Manual to ovseg, a fexible training framework for 3D medical segmentation using deep learning

Thomas Buddenkotte

September 14, 2022

Abstract

This document explains the usage of the ovseg library in detail. The library was designed to simplify training and validation of 3D medical segmentation problems. This document discusses advantages of the library, introduces basic concepts and gives some suggestions for customization. The code is available under `https://github.com/ThomasBudd/ovseg`.

1 Why ovseg?

The library was implemented during my PhD on ovarian cancer segmentation (thus the name ovseg). The main purpose was to have a tool that allows me simple manipulation of model hyper-parameters for tuning and of the **model structure** to play with different approaches (model cascades, deep supervision, learned pre-processing etc.). The key feature of the code is the plug-and-play concept, that will be explain in the next section. Additionally, general reading functions for new dataset allow quick adaptation to new datasets outside of ovarian cancer segmentation.

The library follows the nnU-Net library in many ways and concepts, but differs in some aspects. Generally the nnU-Net library was a great template to use and learn from during the development of my code. I highly acknowledge the authors of nnU-net for making the code publicly available and usable, I couldn't have done my implementation without theirs as a role model, so thanks a lot! The implementation was done using PyTorch 1.9.

2 Model definition and objects

The central part of the implementation is the SegmentationModel(V2). Often people consider only the architecture when discussing the application of a deep neural network. However, in my experience (similar to the main hypothesis of nnU-net [IJK+21]) carefully tuned or chosen hyper-parameters are more important than architectural modifications. This is why in this implementation we grasp a neural network model as the **collection of all functionalities that are needed for network training and inference.** A model holds everything that uniquely describes the segmentation pipeline. For this the model object initialises and holds the **model objects** that represent

1. data pre-processing
2. data augmentation
3. the network
4. network evaluation
5. post-processing
6. data sampling
7. training.

The collection of all hyper-parameters used in each of the objects are called model_parameters in this implementation. The model_parameters are represented as a nested dictionary that needs to be passed to the model on first creation. Models in ovseg are identified via a unique combination of the name of the training data (data_name, e.g. 'kits21'), a name for the preprocessed data (preprocessed_name, e.g. 'kidneys_lowes') and a name for the model (model_name, e.g. 'learning_rate_0.04'). These must be given as well when first creating the model. The names are also needed when searching for a specific model, or model predictions in the data base (see next section).

Next, we give a short overview on what the function of the different model objects are.

The data pre-processing object is prepares raw scans for their introduction to the network. As suggested in nnU-Net [IJK+21], this mainly entails windowing (in case of CT scans), normalization of gray values, interpolation to a given voxel spacing and pooling. To reduce computational burden during training, the pre-processing for the training set is applied before the training is started and saved in a designated folder (see next section). This makes the pre-processing object different from the other objects. This object has to be created once before creating the model to perform initial pre-processing of the training data. The function plan_preprocessing_raw_data can infere median voxel spacing and adequate windowing and scaling coefficients from one or more raw datasets. The function preprocess_raw_data will convert one or more raw datasets into preprocessed data and store them ready for training. Besides this, the object's __call__ function will convert a raw 3D scan into a pre-processed 3D scan. Please see the next section for how data is represented in the library.

The augmentation object executes data augmentation for batches of data. In contrast to nnU-net [IJK+21], which performs all augmentation on the CPU, we typically execute the augmentation operations on the GPU. This is mainly because the pytorch implementation of the interpolation is very fast on a GPU compared to interpolation methods on the CPU. In practise we found the execution time of the augmentation to be negligible compared to forwards and backwards pass through a standard 3D U-Net. This lessens the strain on the CPU, which we found to be the bottleneck of nnU-net training for some server configurations. Most of the augmentations are available for both CPU and GPU, except for the mask augmentations used for the neural network predictions of a previous stage in a cascaded training, which are only available for the CPU.

The network object is simply the pytorch module of the deep neural network.

The network evaluation object is responsible for evaluating a full 3D scan. Consider that networks are typically trained using only crops of full 3D scans. At the moment, the evaluation object performs the SlidingWindow algorithm as suggested by nnU-net [IJK+21], e.g. using the same patch size during training and inference, using half a patch size overlap in each direction and applying Gaussian weighting to prevent stripe artefacts in the prediction. The output of the evaluation object will be tensor containing the softmax (or sigmoid) outputs.

The post-processing object is mainly responsible for computing integer-valued segmentation masks from soft (softmax or sigmoid) segmentations and performs non-differentiable calculations. In our current implementation, the object interpolated the 3D volume of soft segmentations back to the original pixel spacing (before pre-processing) and performs the argmax operation to obtain hard labels. Additionally some cleaning operations like removing small connected-components or morphological operations can be applied.

The data sampling object is responsible for loading full 3D scans for inference and batched data of image crops used for training. The datasets hold by the data object are simply python iteretable objects that return the full 3D scans from raw or pre-processed data including the image (and manual segmentation if available) arrays along with information on the scan such as the (original) voxel spacing or patient names, scanning timepoints if available. The dataloaders are implemented as standard pytorch dataloaders and load batches of crops used for training. The data object also performs splitting the data into folds for cross-validation training.

The training object can be used to perform the network training. The object automatically performs training, loading previous training stages (in case the training was stopped) and preparing files for progressive learning.

As data loading takes time there is also an option to load models only in inference_mode, in which case the data sampling and training object are not created.

The model object has additionally functions to evaluate raw scans (run pre-processing, evaluation and post-processing), and evaluate full datasets (eval_raw_dataset, eval_validation_set, eval_training_set). When evaluating a full dataset the predictions are computed and stored, if manual labels were available

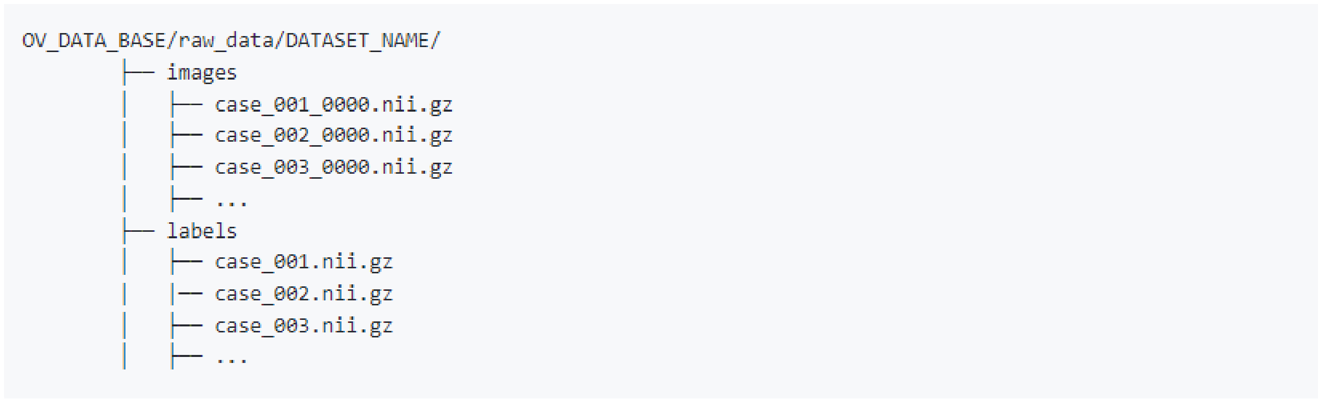

```
OV_DATA_BASE/raw_data/DATASET_NAME/
        ├── images
        │   ├── case_001_0000.nii.gz
        │   ├── case_002_0000.nii.gz
        │   ├── case_003_0000.nii.gz
        │   ├── ...
        ├── labels
        │   ├── case_001.nii.gz
        │   |── case_002.nii.gz
        │   ├── case_003.nii.gz
        │   ├── ...
```

Figure 1: Example for nfiti files encoding a raw dataset.

loss metrics are computed.

3 Database structure

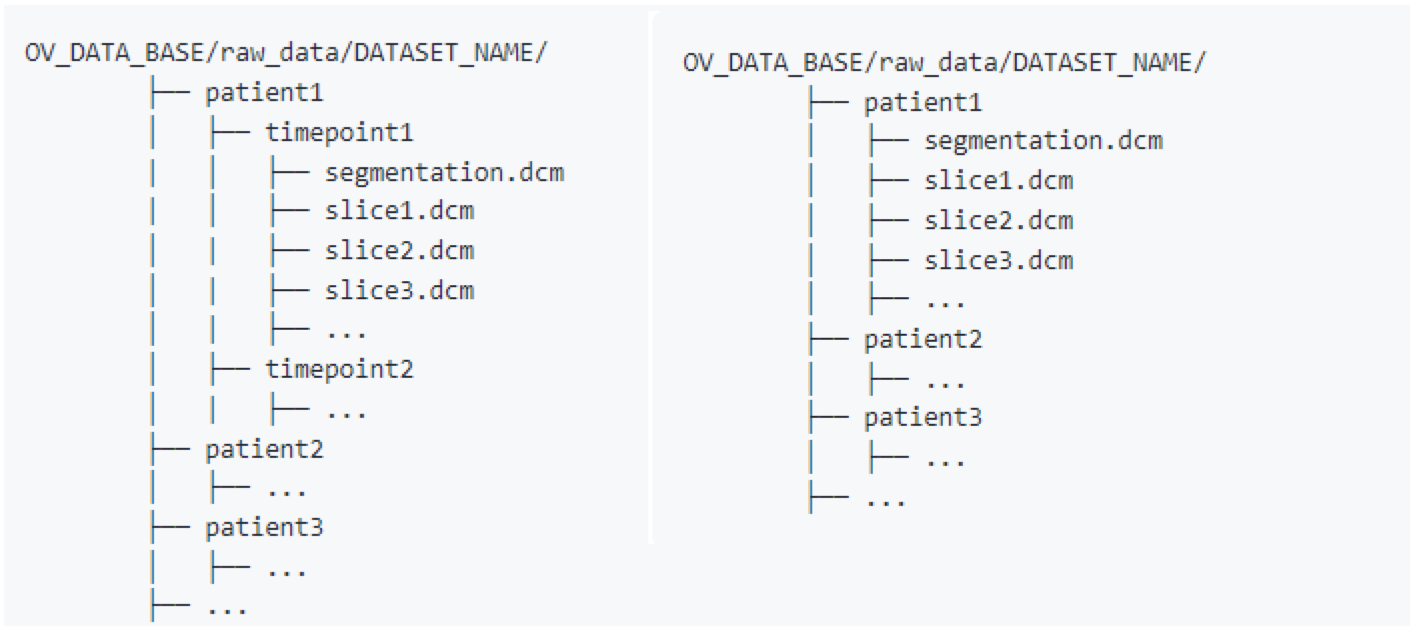
This section describes how data, models, predictions etc. are organized in a data base.

As stated in the github repository, It is necessary to set the environment variable OV_DATA_BASE in order to use the library. This is the central data base in which trained models, raw data, predictions and plots are stored. If you use a system of servers, it is recommended to let the OV_DATA_BASE point to a central directory and additionally create the variable OV_PREPROCESSED to point to a local fast disk. In this case the pre-processed data, which is used during training and thus should be loaded fast, is stored at this position, otherwise it is stored at OV_DATA_BASE/preprocessed.

The path OV_DATA_BASE/raw_data stores the datasets before pre-processing. Each dataset should be given a seperate folder, this name is then used as a data_name when running pre-processing and to identify models trained with this dataset as training data. For example, if you have access to training, validation and test data for your task, you can name your folders 'TASK_trn', 'TASK_val' and 'TASK_tst'. The data_name would be 'TASK_trn' and the validation data can be evaluated via model.eval_raw_dataset(TASK_val). The raw data can be currently encoded in two datatypes: nifti and dicom files. The usage of nifti files is similar to nnU-Net or the Medical Decathlon. The dataset folder is supposed to contain the two folders 'images' and 'labels' that contain the corresponding nifti files. An example is demonstrated in Figure 1. For dicom images any type of folder structure is allowed. Make sure that only axial reconstructions are contained in your dataset, the code won't remove other types of reconstructions such as topograms or sagital slices by itself. The code also assumes that all dicoms found in one folder belong to the same reconstruction, make sure that each reconstruction is contained in a seperate folder. If you're performing training, include the segmentations as dicomrt files. Each folder with reconstruction dicoms should have exactly one additional dicomrt file with the corresponding segmentation. Missing segmentations are interpreted as empty segmentations masks (only backgorund). An example is given in Figure 2.

After pre-processing the raw dataset 'data_name' with to a pre-processed dataset called 'preprocessed_name', the designated folder containing that data can be found at
OV_DATA_BASE/preprocessed/data_name/preprocessed_name
(or OV_PREPROCESSED/data_name/preprocessed_name if the corresponding variable was initialized). This means that the same 'preprocessed_name' can be used for two different datasets.

For each model created with the identifiers 'data_name', 'preprocessed_name' and 'model_name', predictions, plots and the model folders can be found at
OV_DATA_BASE/{predictions, plots, trained_models}/data_name/preprocessed_name/model_name. If the raw data was given in dicom format, the predictions will be stored in nifti and dicom format. If the raw data was given in nfiti format, the predictions will only be stored in nifti format. The trained_models folder will contain a stored version of the model_parameters (as .pkl and .txt file), files that store evaluation metrics of dataset evaluations (e.g. cross-validation and test results) and have a seperate folder for each fold that was trained. These folders contain mostly model weights and training



```
OV_DATA_BASE/raw_data/DATASET_NAME/
        ├── patient1
        |   ├── timepoint1
        |   |   ├── segmentation.dcm
        |   |   ├── slice1.dcm
        |   |   ├── slice2.dcm
        |   |   ├── slice3.dcm
        |   |   ├── ...
        |   ├── timepoint2
        |   |   ├── ...
        ├── patient2
        |   ├── ...
        ├── patient3
        |   ├── ...
        ├── ...
```

```
OV_DATA_BASE/raw_data/DATASET_NAME/
        ├── patient1
        |   ├── segmentation.dcm
        |   ├── slice1.dcm
        |   ├── slice2.dcm
        |   ├── slice3.dcm
        |   ├── ...
        ├── patient2
        |   ├── ...
        ├── patient3
        |   ├── ...
        ├── ...
```

Figure 2: Example for a raw dataset of dicom files.

checkpoints.

A full example of a database is visualized in Figure 3

An example of model_parameters is given in Figure 4.

4 BYO...

One of the biggest strength is flexibility and modality. The easiest way to Bring Your Own (BYO) aspects to the segmentation model is to alter the model_parameters. It is recommended to first train a dummy model and then check out the model_parameters.txt file in the corresponding trained_models folder. A selection of important parameters is explained in the next section.

The next easiest thing it to BYO network architecture or loss function. To BYO network you simply have to create a .py file in which the pytorch module is defined, an import it in ovseg.network.__init__.py e.g. via

```
from ovseg.networks.YOURFILE import YOURNETWORK
```

Next you have to tell the model to use this new module through the model_parameters via

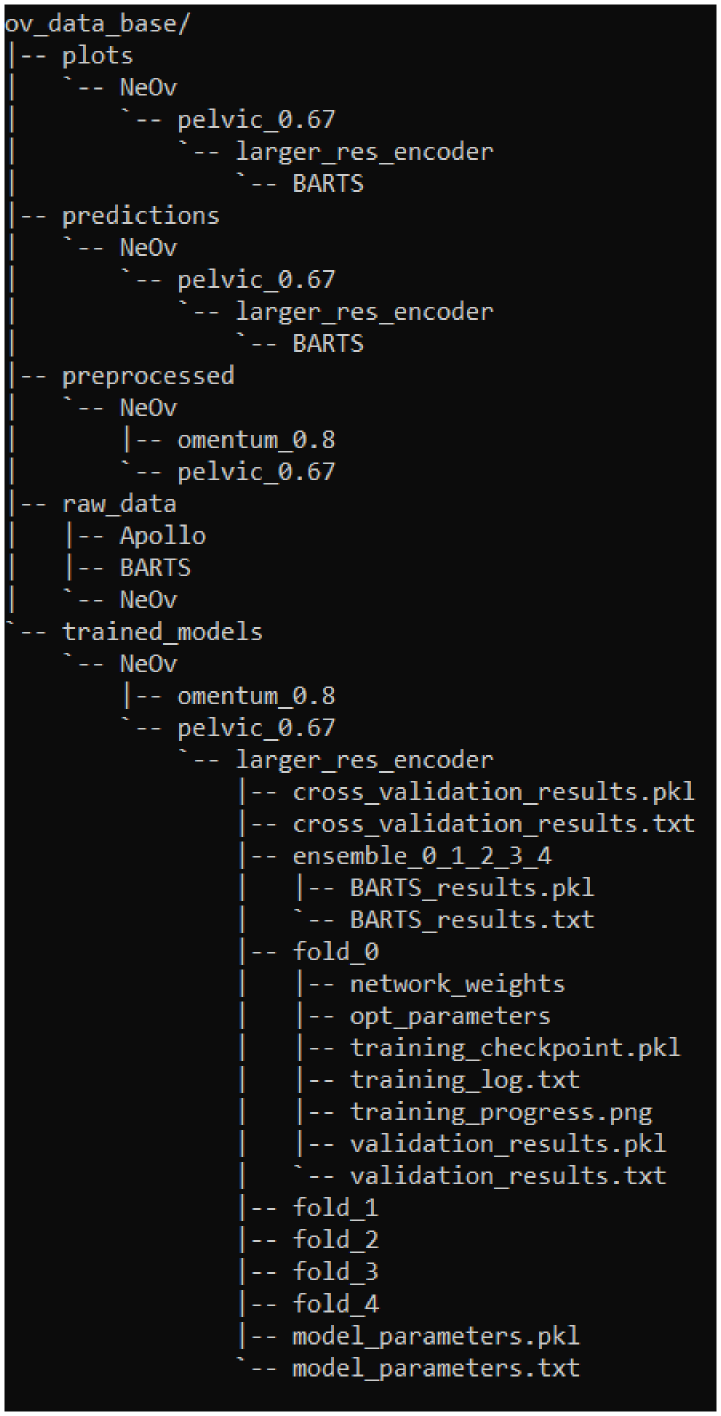
```
model_parameters['architecture'] = 'YOURNETWORK'
```

Changing the loss function or adding a new component to it works similarly. First, include or import the implementation of your new loss function in ovseg.training.loss_functions.py. To use a new combination of loss functions set the parameters for example via

```
model_parameters['training']['loss_params']['loss_names']
    = ['dice_loss_sigm_weighted', 'cross_entropy_exp_weight']
model_parameters['training']['loss_params']['loss_kwargs']
    = [{'w_list': [-1.5, -0.5]}, {'w_list': [-1.5, -0.5]}]
```

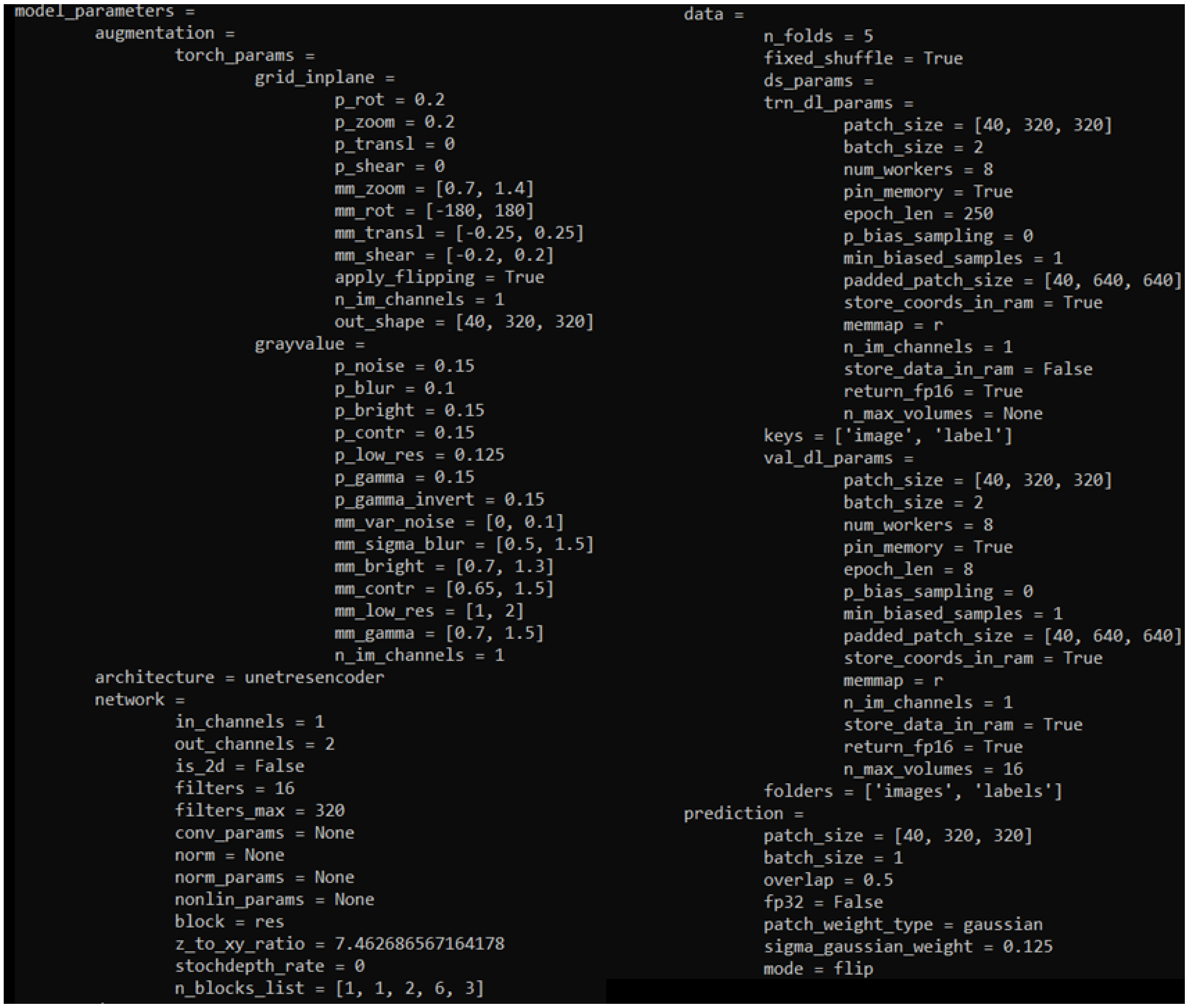
loss_names is the list of all names of the pytorch modules representing each loss function, the loss_kwargs are used while initialization of the losses. Additionally the weights in the linear combination of the losses can be set, by default it is [1,...,1]. Just like nnU-Net, ovseg introduced the loss function to multiple resolutions of the U-Net and uses then a weighted average. This is happening in loss_functions_combined.py.

To exchange model object, create a new model class and overload the 'initialise_XYZ' function (see ModelBase.py). This is more experimental, be careful!



```
ov_data_base/
|-- plots
|   `-- NeOv
|       `-- pelvic_0.67
|           `-- larger_res_encoder
|               `-- BARTS
|-- predictions
|   `-- NeOv
|       `-- pelvic_0.67
|           `-- larger_res_encoder
|               `-- BARTS
|-- preprocessed
|   `-- NeOv
|       |-- omentum_0.8
|       `-- pelvic_0.67
|-- raw_data
|   |-- Apollo
|   |-- BARTS
|   `-- NeOv
`-- trained_models
    `-- NeOv
        |-- omentum_0.8
        `-- pelvic_0.67
            `-- larger_res_encoder
                |-- cross_validation_results.pkl
                |-- cross_validation_results.txt
                |-- ensemble_0_1_2_3_4
                |   |-- BARTS_results.pkl
                |   `-- BARTS_results.txt
                |-- fold_0
                |   |-- network_weights
                |   |-- opt_parameters
                |   |-- training_checkpoint.pkl
                |   |-- training_log.txt
                |   |-- training_progress.png
                |   |-- validation_results.pkl
                |   `-- validation_results.txt
                |-- fold_1
                |-- fold_2
                |-- fold_3
                |-- fold_4
                |-- model_parameters.pkl
                `-- model_parameters.txt
```

Figure 3: Example of the database structure



```
model_parameters =
        augmentation =
                torch_params =
                        grid_inplane =
                                p_rot = 0.2
                                p_zoom = 0.2
                                p_transl = 0
                                p_shear = 0
                                mm_zoom = [0.7, 1.4]
                                mm_rot = [-180, 180]
                                mm_transl = [-0.25, 0.25]
                                mm_shear = [-0.2, 0.2]
                                apply_flipping = True
                                n_im_channels = 1
                                out_shape = [40, 320, 320]
                        grayvalue =
                                p_noise = 0.15
                                p_blur = 0.1
                                p_bright = 0.15
                                p_contr = 0.15
                                p_low_res = 0.125
                                p_gamma = 0.15
                                p_gamma_invert = 0.15
                                mm_var_noise = [0, 0.1]
                                mm_sigma_blur = [0.5, 1.5]
                                mm_bright = [0.7, 1.3]
                                mm_contr = [0.65, 1.5]
                                mm_low_res = [1, 2]
                                mm_gamma = [0.7, 1.5]
                                n_im_channels = 1
        architecture = unetresencoder
        network =
                in_channels = 1
                out_channels = 2
                is_2d = False
                filters = 16
                filters_max = 320
                conv_params = None
                norm = None
                norm_params = None
                nonlin_params = None
                block = res
                z_to_xy_ratio = 7.462686567164178
                stochdepth_rate = 0
                n_blocks_list = [1, 1, 2, 6, 3]
```

```
data =
        n_folds = 5
        fixed_shuffle = True
        ds_params =
        trn_dl_params =
                patch_size = [40, 320, 320]
                batch_size = 2
                num_workers = 8
                pin_memory = True
                epoch_len = 250
                p_bias_sampling = 0
                min_biased_samples = 1
                padded_patch_size = [40, 640, 640]
                store_coords_in_ram = True
                memmap = r
                n_im_channels = 1
                store_data_in_ram = False
                return_fp16 = True
                n_max_volumes = None
        keys = ['image', 'label']
        val_dl_params =
                patch_size = [40, 320, 320]
                batch_size = 2
                num_workers = 8
                pin_memory = True
                epoch_len = 8
                p_bias_sampling = 0
                min_biased_samples = 1
                padded_patch_size = [40, 640, 640]
                store_coords_in_ram = True
                memmap = r
                n_im_channels = 1
                store_data_in_ram = True
                return_fp16 = True
                n_max_volumes = 16
        folders = ['images', 'labels']
prediction =
        patch_size = [40, 320, 320]
        batch_size = 1
        overlap = 0.5
        fp32 = False
        patch_weight_type = gaussian
        sigma_gaussian_weight = 0.125
        mode = flip
```

Figure 4: Example of model parameters. Variables starting with 'p_' stand for a probability, varaibles starting with 'mm_' stand for the minimum and maximum value of an interval.

5 Selection of important model_parameters

Here we describe some of the most important model_parameters.

```
model_parameters['data']['n_folds']
```

determines the number of folds used in the cross-validation split.

```
model_parameters['data']['trn_dl_params']
model_parameters['data']['val_dl_params']
```

are the inputs used when creating the (pytorch based-) dataloaders for the training and validation set.

```
model_parameters['data']['folders']
model_parameters['data']['keys']
```

determine the folders of pre-processed data that are read from. These have to be changed when using cascades or deep supervision.

```
model_parameters['prediction']['mode']
```

can be used to turn the test time augmentations (flipping over each axis) on ('flip) or off ('simple').

```
model_parameters['training']['loss_params']
model_parameters['training']['opt_params']
model_parameters['training']['lr_params']
```

are used to create the loss function, optimizer and learning rate schedule respectively.

```
model_parameters['training']['num_epochs']
```

determins the length of the training. The batches per epoch is typically set to 250 (like in nnU-Net).

6 How to get started

The best way to get started is to use the scripts contained in ovseg.example_scripts. Make the first trainings quick e.g. by reducing the number of epochs, filters in the U-Net and folds in the cross-validation from 1000 to 100, 32 to 8 and 5 to 2 via

```
model_parameters['training']['num_epochs'] = 100
model_parameters['network']['filters'] = 8
model_parameters['data']['n_folds'] = 2
```

Next, it is best to start looking at the model_parameters.txt file and think about whether you understand all parameters. And lastly, play! Go out and explore the code, I hope you like it :)

Acknowledgement

This implementation wouldn't have been possible without using nnU-Net as a role model. Many thanks to the team for making their code publicly available.

References

[IJK+21] Fabian Isensee, Paul F. Jaeger, Simon A. A. Kohl, Jens Petersen, and Klaus H. maier Hain. nnu-net: a self-configuring method for deep learning-based biomedical image segmentation. *Nature Methods*, 18, 2021.