ImaginAItion

Eugenio De Luca

Multimedia Data Management, University of Bologna

August 23, 2024

Outline

Table of Contents

Introduction

ImaginAItion is an app I developed as the final project for the Multimedia Data Management course (MDM) at the University of Bologna. The main idea was having a multimedia app that was able to:

- Accept an image as input
- Segment the relevant subjects within it
- Add it to the database if the user requests it
- Query said database to retrieve "similar" images

While this kind of software is not new and there are plenty of similar apps, the idea behind this project was to use deep learning features to achieve the similarity search and the image segmentation.

Table of Contents

The main window - Overview

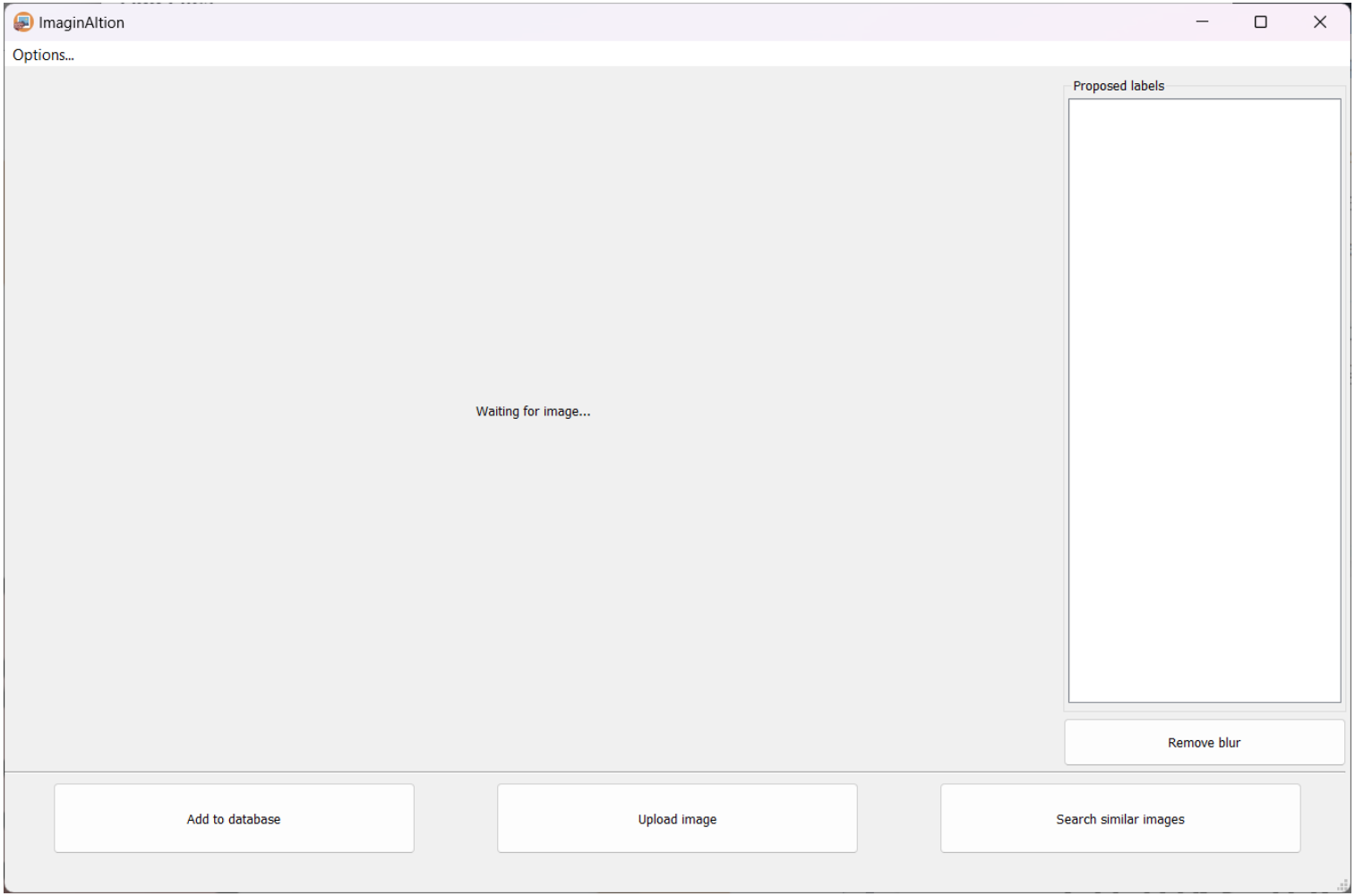

Figure: Main window on startup

The main window - Buttons

The main window features 4 buttons:

- **"Upload image"** button: Is used to select an image from the file system and add it to the application. After the image has been loaded the **"Proposed labels"** section will be populated by the classes that were detected by the DeepLab model; clicking on them will highlight the correspondent segmentation mask on the image.
- **"Add to database"** button: This button will add the current image and its relevant data to the database.
- **"Search similar images"** button: Will query the database with the current image.
- **"Remove blur"** button: Will clear the currently shown segmentation mask

The main window - Loaded image

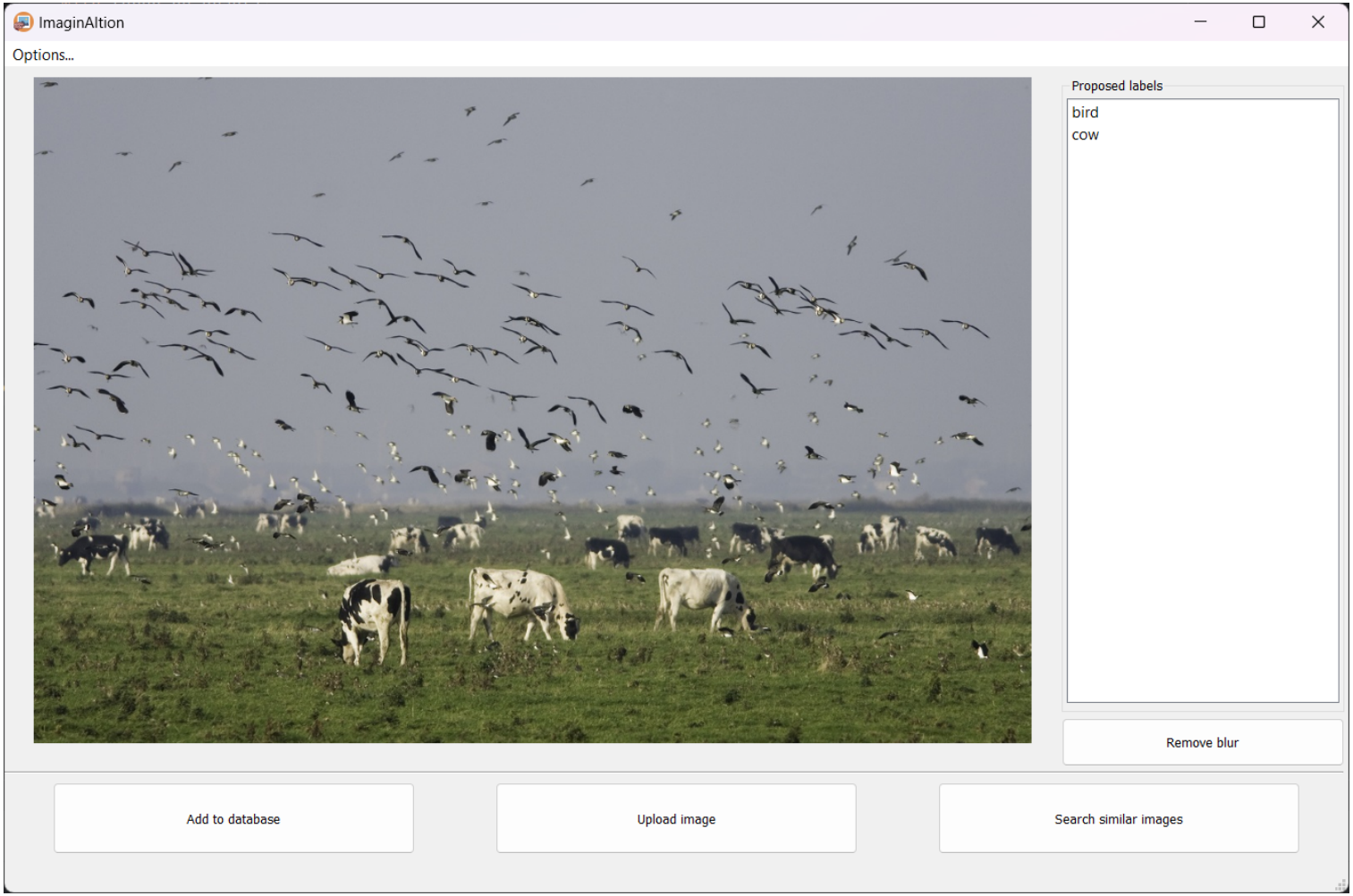

Figure: Main window after an image has been loaded. DeepLab detected cows and birds.

The main window - Segmentation mask

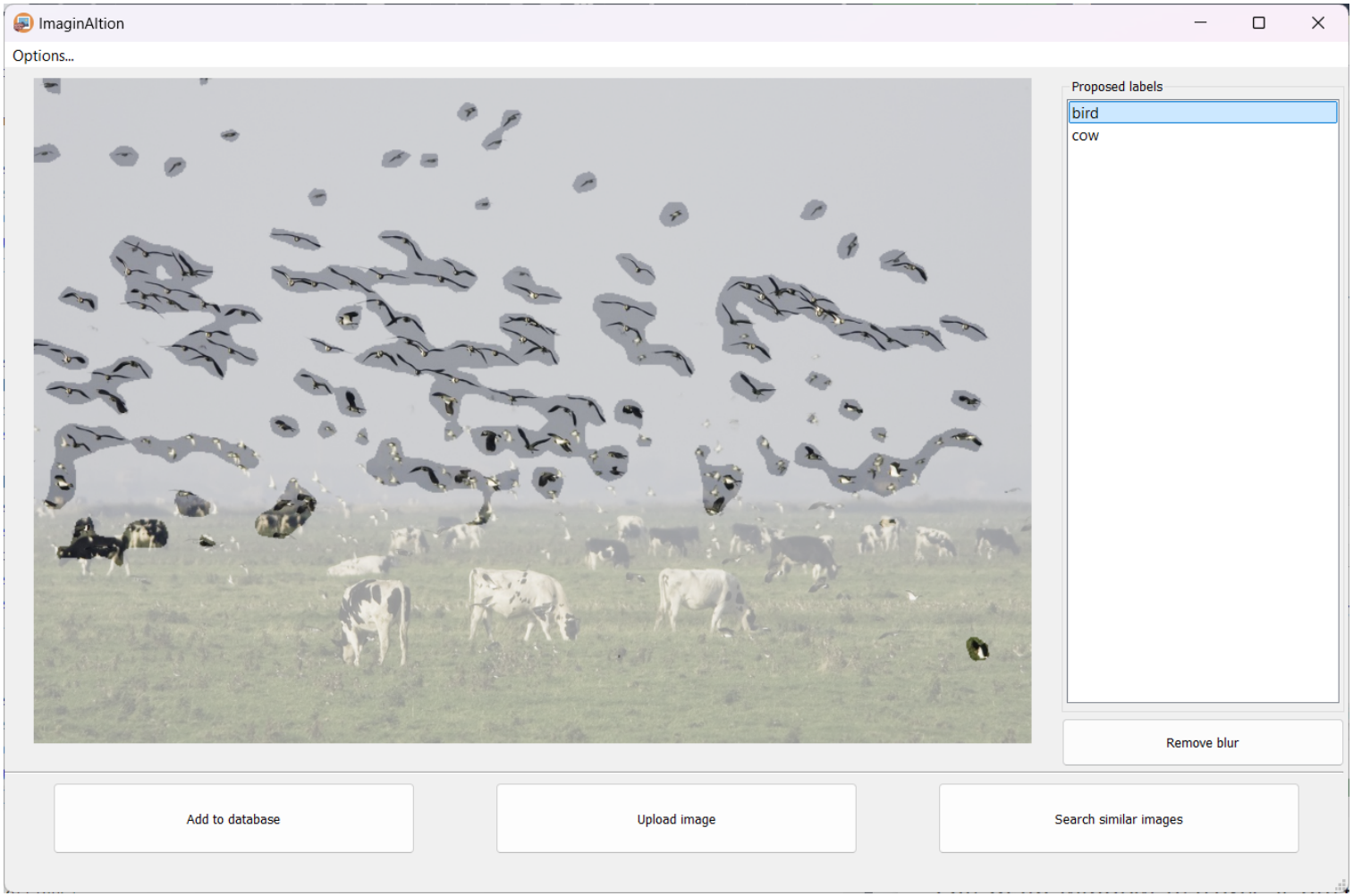

Figure: Clicking on "bird" on the **Proposed labels** list will show the segmentation mask for that class.

How to open them

The user can reach the settings window by clicking **Options** and then **Preferences** from the main window

Settings - The window

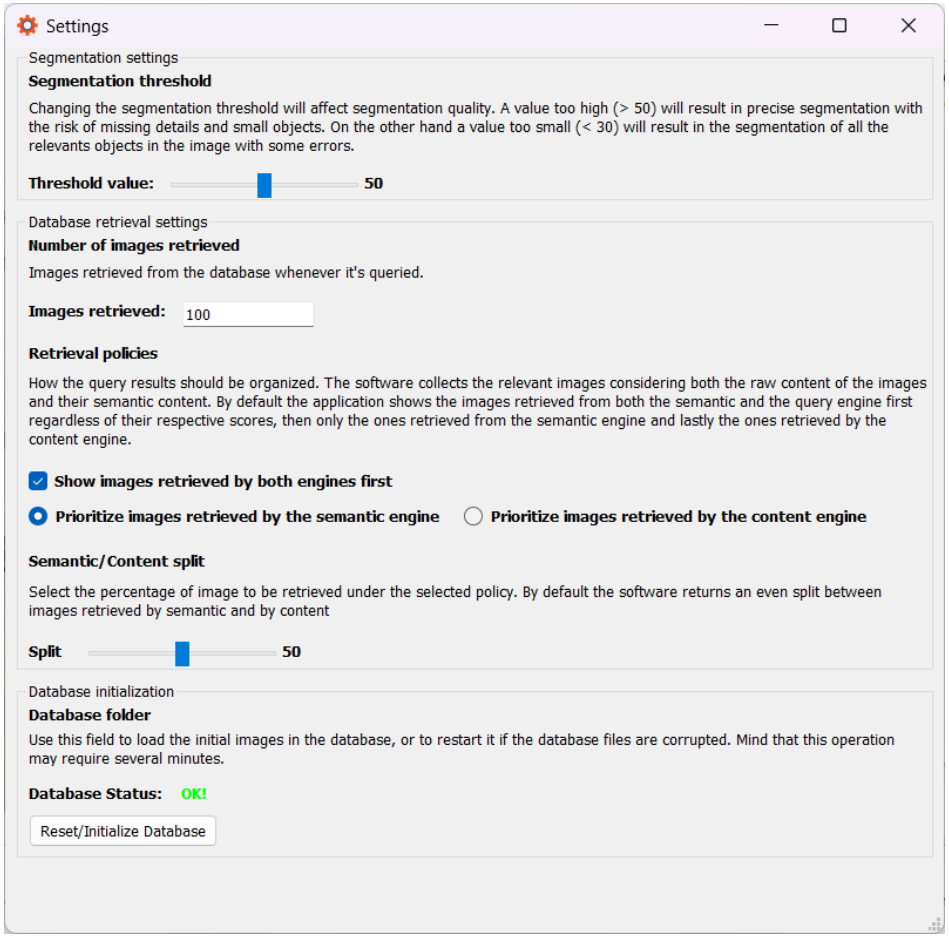

Figure: Overview of the settings' window.

Settings - What can the user tune?

From the settings the user can:

- Edit the segmentation threshold used to produce the segmentation maps from DeepLab's output. An high value will result in very precise segmentation masks that will miss several small details; on the other hand a low value will result in imprecise segmentation masks that will likely detect all the objects from a certain class.
- Decide how many images will be retrieved by a single query.
- Decide if the database should prioritize images retrieved by both the content and semantic indexes. By default the query processor will put on top images that appear in both indexes' results.
- Decide whether to prioritize the semantic retrieved images first or the content retrieved ones. By default the settings prioritize the semantic retrieved images because they tend to be more reliable.
- Decide the split between content/semantic retrieved image. By default the split is 50/50 but it can be tuned to the user's preference.
- Reset or initialize the database.

Resetting/Initialize the database

After clicking the **"Reset/Initialize the database"** in the settings window the user must choose a folder from their filesystem that **contains only images**. Upon pressing the reset button the application will initialize and index the database with the images contained in the folder; this process may take several minutes depending on the number of images in the folder.

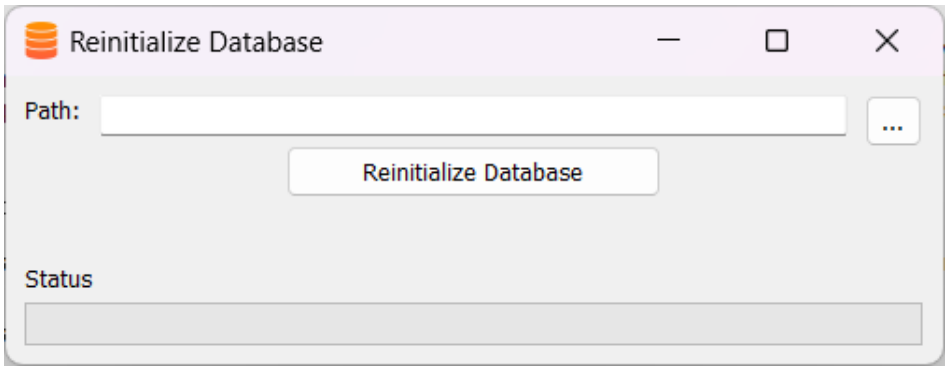

Figure: The reset database window

Behavior

Once the users clicks on the **"Search similar images"** button in the main window the result will show up according to the user's preferences. The application color codes the result to inform the user on how they were retrieved:

- **Blue**: For images retrieved by both indexes.
- **Green**: For images retrieved by the semantic index.
- **Red**: For images retrieved by the content index.

The results are sorted with decreasing similarity scores.

Examples - Images retrieved by both indexes

Figure: Example of images retrieved by both indexes

Examples - Images retrieved by the semantic index

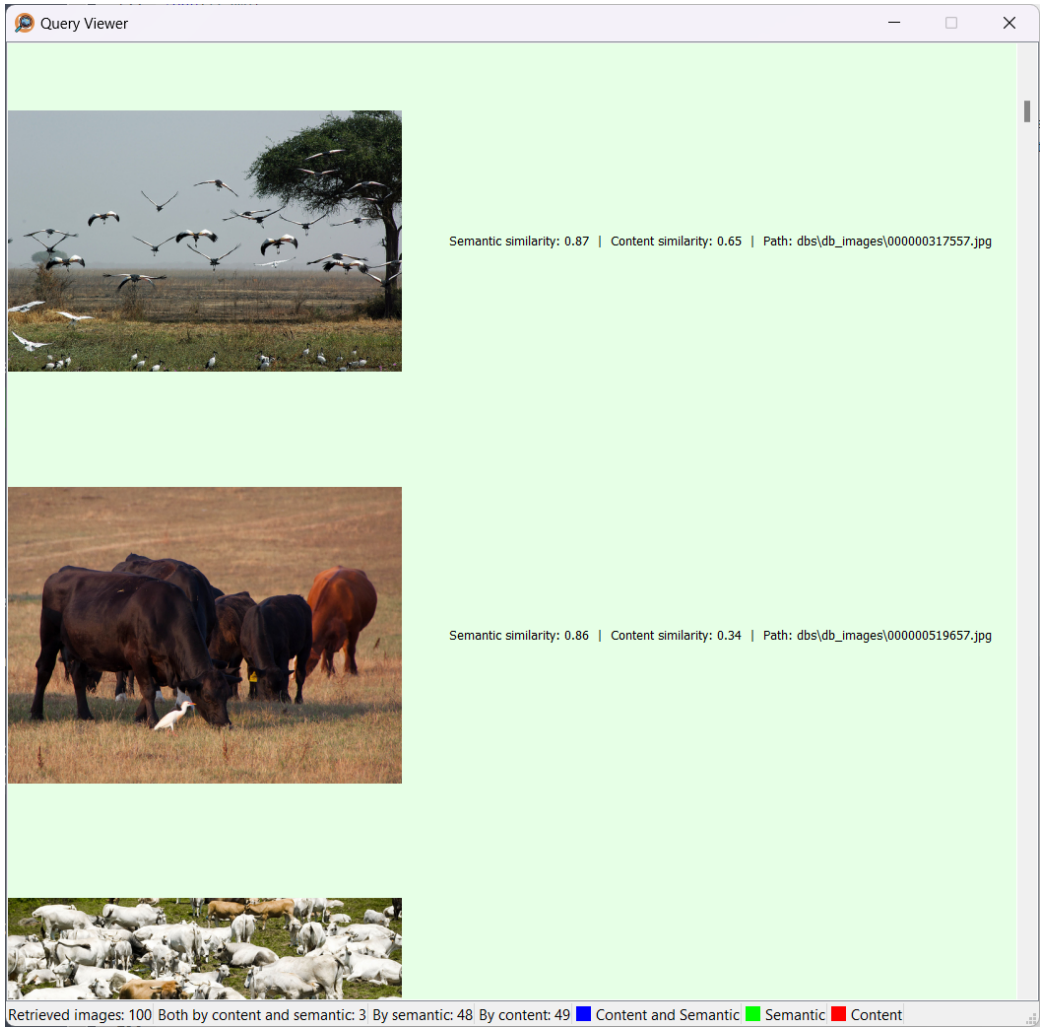

Figure: Example of images retrieved by the semantic index

Examples - Images retrieved by the content index

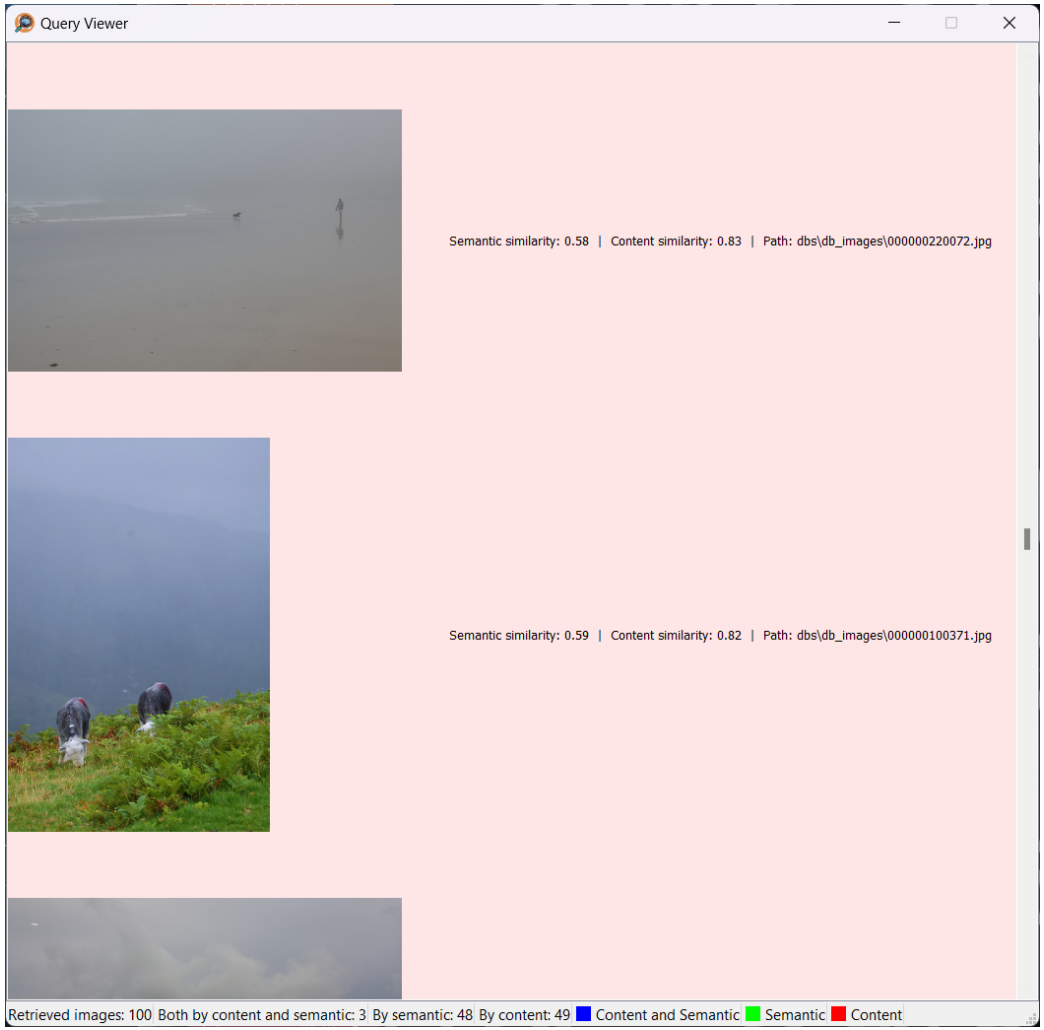

Figure: Example of images retrieved by the content index

Table of Contents

Loading an image

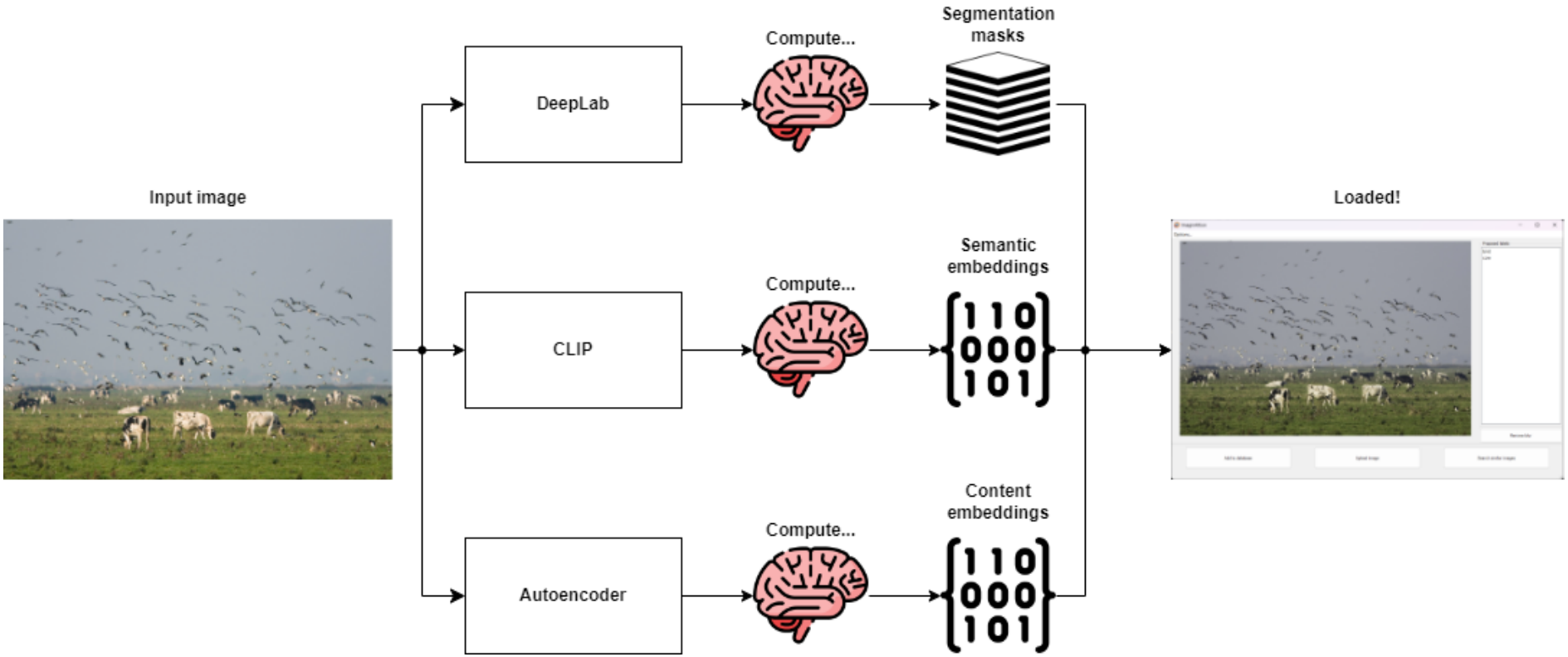

Figure: Loading an image

Saving an image in the database

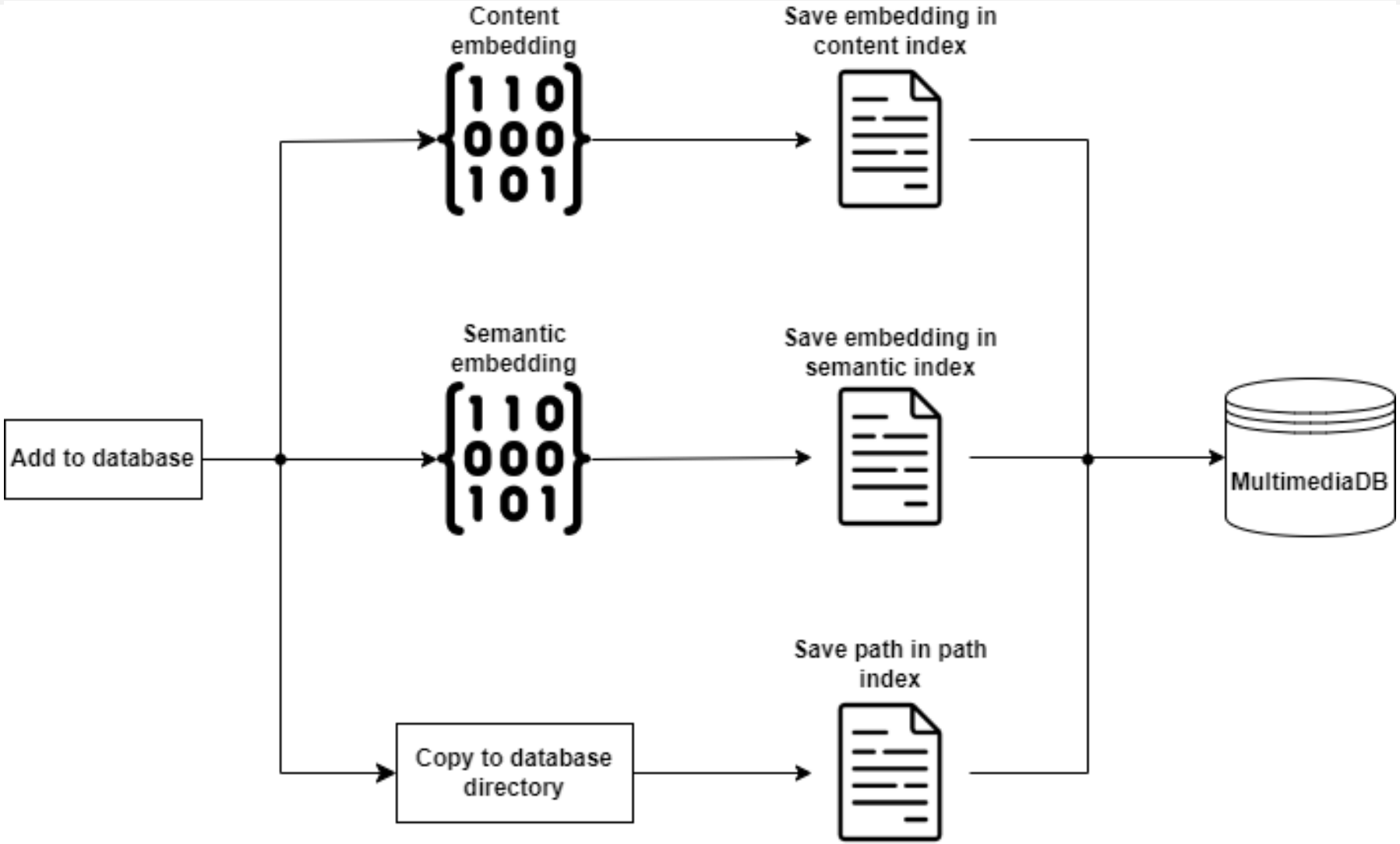

Figure: Saving an image to the database

Searching similar images

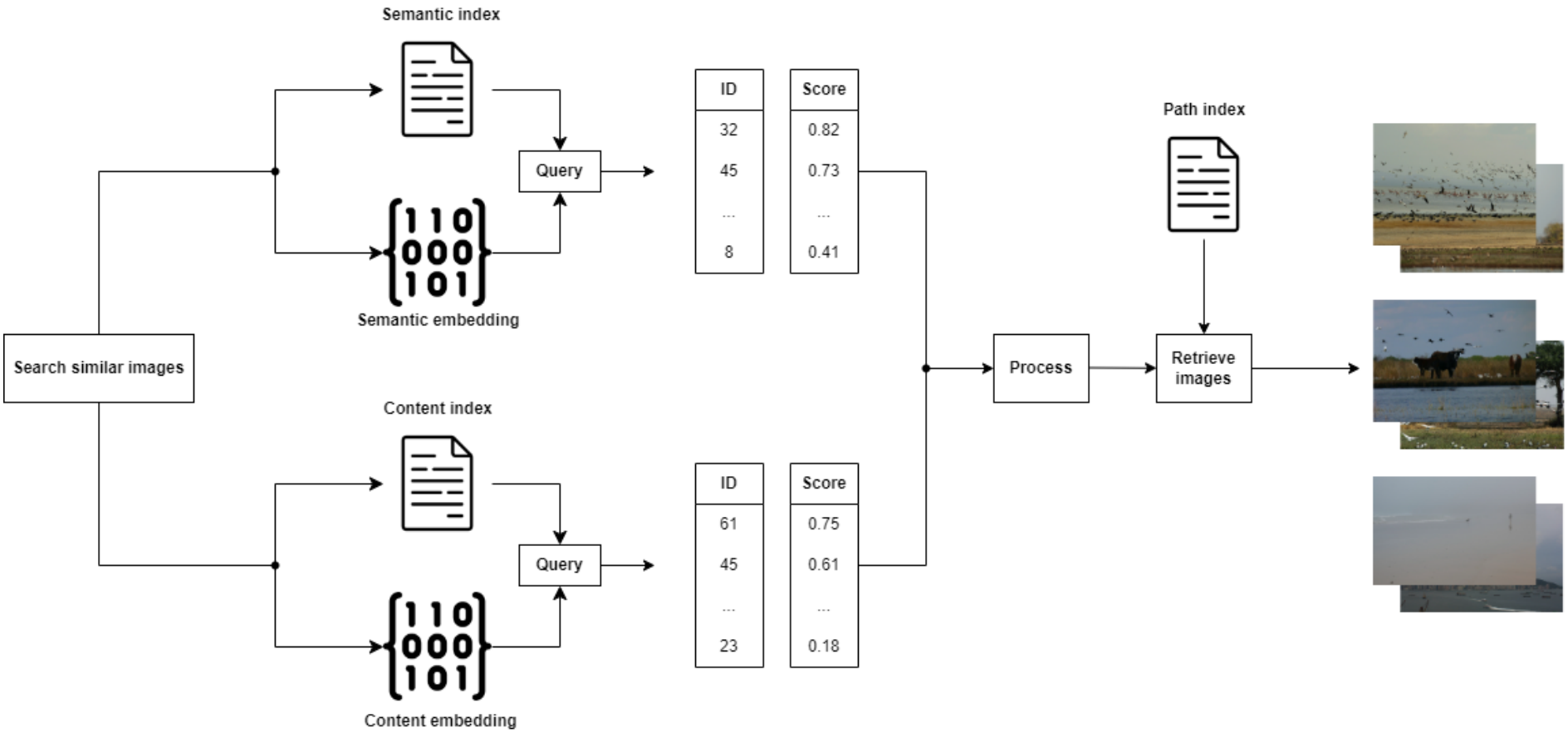

Figure: Querying the database

DeepLabv3

DeepLab is a convolutional architecture for semantic image segmentation published in 2017. It's composed by an architecture that takes care of elaborating the image to estract deep features (backbone) and an architecture that elaborates these features at different resolutions and concatenates them to express the final segmentation masks; this final layer is called ASPP (Atrous Spatial Pyramid Pooling layer).

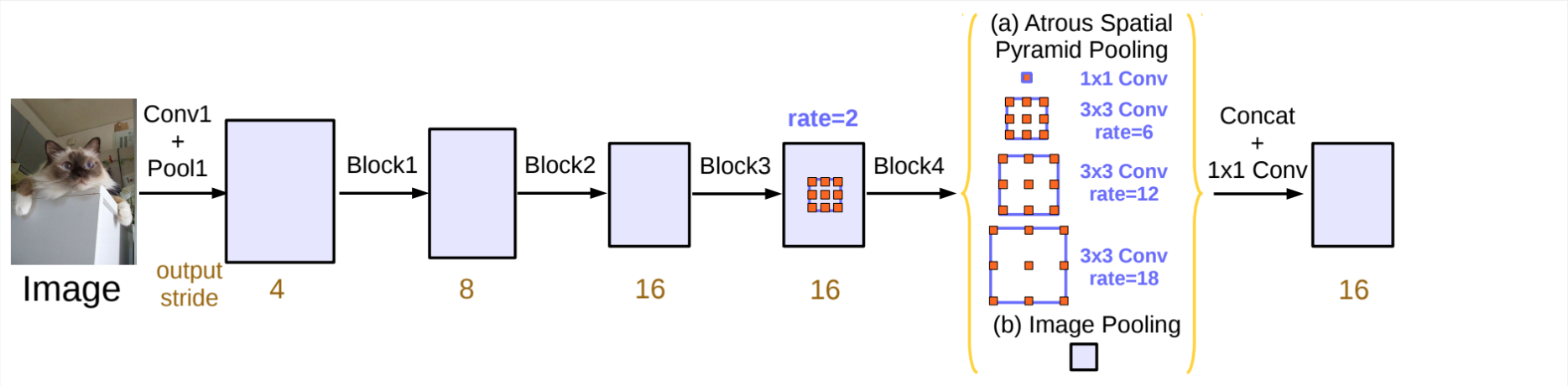

Figure: DeepLab architecture

DeepLabv3 - Implementation

DeepLab is a very big model that is very hard to train due its dimension without the proper hardware. To use it I loaded a pre-trained version powered by pytorch. The model uses ResNet101 as a backbone and is trained on the Pascal VOC dataset, and is therefore able to classify 20 different classes; these classes are fairly wide and common categories like cat, dog, sofa, car, etc...

CLIP

CLIP is a model architecture that is trained on images with language supervision by OpenAI. The big difference between CLIP and other architectures is the way in which it was trained. Normally convolutional networks are trained with data in the form of $(image, target)$ where the target can range from a categorical label, a numerical value and even another image (think about the denoising process for example, we input in the model the noisy image and we give as the target its denoised version!).

CLIP - Training

CLIP is instead trained by:

- Taking a vision architecture pre-trained on a classification class and removing the classification head (output) from it. This is called **Image Encoder**
- Taking a text encoder, so an architecture that given a text returns a vector representation of it.
- The image encoder is then trained to maximize the dot product (cosine similarity) between the image embedding and the text embedding.

CLIP - Example

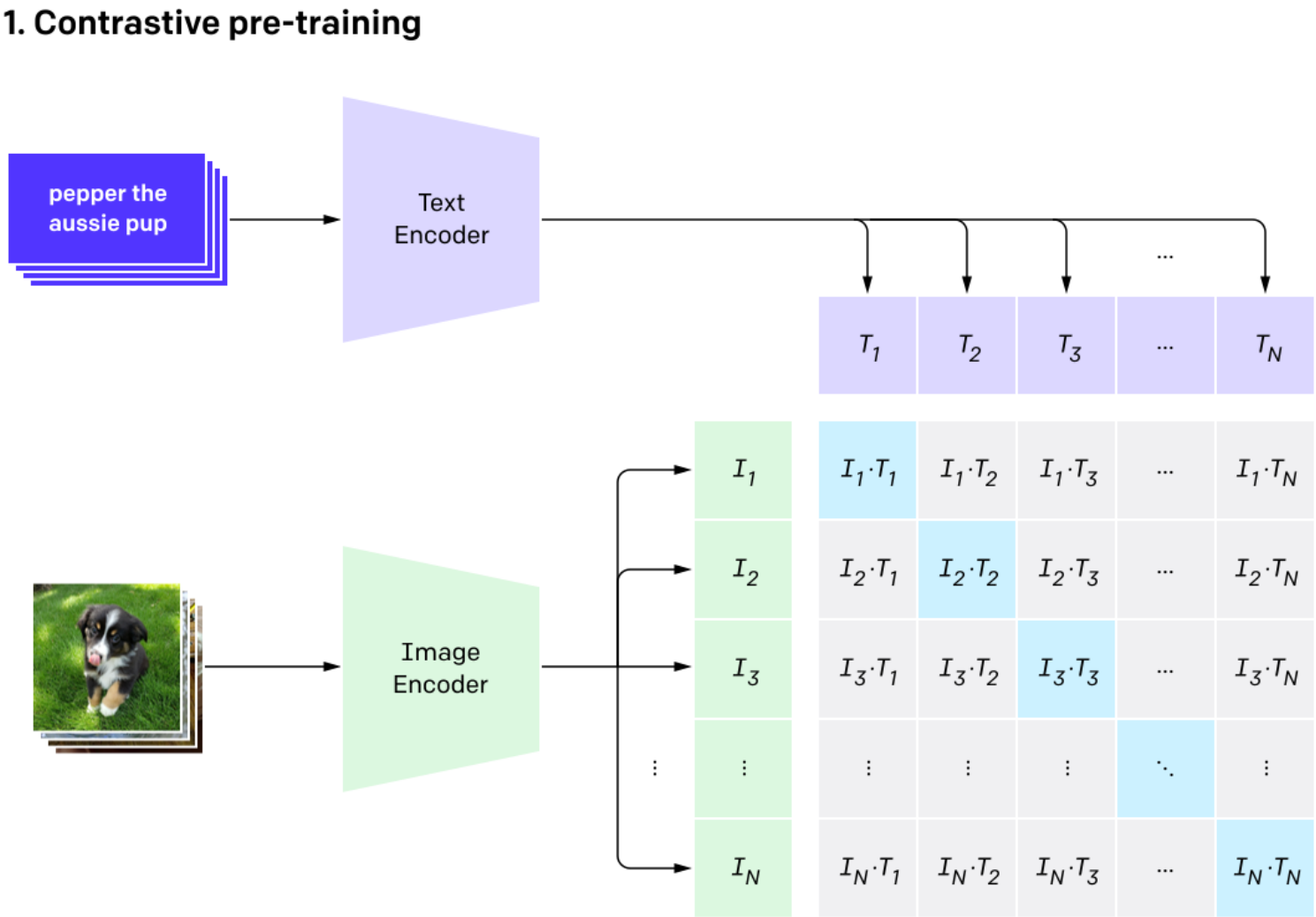

Figure: CLIP training

CLIP - Why use it?

The main idea is that since CLIP is trained to represent images in a way that maximizes the similarity with text embeddings it should be able to internally encode the semantic meaning behind each image. For example the sentences "dogs playing on the beach" and "dogs playing in the park" should have similar text embeddings since they are semantically similar and therefore CLIP should encode the images that "materialize" these sentences with similar vectors, achieving a meaningful semantic encoding.

Autoencoder - A way to encode the content of the image

With CLIP we encode the semantic value of each image, to encode the content of the images I decided to use a common model architecture called **autoencoder**. An autoencoder is a neural architecture that is composed of an **Encoder** and a **Decoder**:

- The encoder gradually elaborates the image through convolutions with learnt 3x3 kernels and, at the end of each step, downsamples the resulting activations with 2x2 2D max pooling kernels. This process continues until the desired dimension is reached. In my implementation the encoder encodes the original 3x64x64 image into a 4x16x16 tensor. The tensor is then flattened to obtain the final 1024-d embeddings.
- The decoder is tasked to perform the opposite operation: starting from the 3x16x16 feature map it must reconstruct the original image by upsampling its input using **transposed convolutions**, that can be seen as a "learnt upsampling" operation.

Autoencoder - Architecture

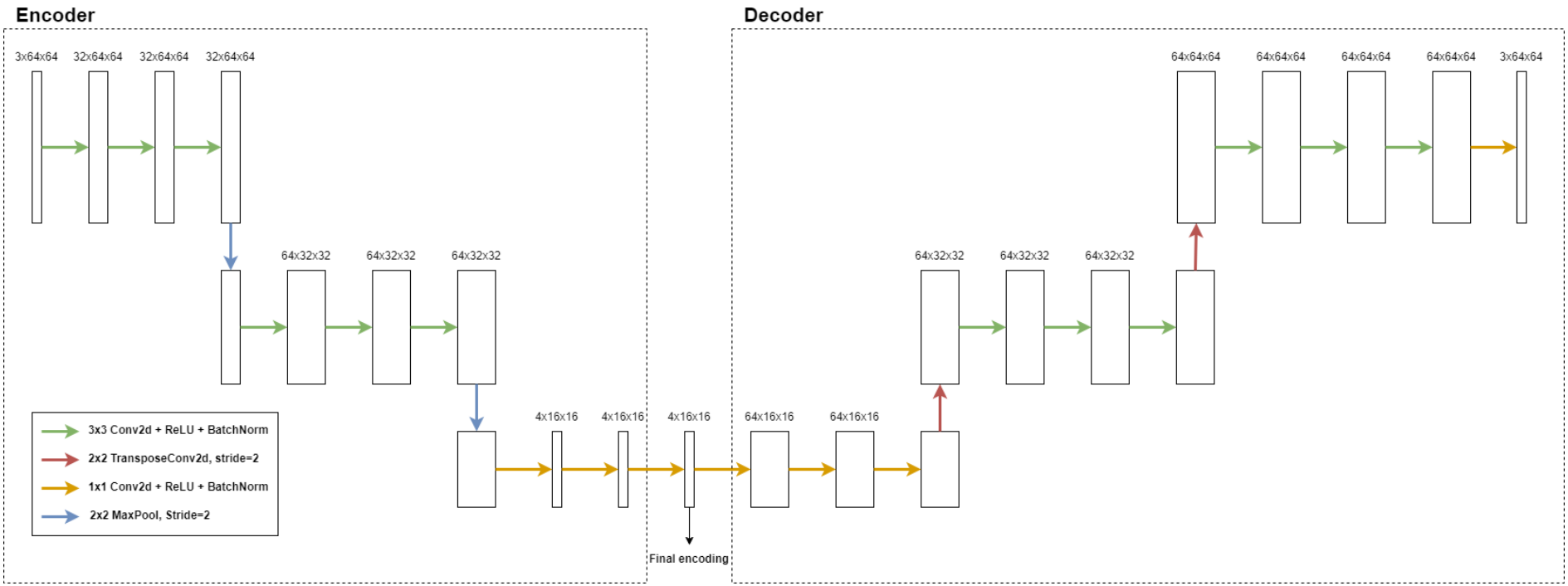

Figure: Autoencoder architecture

Autoencoder - Is the decoder needed?

Notice that this setup is only needed at training time, where we feed the model with pairs $(image, image)$ and we use as the loss function the pixel-wise mean squared error between the original image and the reconstructed one. Once the training ends we know that we obtained an interal representation that can encode the original image well enough to reconstruct it up to a certain degree of error. Once the training ends the decoder is therefore not needed anymore! we just use the encoder to map an input image to its 1024-d representation.

What's FAISS?

FAISS it's a software library developed by Facebook to perform similarity search efficiently.

FAISS - How to use it?

To use FAISS you need to create an index by specifying:

- The length of the vectors that are going to be stored in the index
- The distance used to perform the search

Once the index is initialized the index can be queried with any number of vectors and the library will return a pair of arrays that contains the retrieved indexes and their distances from the query vector respctively.

FAISS - How are the vectors indexed?

- The semantic vectors are normalized and indexed with the **L2 distance**.
- The content vectors are normalized and indexed with **L1 distance**. I decided to use the L1 norm to reduce the effect of the sparsity induced by the dimension of the vectors themselves that resulted always in very small distances if the L2 distance was used.

Database - The necessary files

The database needs 3 files to work properly:

- **The path index:** A dictionary that stores pairs $(id, path)$ were the id is the identifier integer number in the range `[0, stored_images-1]` and the path is the path where the image **copy** for the database is stored.
- **The content\semantic FAISS indexes:** The FAISS indexes, they store a mapping $(id, vector)$ and they can be queried for similarity search. Clearly the *id* used by the FAISS indexes and the one used by the path index is the same.

Database - Adding an image

When the user decides to add an image the database goes through the following steps:

- The content and the semantic representations are added to the FAISS indexes. Notice that by default FAISS assigns them to the id `current_max_index+1`.
- The original files is copied in the `dbs\db_images` folder, this is to avoid the user deleting the original file and database losing track of it.
- the pair `(current_max_id+1, dbs\db_images\new_file_name)` is added to the path index.

Database - One glaring issue

The speed and reliability provided by FAISS indexes comes at a cost: you cannot really remove a vector from the index; you can mask them out, but this operation starts to be efficient only when you mask a lot of vectors. Due to this limitation you can't remove an image from the database once it has been added. One solution could be storing the removed indexes in a different data structure and mask out the indexes removed when they are retrieved from the search, but this would require an "overquerying" to be sure that the end user receives in output the number of images specified in the settings, and this could greatly impact the performances of the database in the long run. In the end I didn't implement a feature to remove images from the database, as this project was more focused feasibility of AI regarding the retrieval of images, rather than the management aspect of it.

Thank you!

And that's it!

Thanks for listening!