3D Augmented Reality
Project B.2
Local feature compression using autoencoders - SfM tests

Donald Shenaj, Daniele Foscarin

January 16, 2021

Contents

1 Introduction

In this project we design a compression strategy for the Surf image descriptors and we integrate it in two python scripts that are able to extract the descriptors, compress them, reconstruct them and generate the matching points together with the files necessary to launch a Structure from Motion reconstruction using the software Colmap. Our implementation is based on OpenCV for the extraction and matching of the descriptors, and PyTorch for the development of an autoencoder that encodes and decodes the descriptors.

The code we have produced is contained in the following files:

- generate_dataset_autoencoder.py is the script that reads the images inside one or more directories and extract the Surf descriptors, using the 64 values version. It generates all the descriptors necessary to create the entire dataset used in the autoencoder development.
- Surf_Autoencoder_3Ddesc.ipynb is the python notebook that has been used for the development, training and testing the autoencoder.
- generate_sfm_data.py is the python script used to extract the Surf descriptors, compute the matches and produces the txt files needed by Colmap. The results obtained with this script will be considered as reference for the comparison after the compression.
- generate_sfm_data_encoder.py is the script that extracts the descriptors from the images and infers them on the encoder model, in order to produce the compressed descriptors. It produces a txt file that will be forwarded to the decoder.
- generate_sfm_data_decoder.py is the script that reads the compressed descriptors and infers them on the decoder model in order to rebuilt the 64 values Surf descriptors; then compute the matches on the descriptors and produce the txt files needed by Colmap.
- sfm_colmap_reconstruction.sh is the shell script that performs the SfM reconstruction by using the Colmap command line interface.

2 Descriptors compression with autoencoder

2.1 Dataset generation

The Surf descriptors dataset used for training the autoencoder is generated using the images of both dataset Portello [1] and Castle-P30 [4], instead the dataset used for testing the performances is generated using the images of the dataset Fountain-P11 [3] and Tiso [2]. The code is implemented in *generate_dataset_autoencoder.py*, by receiving in input the path of the directories containing the images, it produces in output a unique binary file containing the

Surf descriptors of each image contained in the directories. The code is executed two times and produces two datasets (binary files), one for training and the other for testing the autoencoder.

We exploit the OpenCV class cv::xfeatures2d to detect the keypoint in every image and compute the Surf descriptor for every keypoint. We obtain an array that contains in every row the 64 values of a Surf descriptor.
Since the number of keypoints and relative Surf descriptor is very high (723252 for the training set and 1174700 for the test set) we need to store them in a efficient format. The most efficient and simple solution is the creation of a binary file for the arrays containing the descriptor for every keypoint. We produce in this way a binary file for the training set and one for the test set. These files are ready to be read by the notebook containing the Autoencoder development.

2.2 Implementation of the Autoencoder

We decided to use PyTorch for the implementation of an autoencoder, since it provides a fine grained control over the training process of a neural network model [8]. The development and training of the autoencoder has been performed in the Google Colab enviroment.
An autoencoder is a neural network that is composed by an encoder that maps the input tensor into a low dimensional space, and a decoder that takes the low dimensional representation and it generates back a tensor with the same dimensionality of the input. The aim of this network is to have the decoded output as similar as possible to the input of the encoder, even if a lot of information have been lost during the dimensionality reduction process.

Since our descriptors are monodimensional arrays of 64 values, the most simple architecture we can use is an autoencoder where the encoder and the decoder are symmetric and made by linear fully connected layers. It is evident that a fully connected encoder-decoder couple can execute the compression and decompression of any type of data, but we can search for the best architecture if we know how to exploit the structures and regularities of the data that we are working with.

For this purpose we can plot the superposition of many descriptors (Figure 2) and we notice that the amplitude of the values is grouped in chunks of four consecutive values. That is because the Surf descriptor [5] select a square window of (20,20) pixels centered on a keypoint and divide this windows in 16 cells; for every cell it computes the response to the first Haar wavelet in horizontal and vertical direction and in absolute value. So the four values that describe every cell are

$$\sum d_x, \quad \sum |d_x|, \quad \sum d_y, \quad \sum |d_y|. \tag{1}$$

After this, the values in every cell are normalized with a gaussian window centered on the keypoint, so the cells in the edge of the (20,20) windows have smaller

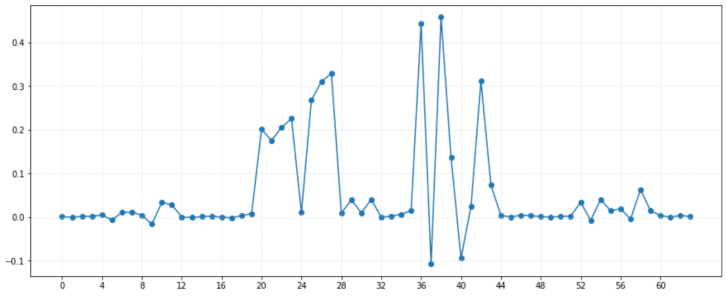

Figure 1: Visualization of a random chosen descriptor array

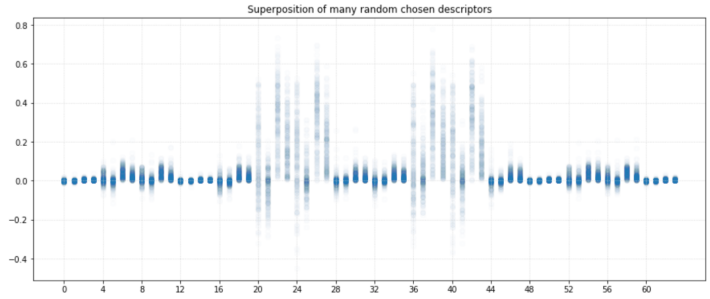

Figure 2: Superposition of 300 random samples fromm the dataset

values then the 4 central cells. That explains why, looking at the descriptor array, we have some group that have a consistent higher average amplitude than others. We can consider now the four metrics as different features of the every cell, so it is intuitive to reshape the 64 values array in a matrix of size (4,4,4), that is like a small image of (4,4) pixel with 4 channels. After implementing a transformation that reshape our data in this way, we can use a 2D convolutional architecture in our autoencoder.

Remembering the structure of a 64 values Surf descriptor (described in section 2.1) we can reshape the linear array in a three dimensional array with sizes (4,4,4). Doing so we are considering the descriptor as a small image patch of (4,4) pixels in 4 channels, and we can now use a convolutional architecture in our autoencoder. Now we can compare the performances between a convolutional autoencoder and a fully connected one.

We used the Optuna framework [6] to perform an optimized grid search of the hyperparameters of the models. The advantages of using optuna instead of implementing a grid search by hand is that we can ask Optuna to suggest the best parameters in a continuous range of values, and it can automatically prune the trainings the are under performing, allowing us to save a lot of time. This was particularly important since the dataset is very large and training an autoencoder is a lengthy process.

In our search for the best hyper parameters we let Optuna find the optimal values about:

- architecture, it's the choice between a linear or a convolutional autoencoder. This choice also determines the shape of the dataset samples that are used.
- batch size, ranging between 100 and 600
- learning rate, in the range 1e-4 and 1e-1
- the size of 4 layers in the chosen model. For the linear model they are the number of neurons in 4 fully connected layers, and for the convolutional model they are the number of filters for 3 convolutional layers and the number of neurons in a fully connected layer. All these sizes are chosen by Optuna in the range between 32 and 128.

2.3 Analysis of the best model

The final Optuna search was performed with 30 trials and 50 epochs for every trial on the values described before. We pruned the trainings that were performing worse than the median of the previous trainings.
The parameters of the best model found by Optuna are the following:

batch size	294
architecture	convolutional
conv1	126
conv2	99
conv3	106
fully connected	59
learning rate	1.0881094987831417e-3

We can also look at the scatter of the validation loss regarding the variation of some parameters during the different trials. In Figure 4 we can notice that the convolutional model performed consistently better than the linear one. We need also to remember that the bad performing pruned training are not represented in the figure.
That allows us to affirm that a convolutional autoencoder based on the feature reshaped as 4 channels images gives better performances than a linear autoencoder based on the linear descriptor.

The model described by those parameters have been retrained on the full dataset. The final loss value in the test set is less than 0.0008. We can do an intuitive check on the performances of the Autoencoder looking at the similarity between an original descriptor and its encoded-decoded copy (Figure 5). We can see that in this random chosen samples from the test set, the reconstruction is very close to the original descriptor.

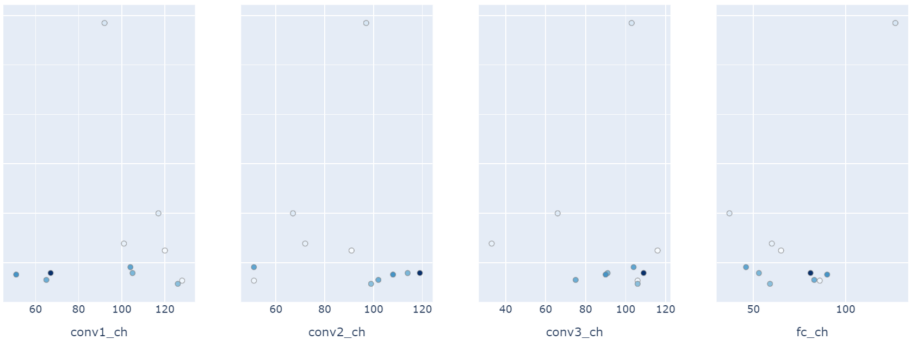

Figure 3: Validation loss with respect to the layers sizes.

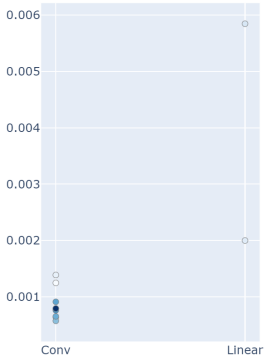

Figure 4: Validation loss respect to the convolutional or linear model, pruned trials are not displayed.

Figure 5: Visualization of five random chosen descriptors with their reconstruction and MSE.

3 Testing on 3D reconstruction using SfM

3.1 Descriptors matching

At this point we have the trained autoencoder and we want to see his performances for the final task which is the SfM sparse reconstruction with the software Colmap. In order to do so we have to establish before a reference result for the compression.
For this task we implemented the python script *generate_sfm_data.py* which, given in input k images of a dataset, it will produce in output a directory with the txt files needed by Colmap in the following tree structure:

```
output_dir
├── desc
│   ├── image_name_0.txt
│   ├── ...
│   └── image_name_k.txt
└── mmm_file.txt
```

More in detail, for each image we compute the Surf descriptors with the OpenCV class cv::xfeatures2d::Surf and we create a txt file with the same name of the input image plus the extension *.txt* and we store them inside the *desc* directory.
The format is the one established by Colmap for the input of Sift descriptors that we will use in a opportunistic way. In fact Colmap internally uses just Sift descriptors and requires them in input, but we will pass some "dummy" Sift descriptors plus our matches done with Surf descriptors and perform the reconstruction without using Sift.
Regarding the input of Sift descriptors, Colmap requires one txt file for each image where in the first row must be specified the number of descriptors followed by the length of each descriptors. Even thought we are using Surf with 64 values, in our txt files inside the *desc* we have to set the descriptors length to 128 since we are importing Sift descriptors with empty values. After the first row the file will be filled by as many rows as the number of descriptors found in the image and each row is composed by the keypoints coordinates (u, v, scale, orientation) followed by 128 zeros that represent the empty Sift descriptors.
Instead for the matching of the Surf descriptors we will use a brute-force approach by exploiting the OpenCV cv::BFMatcher() class. We optimized the matching by setting the L2 norm which is typical for the case of Surf (and Sift) descriptors and we set the crossCheck parameter to be true in order to return only the consistent pairs, that is alternative to the ratio test, used by D. Lowe in [7].
The matching is performed iterating between each combination of couple of images and the results are stored inside *mmm_file.txt* writing for each couple of images their name followed by as many row as matches found between them, and in each row are present the index of the descriptors that are matching.

This solution allows us to have more control on the generation of the descriptors and matching and use Colmap just for the structure from motion recon-

struction. Such technique usually produces best results with minimal number of outliers when there are enough matches.

3.2 Encoding and Decoding

After the training of our autoencoder, we can save the encoder and decoder models and use them as two independent modules. Moreover we decided to divide the problem in a way that simulate a real world application. The simpler example is that in fact it might happen that we want to transmit the Surf descriptors in a distributed application, in order to reduce the bandwidth we can decide to send the compressed values, otherwise there could be the necessity to reduce the size of the descriptor for a big dataset of images.
We also assume that just the server has access to the OpenCV with Surf descriptors since it is a patented algorithm and it is not present in basic OpenCV-Python installation, while the client cannot compute them directly but can decode the received ones, compute the matches and perform the reconstruction using Colmap.

In order to send the compressed Surf descriptors in this scenario we designed a protocol where each packet contains information of the descriptors of one image. In the header we specify the image name followed by the number of descriptors which is the payload length, and in the payload we have the information about keypoints coordinates and compressed Surf descriptors. Recalling the implementation of the autoencoder which requires 3D shaped descriptors, in *generate_sfm_data_encoder.py* we extract the Surf descriptors as was done in *generate_sfm_data*, perform the reshaping to the 3D structure and finally forward pass to the encoder, producing the encoded Surf descriptors that will be sent to the decoder using our protocol.
To simplify the implementation the encoded descriptors data will be written as txt file by the *generate_sfm_data_encoder.py* and will be read by *generate_sfm_data_decoder.py*. In the first row of this intermediate txt file there will be the image name followed by the number of descriptors followed by as many rows as the descriptors, in this way we know how many rows to read after the header. Similarly to the Colmap format previously mentioned, each row is composed by the four keypoints coordinates followed now by the compressed Surf descriptors, instead of the 128 empty Sift values. Instead *generate_sfm_data_decoder.py* receives in input the intermediate txt file, decodes the descriptors, reshape to linear structure and produces the same output directory for the Colmap reconstruction seen before by *generate_sfm_data.py*, with the difference that now the matches are performed over the decoded descriptors.

3.3 Reconstruction with Colmap Command-line Interface

In order to fully automatize the workflow we decided to implement each Colmap operation that are imports of descriptors and matches, matching operation and output of the SfM sparse reconstruction with the command line interface. This happens to be very useful in case we want to execute some per-

formances evaluation because it allows us to perform programmatically the SfM reconstruction for the given images. The shell script is implemented in *sfm_colmap_reconstruction.sh*, it requires in input the image dataset, the directory with empty Sift descriptors and matches required by Colmap and the ouput directory path where the reconstruction will be stored as binary and PLY file. The first is required in case we want to see the reconstruction by using the Colmap GUI that allows as to see better the camera positions reconstructed, but we provided also the PLY format which is more portable and tipically more convenient if we want just the visualization, losing the camera positioning. Additionally the script performs also some statistical analysis useful for the comparison between different reconstructions containing information about: cameras, images, points, observations and some errors measures. The code will be run four times, namely for the matches of the compressed and not compressed Surf descriptors of the dataset Tiso and Fountain previously mentioned.

3.4 Performances analysis

We can look at the reconstruction performed by Colmap for the original set of uncompressed descriptors and for the decoded descriptors (Figure 6, 7, 8, 9). The two point cloud models looks very similar, with the uncompressed reconstruction being a little more dense. This result is expected since we reduced the size of the compressed descriptor from 64 values to 16 values, but the final result is satisfying. By looking at the reconstruction statistics we can also make some considerations about the number of points that are present in the different 3D models. It is also interesting to confront them with the number of matches that have been computed by cv::BFMatcher in our scripts.

data	Colmap 3D points	BFMatcher matches
fountain	27343	466697
fountain compressed	19062	572073
tiso	73355	3104924
tiso compressed	49321	4271222

We can see in the table that the number of 3D points have reduced by circa 30% between the uncompressed and compressed version of the fountain dataset, and reduced by 33% for the tiso dataset. This while the number of matches found by cv::BFMatcher has greatly increased.
This means that the information lost during the compression have significantly reduced the quality of the matches found, leading to the creation of many "false positive" during the matching process. This result is also expected, because if we remember our knowledge about the autoencoders, the neural network we have trained is capable of the generalization of the archetypal shape of our Surf descriptors, and without a strong clustering or classification (that is completely absent in our data distribution) the autoencoder struggles in the reconstruction of particular descriptors. In other words the set of reconstructed descriptors contains samples that are more correlated to each other, and this phenomenon

leads to the insurgence of many “false matches”.
On the other hand, even if the 3D model reconstructed by Colmap is sparser when using the compressed dataset, it stills look very good.
A fine tuning of the matching process performed by cv::BFMatcher will surely improve the quality of the reconstruction by preventing the creation of too many false matches.

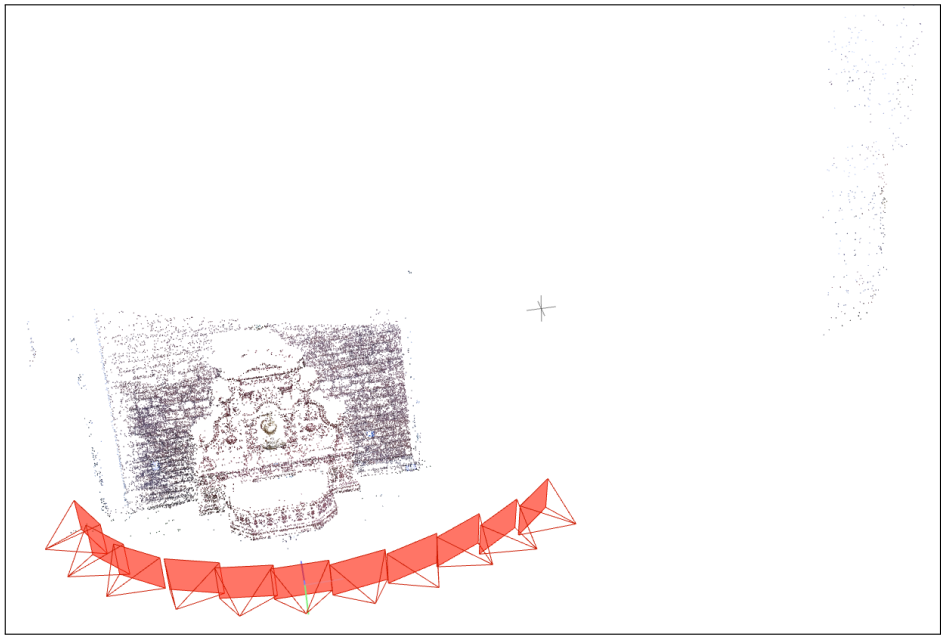

Figure 6: Fountain model, original descriptors

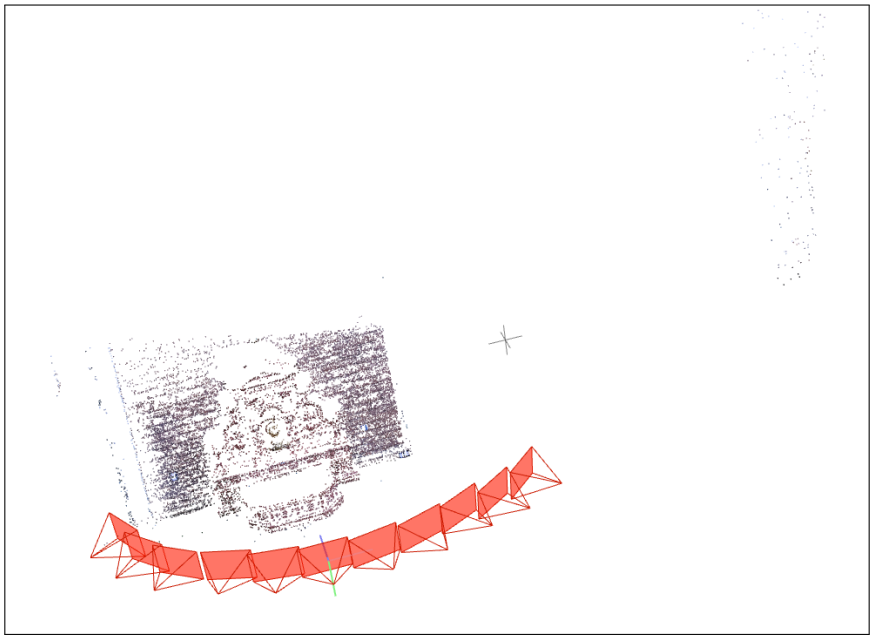

Figure 7: Fountain model, decoded descriptors

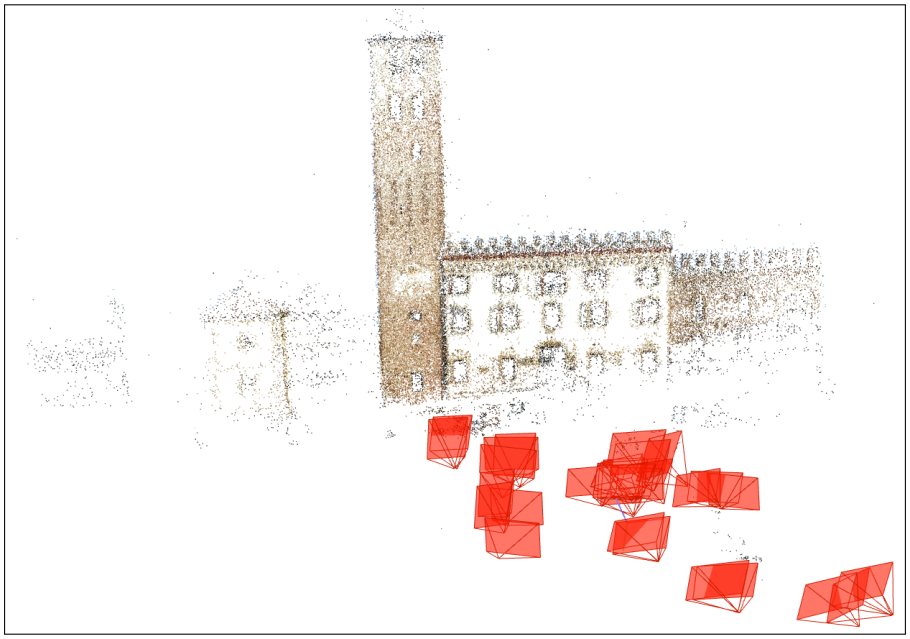

Figure 8: Tiso model, original descriptors

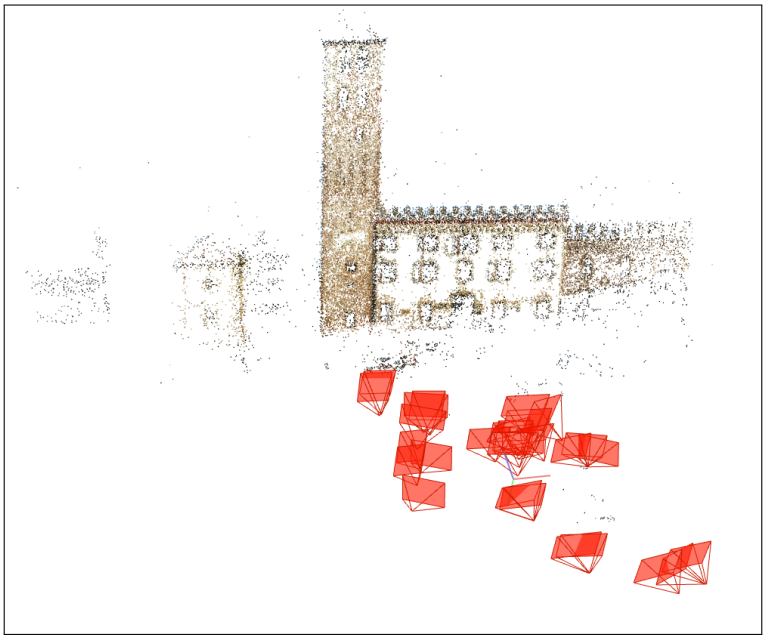

Figure 9: Tiso model, decoded descriptors

4 Conclusions and possible improvements

We have built a series of script that simulates the workflow of a distributed application. The server side, with access to the licensed modules of OpenCV like the Surf descriptor, extracts the descriptors and encode them reducing their size by 75%, while the client size decode them. execute the matching and launch the Colmap reconstruction in an automated way using its command line interface. The reconstruction obtained looks very similar to the reconstruction obtained with the uncompressed descriptors.

The development of the encoder and decoder model was carried out using a smart optimization process for the hyperparameters, and comparing the performances of a fully connected model and a novel convolutional model based on the interpretation of the descriptors array as multichannel image-shaped arrays, that showed to produce consistently better results.

This result could be still refined using a longer search and training in the autoencoder development phase, since the dataset was heavy and it lead to a very slow training process, and a fine tuning of the cv::BFMatcher object in order to reduce the false matches in the client side.

5 Appendix

Parameter	Original	Compressed
Cameras	11	11
Images	11	11
Registered images	11	11
Points	27343	19061
Observations	131345	77515
Mean track length	4.803606	4.066681
Mean observations per image	11940.454545	7046.818182
Mean reprojection error	0.563558px	0.450488px
Elapsed time	0.844[min]	0.583[min]

Table 1: SfM statistics: Fountain reconstruction from original and compressed Surf descriptors.

Parameter	Original	Compressed
Cameras	30	30
Images	30	30
Registered images	30	30
Points	73376	49236
Observations	418827	232675
Mean track length	5.707956	4.725709
Mean observations per image	13960.900000	7755.833333
Mean reprojection error	1.146193px	0.978028px
Elapsed time	9.791[min]	6.377[min]

Table 2: SfM statistics: Tiso reconstruction from original and compressed Surf descriptors.

References

[1] M. Simone: *Portello Dataset*, `www.dei.unipd.it/~sim1mil/materiale/3Drecon/portelloData.zip`, 2017

[2] M. Simone: *Tiso Dataset*, `http://www.dei.unipd.it/~sim1mil/materiale/3Drecon/tisoData.zip`, 2017

[3] P. Moulon, R. Janvier: *Fountain Dataset*, `https://github.com/openMVG/SfM_quality_evaluation/tree/master/Benchmarking_Camera_Calibration_2008/fountain-P11`, 2008.

[4] P. Moulon, R. Janvier: *Castle-P30 Dataset*, `https://github.com/openMVG/SfM_quality_evaluation/tree/master/Benchmarking_Camera_Calibration_2008/castle-P30`, 2008.

[5] Bay, Herbert and Tuytelaars, Tinne and Van Gool, Luc: *Surf: Speeded up robust features*, 2006, Computer Vision-ECCV 2006.

[6] `https://optuna.org/`.

[7] D. G. Lowe: *Distinctive image features from scale-invariant keypoints*, International Journal of Computer Vision, 60, 2, 2004.

[8] Autoencoder implementation based on the code provided by Dr. Alberto Testolin, Dr. Matteo Gadaleta, *Neural Networks and Deep Learning* course, DEI Unipd.