Schyns, Axelle

**Promoteur(s) :** Maree, Raphael; Geurts, Pierre

**Faculté :** Faculté des Sciences appliquées

**Diplôme :** Master en ingénieur civil en informatique, à finalité spécialisée en "intelligent systems"

**Année académique :** 2022-2023

**URI/URL :** <http://hdl.handle.net/2268.2/17731>

---

### Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

---

# Details of the open-source Implementation

Axelle Schyns

June 9, 2023

## 1 Intro

The implementation can be found in <https://github.com/AxelleSchyns/cbir-tfe> with the code files being placed in the DATABASE folder. The SCRIPT folder contains automatized scripts that were used during the thesis to run several methods at once. The DATA\_VISUALIZATION folder contains different graphs and files summarizing the data (samples from each class, histogram,...). The folder CMS finally contains all the graphs generated for the models.

To explain quickly the code files, there are in total 16 python files in the DATABASE folder. Some of them are used for tasks outside of the CBIR framework itself, others only consist in functions called in other files and do not have a main function (i.e. cannot be directly run). Here is a quick description of the content/use of each file.

- ADD\_IMAGES.PY: Code to run when indexing images into Redis. Requires to have an already trained feature extractor.
- AUTOENCODERS.PY: Contains the architecture or links towards the architectures and loss functions relative to the implementations of the autoencoder models.
- CLASSIFICATION\_ACC.PY: Computes the metrics for a classification model (Important to note that this code was done early in the thesis and not reran later on. Its functioning is not guaranteed but it provides a great base if the need were to arise again to deal with classification models).
- DATASET.PY: Contains the elements relative to the creation of the dataset such that it can be exploitable by the training, indexing and search methods.
- DB.PY: Contains all the methods related to the database and indexing structures: initialisation, indexing, search, training of the index. It is the file that must be run in order to train the index. It requires to have already indexed the dataset in the database and in the index.
- KMEANS.PY: Contains all methods linked to the K-means methods, be it for training, loading the model or analyzing the quality of the clusters formed.
- LOSS.PY: Contains the definition of all the losses used for the training of the feature extraction models.
- MODELS.PY: Code to run when wanting to train a new feature extractor. It contains the initialisation of the architecture and downloading of the pretrained weights as well as the different training concepts implemented.
- REC\_IMAGES.PY: Code used to generate the reconstructed images of an autoencoder model.

- RESNET.PY AND VGG.PY: Architectures of the models of the first implementation of the Autoencoder.
- RETRIEVE\_IMAGES.PY: Code to run to display the 10 most similar images to a query given as argument. Requires to have an already trained feature extractor as well as having indexed the data in the index and database.
- TEST\_ACCURACY.PY: Code to run to obtain the results of a CBIR framework. It has the same requirements as the previous file.
- TSNE.PY: Code to run to perform t-SNE on the vectors obtained using a chosen feature extractor. Requires to have previously computed the vectors and indexed them in the database and index.
- UTILS.PY: collection of functions used in the entire project.
- VISUALIZATION.PY: Functions to compute/display different characteristics of the dataset.

The document does not explain how to run the code, as it is directly explained on the Github page. This document is used to provide more details on some parts of the implementation that could be more difficult to grasp or that are not directly understandable from the code.

## 2 Dataset

The separation of the dataset in train - indexing - query sets was made in the original dataset per class (i.e. each class has been separated into 3 parts and the individual parts were then regrouped to form one big train superset, test superset and validation superset containing each all classes). Hence the absence of methods in the code to format the data.

For the correctness of the methods, two small modifications were applied to that original dataset.

- Images of height = 1 pixel in the class 0 of project janowczyk6 have been removed (68). Their flat dimension would make the application of the data transformations impossible and eventually lead to inconsistent results from the models.
- All the classes of the project umcm colorectal in the validation set were empty. To be able to compute accuracy results for all classes, one image of the test set has been selected and placed in the validation set.

## 3 Methods

### 3.1 Training

While the approaches used in the thesis are fundamentally different, they still share some common elements/options in regards to their training. Those elements comprises the scheduling approach, the hyperparameters and the number of epochs on which to train the models.

The optimizer used is the Adam optimizer while three options are available for the scheduler:

- None: The learning rate is not modified over time and stays constant.

- Exponential: The learning rate is obtained by multiplying the initial learning rate at each epoch by a tunable parameter to the power related to the epoch number. Mathematically<sup>1</sup>:

$$lr_t = lr_0 \cdot \gamma^{t-1}$$

- Step: Reduces the learning rate by multiplying it by gamma at the milestones, here after half the epochs and at the last epoch. Mathematically:

$$lr_t = lr_0 \text{ for } t \in [0, nb\_epoch/2]$$

$$lr_{nb\_epoch/2} = lr_0 \cdot gamma$$

$$lr_t = lr_{nb\_epoch/2} \text{ for } t \in ]nb\_epoch/2, nb\_epoch]$$

$$lr_{nb\_epoch} = lr_{nb\_epoch/2} \cdot gamma$$

Depending on the elements used (in particular the loss), different hyperparameters can be tuned. It is important to note that they were not considered at all in this study and default values have been used for all. The learning rate and the weight decay are the only parameters with  $\gamma$  for the scheduler, used in all the implementations and have the respective default initial value of 0.0001 and 0.0004.

One more tunable parameter for the training is the number of epochs during which the model is trained. As for the other parameters, it was not tuned at all and chosen by rule of thumb, taking into account the training loss and the time one epoch would take. It thus varies between 5 and 100, with 100 only for the fastest models (autoencoders) and more usually 15-20 epochs for the rest.

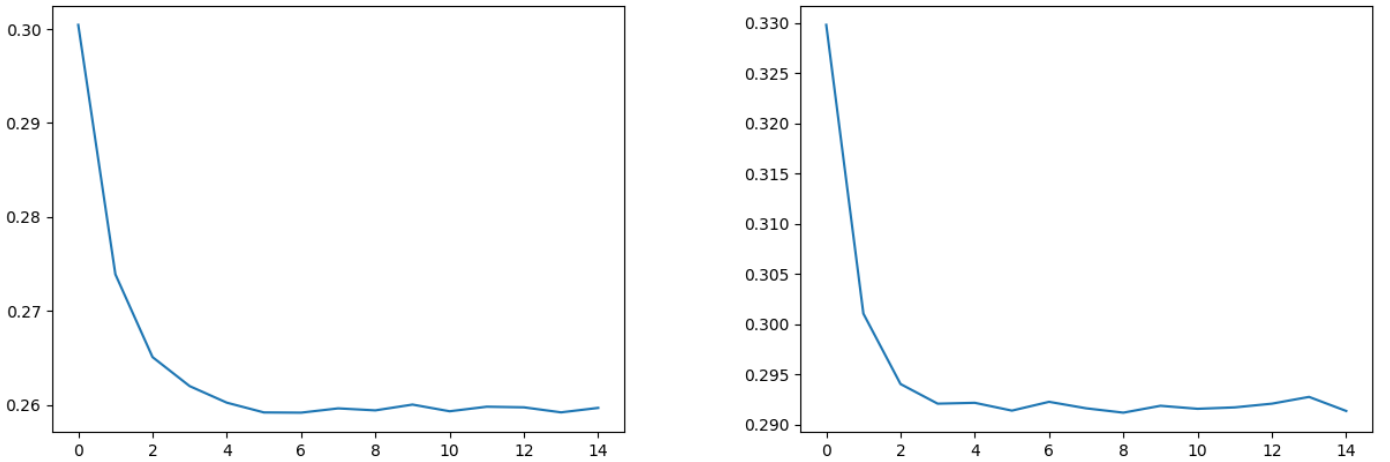


Figure 1: Training loss over 15 epochs for a Resnet Model with Margin loss

Finally, the size of the batch considered is tunable. Several batch size have been tested to see their effect on the training time and the accuracy. This parameter has also been tested in unison with the testing of the parallelism of the architectures and its effects. For hardware reasons, the most used batch\_sizes have been 256 when trained in parallel and 32 when trained on only one GPU, to keep the values of the previous thesis implementation.

<sup>1</sup>Based on the implementation of Pytorch exponential scheduler.

## 3.2 Supervised models

All supervised models are classification models that were modified to be used as feature extractors. In the `MODELS.PY` file, the second part of the model initialization function consists in replacing the last layer of all the supervised models by a new fully connected layers whose output number is the number of features wanted. All the architectures have different name for that last layer, explaining why the same line of code was not used for all but rather adapted to each case.

Note that the VGG19 architecture as well as the `inception_V3` architecture are available for use in the implementation but their results are not discussed in the master thesis because they were not of interest. Same for the second implementation of the Convolutional transformer as well as the SwimFormer architecture that are available for use in the implementation but are not discussed in this work as they don't bring anything more than the others.

The two last models (DeiT and ViT) require the use of a specific structure for data preparation (but whose effects are the exact same as for the other models) as well as a supplementary processing if the models are taken pre-trained and such that the obtained weights are frozen. This is due to a different library from which the models were retrieved.

## 3.3 Self-supervised models

### 3.3.1 Autoencoders

Each implementation of the AE has been subject to several experiments, leading to difference in the implementation architecture given the experiment. To use the architecture specific to an experiment, the value of the `EXP` parameter must be manually changed before running the code to the correct experience number. Note that that value might have to be set several times in different cod.: For implementation 1, it must be set in the main method as well as in the `resnet` and `vgg` files. For the specific case of experience 7, it must also be changed in the `utils.py` file in the `encode` function. For implementation 2, in the model initialization (class `VAE`) as well as in the loss right below. For implementation 3, it must be set in the model initialization (`AutoEncoder`) as well as in the two methods used for the loss right below.

Note that for implementation 1, when loading the model after having trained it, the load is slightly different from the load of the other models because it was not trained using this implementation but an external one. Hence the use of the specific load function.

### 3.3.2 Kmeans

A few important aspects had to be taken into account when training the K-means model/

- Due to the amount and type of data, the Kmeans algorithm had to be adapted such that it would only consider smaller groups of data one at a time. Indeed, the data could not be retrieved in its entirety at the same time as it is too big to fit in the memory. Therefore, MiniBatch K-means was used instead. It consists in updating the clusters centers by using a batch of new data at each iteration instead of the entire data set.
- To limit the time taken by the models, as the new labels only need to be generated once, they are saved in a pickle file and downloaded every time a model has to be trained using those labels. Given the random ordering of the files, it is checked that the correct file name is associated to the correct new label. If no label exists for a file, then only that file's label is generated. Similarly, the Kmeans model is also saved, first for reproducibility and comparability between models, but also to save the time it takes it to train (around 40 min on the setup described in 5.8). Due to that saving/loading, it was not built such that it uses/is optimised for GPU use.

Regarding the sampling, while a non random ordering had been first used for simplicity of the reproducing, it would have let to clusters being based on one class only (first batch is only composed of samples of one class) and then the other classes being forced into those clusters. Balanced sampling might have been better than the selected random sampling as it would have forced the clusters to be defined based on all classes. However, it would have been more complex and would have ruined the unsupervised aspect of this method as the past labels would have been considered to create the informative batches.

## 3.4 Indexing

### 3.4.1 FAISS

A FAISS index is easily created, it just requires to instantiate the chosen index obtained from the FAISS python library by giving it the size of the vectors that will need to be stored in it, and allocate memory, again through the use of functions of the interface. This operation creates a local file that can be later accessed to reload at will the created index.

Once created, the index is wrapped into another structure, `INDEXIDMAP`, that allows to map the FAISS ids to ‘external’ ids, *also integers*. This mapping is important as, in case of removal of an image, the FAISS ids will be changed but not the ‘external’ ids which allows to keep the coherence with the Redis database without necessitating to change all other ids. Finally, a vector is added to the structure by simply calling the `ADD_WITH_IDS` function of FAISS, with as arguments the vector and the ‘external’ id.

### 3.4.2 Redis

The database is accessed by starting a Redis server, connecting to it through the port 6379 and indicating the index of the database of interest (0 by default). An entry is added to it by using the function `SET` and retrieved by using the function `GET` of the Redis python library.

Two types of mapping are stored in Redis.

- The first one is the classic case, used in all models but the Kmeans procedure. The mapping consists in one ‘external’ id linked to the corresponding filename. Note that the filename contains the class and project name to which belongs the image. It is important because it allows to not add these information separately in the mapping and in the database (those information being important for the evaluation). Two entries are created per mapping, one where the key is the filename, and one where the key is the id. Fast retrieval is needed both when using the filename (in case of removal or retrieval of the corresponding vector) and when using the id (similarity search).
- The second case is when K-means is used. In that case, as the new labels are not contained in the filename, they must be added to the mapping. Hence, the mapping contains the filename, the new label and the id. Again, two entries are created for each mapping, with the id and the filename being the respective key. However, as Redis only allows a unique value per key<sup>2</sup>, a json structure is created as follows: `[{'name': val1}, {'label': val2}]` and used as value when the id is the key. When the filename is the key, only the id is the value as the label is useless in the situations calling for that access.

In addition of the mapping entries, the value of the next available index is also added to the database, with key `last_id`.

---

<sup>2</sup>When using key-value. Also the reason why *Sorted index and hashing* was thought of but finally discarded as it was overkill.

### 3.5 Search

Here, two indexes are used: INDEXFLAT and INDEXIVFFLAT. By default, INDEXFLAT is used, wrapped into INDEXIDMAP as explained before. However, INDEXIVFFLAT can also be obtained by using the train method in the database.py file. An INDEXFLAT structure must first be created in order to be used as argument for instantiating the other structure. It is then trained using clustering methods on the images of the dataset to lead to more efficient indexing. This index cannot be used without having been trained before. Furthermore, using the data to train the index does not automatically add it to the index. It must be added ‘manually’ afterwards.

#### Changes

This section will quickly describes some of the changes made in this implementation compared to the previous implementation to explain the discrepancies that could be found between the two codes.

First, the original implementation used a text file in addition to FAISS and Redis. This file contained the mapping between the filenames and the features vectors directly. It was used when training the FAISS index, probably due to the complexity of directly retrieving the vectors from the index. A first modification made had been to change the text file into a binary because the text file was saturating when a higher number of features was used and/or more vectors. It was then completely removed as the vectors could be retrieved directly from FAISS, in order to simplify the code and optimize the number of structures used.

Second, two FAISS indexes were used as well as two different ids for Redis, one attributed to labelled data and the other for unlabelled data. In addition to the structures, all the methods were also duplicated and the keywords ‘labeled’/‘unlabeled’ were added to the ids in all the entries of the Redis db. Besides that keyword in the variable names, all methods and entries were identical (same code, same exact effect/behavior). It seems like the labelled method was the only one used but the unlabelled had been designed in case unlabelled data would be used in the future. However, as the labels are not involved in this process in any way (at the exception of the kmeans), this duality was discarded in this master thesis in favor of more simplicity (especially in the search method that had to combine both db) and a more condensed code.