Use of the code for heterogeneous reconstruction

The purpose of this document is to present the use of code to perform heterogeneous reconstruction (Chapter 4 of my thesis). The main folder is *reconstruction_with_dl*. The code in this folder depends on a number of files in the following folders: *data_generation*, in particular the file *generate_data.py* (for generating synthetic data), *manage_files*, in particular the file *read_save_files.py* (for reading and saving files) and *common_image_processing_methods*, which contains functions commonly used in image processing (registration, rotation, resizing, etc.).

1) Brief overview of the code

- The main file is *end_to_end_architecture_volume.py*. It is used to 1) generate the FluoFire encoder-decoder architecture using the class named *End_to_end_architecture_volume* 2) train this architecture on a user-specified dataset. The training of the architecture is launch in the __main__ part of this file.

- The file *reconstruct_on_real_data* allows to launch the training of the architecture on a real data set

- The file *default_params.py* in folder *test_params* contains two dictionnaries that regroup all hyperparameter of the code. The dictionnary *params_learn_setup* regroups parameter used in the learning algorithm and the dictionnary *params_data_gen* regroups parameters used to generate simulated data. Details of the parameters are specified in the file.

- The file *SIREN.py* contains 1) the class that represents the fully connected architecture with sinusoidal activation functions (*Siren*) 2) A function *fit_SIREN_with_image* that takes an image as input and fit a sinusoïdal representation network on it. In particular, this function can be used if we want to start the learning of FluoFire with a non-random initialization

- The file *pose_net.py* contains a class *PosesPredictor* that generates the encoder architecture, a network that takes as input a batch of views and returns the associated poses and confomrational parameters. Different kind of encoder can be used (Vgg16 or hollyNd) the details of these architectures is coded respectively in the files *VGG16.py* and *hollyNd.py* , present in the subfolder *encoder.*

- The file *data_set_views.py* that contains mainly two classes . The class *ViewsRandomlyOriented* represents a data set of real data views. To initialize this class, you need to specify the input views as a 4d array (nb_views, size, size, size) and the list of corresponding file_names. On simulated data, you can not directly use this class, you need to use the class *ViewsRandomlyOrientedSimData* that inherits from the previous one. It takes as input the views but also the true rotation matrices, translation vectors and heterogeneity parameters. This file contains also an important function, *get_mgrid* that generates a flatten grid with specify side lenght and dimension.

- The file *donut_cylinder.py* implements the form of tore that can be used instead of using the Siren representation. It is used to impose a strong prior on the reconstructe object when we do not have enough data.

- The file *losses.py* implements the symetric loss, which is called *L2LossPreFlip* . This loss is not much used since the pose to pose loss allows to correct the problem of convergence to local minima without raising significantly the memory usage as the symmetry loss do

- The file *generate_data.py* in data_generation folders contains the function *heterogene_views_from _centriole* used to generate heterogene data with one degree of freedom from a model of centriole. The function *heterogene_views_from_centriole_2_degrees_of_freedom* generates instead two degrees of freedom in the data.

2) Run the code on synthetic data

2.1. Generate the data

The synthetic data is generated from a model of centriole that can be created with the function *simu_cylinder_rev* in file *create_centriole.py.* The centrioel model is a point cloud model that is transformed to an image by evaluated a gaussian on grid centeres on the points of the point cloud.

Open the file *generate_data.py* in the folder *data_generation.py.* Then, launch the function *heterogene_views_from _centriole.* You need to specify
- *save_fold* → path where views are saved
- *params_data_gen* → the dictionnary imported from *default_params.py* file . Eventually modify some values of this dictionnary
- nb_channels → nombre de canaux
- *het_vals_all_channel* → two dimensional array (or list of list) of shape (nb_channels, nb_views) het_avl_all_channels[c, l] corresponds to the heterogeneity values (i.e. the lenght of the generated centriole in the case of this function) of the l ^ th view of channel c
- *cs* → list of lenght *nb_channels* : cs[c] is the radius of the centriole of channel c (identical for all the views in the case of this function)
- *zero_rotation* : if True, all views are generating without rotating
- *nb_missing triplets* : number of triplets of microtubule romoved to the model
- *sig_gauss* : standard deviation of gaussian used to evaluate the point cloud on a grid
- *homogene* : if true generate homogene data and do not take into account the parameter *het_vals_all_channels*

This call will generate a dataset with one degree of freedom : the lenght of the centriole. You can also generate a data set with two degrees of freedom. In this case, you can launch the function *heterogene_views_from_centriole_2_degrees_of_freedom* . The first degree of freedom is the lenght and the second the radius.
- het_val_lenght → two dimensional array of shape (nb_channels, nb_views), het_val_lenghts[c, l] corresponds to the lenght of the centrioke view l of channel c
- het_val_radius → two dimensional array of shape (nb_channels, nb_views), het_val_radius[c, l] corresponds to the raidus of the centrioke view l of channel c.

2.2. Launch the reconstruction

Go to the file *end_to_end_architecture_volume.py* . The call of the training procedure is already written in the __main__ part but you need to be aware of some points.
T
he parameters dictionnary *params_data_gen* and *params_learn_setup* are imported from the file *default_parms.py* . You need to set the values of the dictionnary *params_data_gen* to the values that were used to generat the data. In particular you need to be sure that the standard deviations of the psf along z axis and in xy plane are the same that the one you used when generated the data. An important parameter to set is also the path where you saved the views
(params_data_gen[pth_views])

For the dictionnary *parms_learn_setup*, you can modify the parameters as you want. They are all described in default_params.py file. You can keep the default parameter for every parameter except « save_fold » that is mandatory to fill. It is the path where the results will be saved.
If you want to use the pose to pose training, you need to modify the parameter *"nb_epochs_each_phases_ACE_Het"* from None value to a value in the form [n, m].
Make sur that you set « nb_channels » to the right value

Another important parameter to set is the number of dimension in the latent space « nb_dim_het ».

2.3. Analyze the results

I recommand to launch the algorithm for a large number of epochs (at least 3000 when you use 250 views, and even more if you use less views) because it can take a long time to converge. For most of the runs, the reconstrfuction will be quite bad for a long time and it will start to ressamble the original object only after several thousands epochs.