DeepFinder

Release 0.0.1

May 25, 2022

Contents:

CHAPTER 1

Introduction

DeepFinder is an original deep learning approach to localize macromolecules in cryo electron tomography images. The method is based on image segmentation using a 3D convolutional neural network.

1.1 Context

This method has been developed in the frame of **Emmanuel Moebel**'s PhD thesis at Inria Rennes, under **Charles Kervrann**'s supervision (team Serpico).

If you are curious, you can access his PhD manuscript here , entitled "New strategies for the identification and enumeration of macromolecules in 3D images of cryo electron tomography".

1.2 What DeepFinder is

DeepFinder is a python3 package that allows analysing **3D images** with a **deep learning** method. It possesses a **graphical user interface** to allow non-computer scientists to work with this tool. It is based on the Keras package.

CHAPTER 2

Guide

DeepFinder consists of 5 steps (blue boxes below), which constitute a workflow that allows to locate macromolecular complexes in crowded cells, when executed in depicted order. Each step can be executed either using a script (see examples/) or using the graphical user interface (see pyqt/). These steps may be used in other workflows, e.g. if the user needs only the segmentation step.

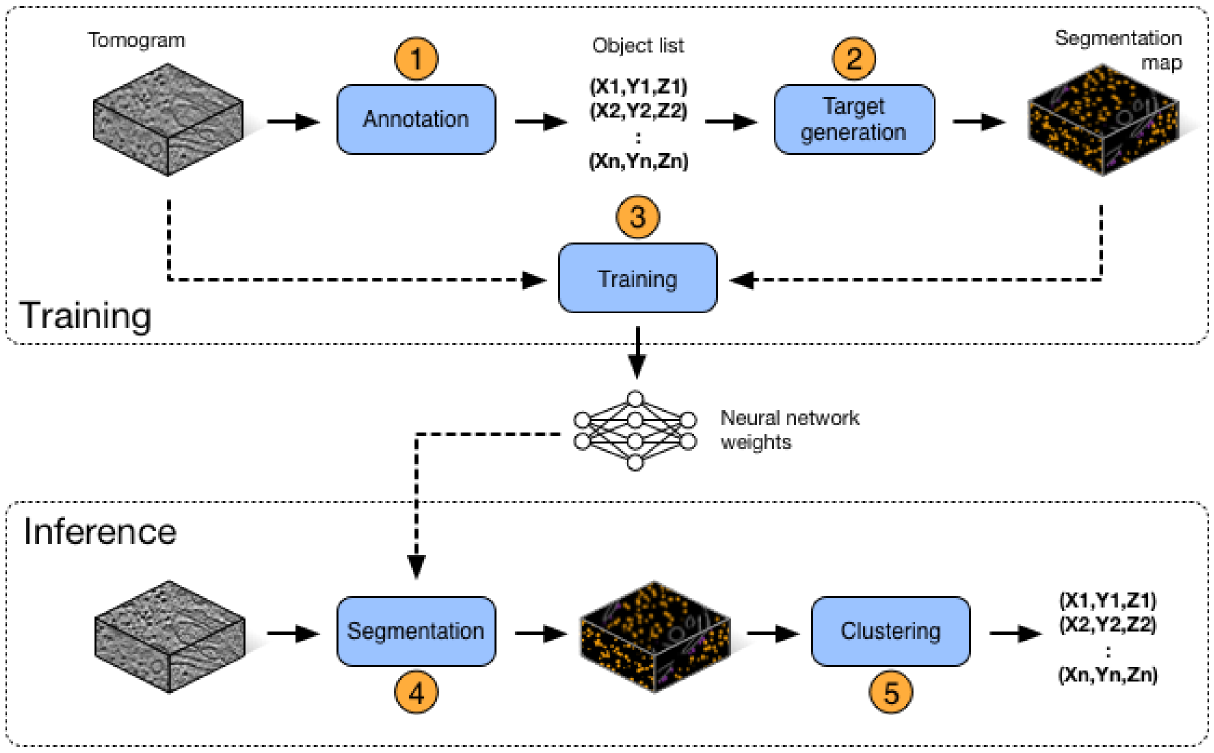


Fig. 1: DeepFinder workflow

Vocabulary:

- **Object list:** contains information on annotated or found macromolecular complexes
- **Label map:** segmentation map, i.e. a 3D array of integers {0,. . . ,Nclasses}, where '0' is reserved for the background class.
- **Target:** a label map used for training

Note: DeepFinder can read and write tomograms in **mrc**, **map** and **h5** formats.

2.1 Annotation

DeepFinder provides a GUI for annotating your tomograms. The goal is to constitute an object list per tomogram. The term 'object' corresponds here to a macromolecular complex.

2.1.1 The object list

An object list contains following information:

- Class label
- Coordinates (x,y,z)
- Orientation (phi,psi,theta): needed for generating shape targets
- Tomogram index: needed for training
- Cluster size: obtained after clustering step

A list contains at least class label and coordinates of each object. Other information (e.g. orientation etc) is included depending on considered operation (e.g. shape target generation).

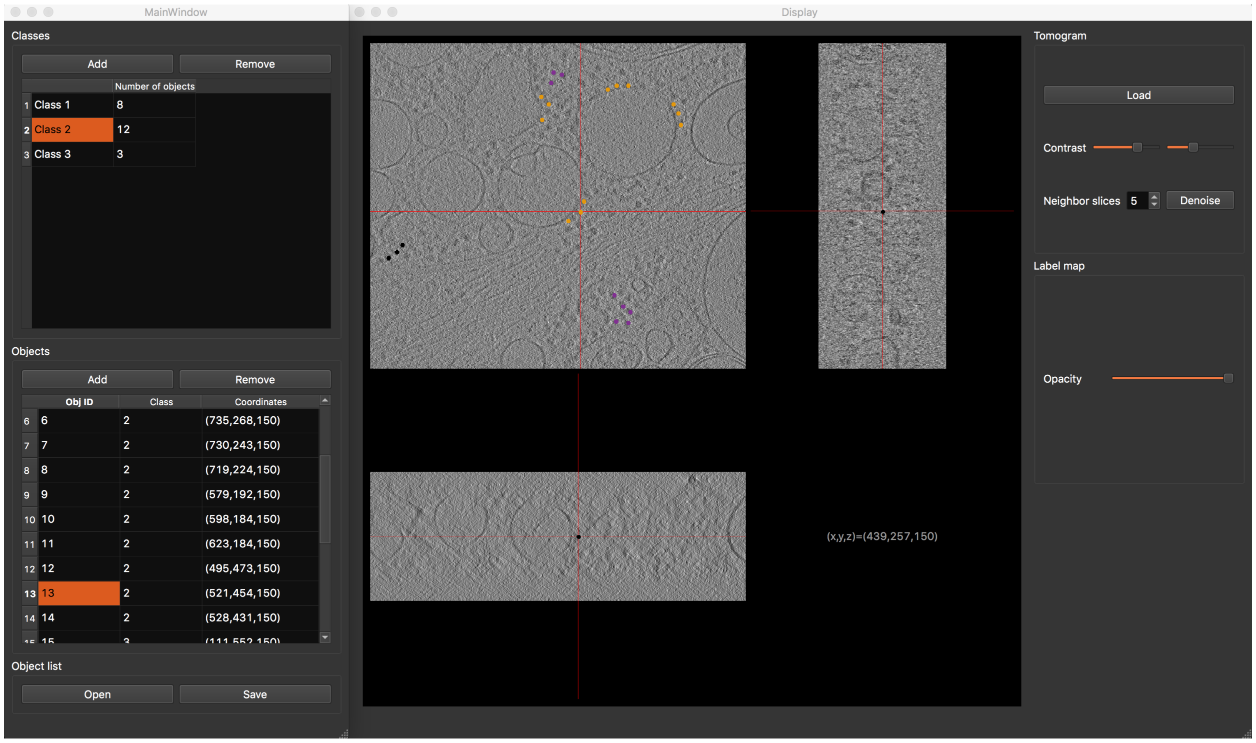
Note: An object list can be saved in **xlsx** or in **xml** format. This allows you to edit your object lists in Excel or in a text editor (for ex. for merging object lists).

Note: The API provides several functions for editing object lists in your scripts. You can for ex. scale coordinates, extract all objects from a specific class etc. See API section for more information.

2.1.2 Display window

This window allows to explore a tomogram with ortho-slices (i.e. 2D slices in each dimension). By clicking on the slices, you can scan through the volume in x, y and z directions. Furthermore, you can adjust the contrast interactively and denoise the slices to improve visibility. Denoising is performed by averaging neighboring slices, the number of neighbors being configurable by the user.

Also, a label map (i.e. segmentation map) can be superimposed on the tomogram slices and its opacity can be adapted.



		Number of objects
1	Class 1	8
2	Class 2	12
3	Class 3	3

	Obj ID	Class	Coordinates
6	6	2	(735,268,150)
7	7	2	(730,243,150)
8	8	2	(719,224,150)
9	9	2	(579,192,150)
10	10	2	(598,184,150)
11	11	2	(623,184,150)
12	12	2	(495,473,150)
13	13	2	(521,454,150)
14	14	2	(528,431,150)
15	15	3	(111,552,150)

(x,y,z)=(439,257,150)

Fig. 2: Annotation GUI

2.1.3 Annotation GUI

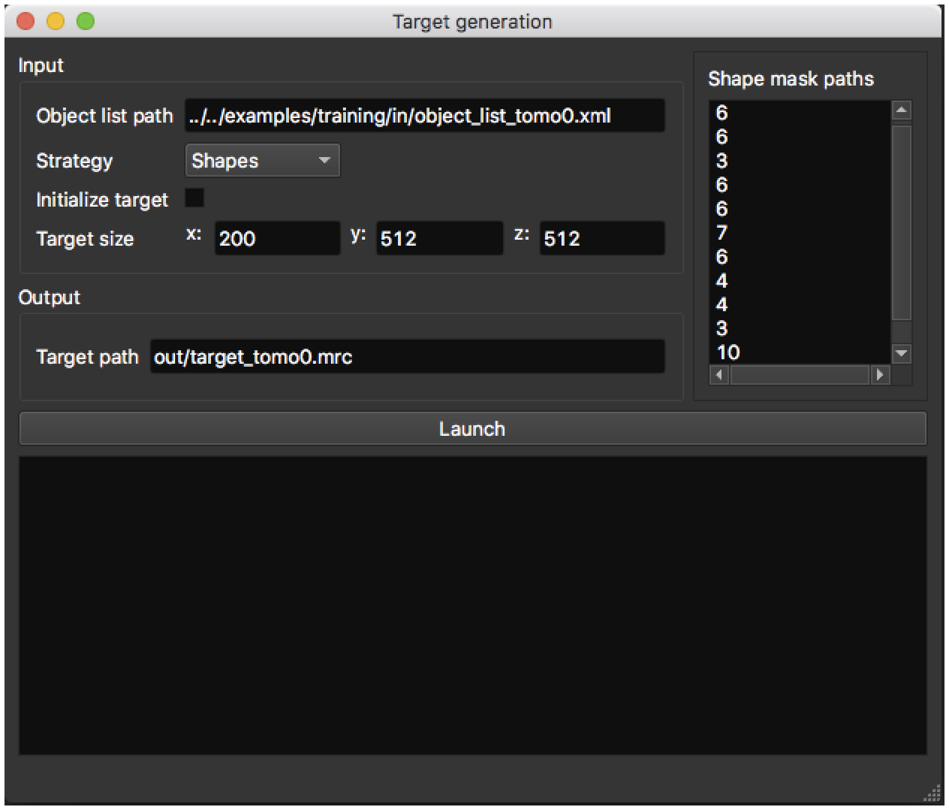
The annotation graphical interface is constituted of a display window and an annotation window, and allows you to interactively annotate your tomograms. First, select desired class. Then browse your tomogram to find instances of desired macromolecule species, and double-clic on their position. You will see a dot appearing on the tomogram slices and a row appear in the object table, confirming that the position has been saved. Repeat the process until all visible macromolecules have been annotated. To finish, save your object list. In the same way, annotate as many tomograms as you can, then proceed to the next step.

Note: As annotation is a time-consuming task, you can save the current state and resume the task where you left it by saving the object list.

2.2 Target generation

This step converts an object list (i.e. position-wise annotations) into a label map (i.e. voxel-wise annotation). We propose two strategies to do so: **spheres** and **shapes**.

- **Object list path**: path to the object list obtained from the annotation procedure.
- **Initialize target**: allow you to initialize the target with an array already containing annotated structures like membranes.
- **Target size** (in voxels): if you don't use the target initialization option, you need to specify the size of your target volume, which should be the same size as the tomogram it describes.
- **Target path**: where the target volume should be saved.



Object list path ../../examples/training/in/object_list_tomo0.xml
Strategy Shapes
Initialize target
Target size x: 200 y: 512 z: 512
Target path out/target_tomo0.mrc

```
6
6
3
6
6
7
6
4
4
3
10
```

Fig. 3: Target generation GUI

2.2.1 Sphere targets

Here, targets are generated by placing a sphere at positions contained in the object list. You can specify a different radius per class. This radius should correspond to the size of the object. This technique is quick to execute in comparison to 'shapes' and yields decent results.

- **Radius list** (in voxels): sphere radius per class. The list order should correspond to the class label as follows: 1st line -> radius of class 1 ; 2nd line -> radius of class 2 . . .

2.2.2 Shape targets

This strategy is more precise but needs more time and external tools to execute. Instead of using spheres, more precise masks (corresponding to macromolecule shapes) are placed at specified positions. However, to obtain these masks and also the orientation of each object, a sub-tomogram averaging procedure is needed (as available in PyTOM or Scipion). So using this strategy involves more efforts and time, but yields better results, especially for small objects.

- **Shape mask paths**: list of mask paths (1 mask per class). The masks are 3D arrays which contain the shape of macromolecules ('1' for 'is object' and '0' for 'is not object'). The path order should correspond to the class label as follows: 1st line -> path to mask of class 1 ; 2nd line -> path to mask of class 2 . . .

Warning: When the 'shapes' strategy is selected, the object list needs to contain the orientation (i.e. Euler angles) of each object.

2.3 Training

Note: If you are a beginner in deep learning, and would like to gain a general understanding, you can read Section 3 "An introduction to deep learning" of E. Moebel's (author of DeepFinder) PhD thesis .

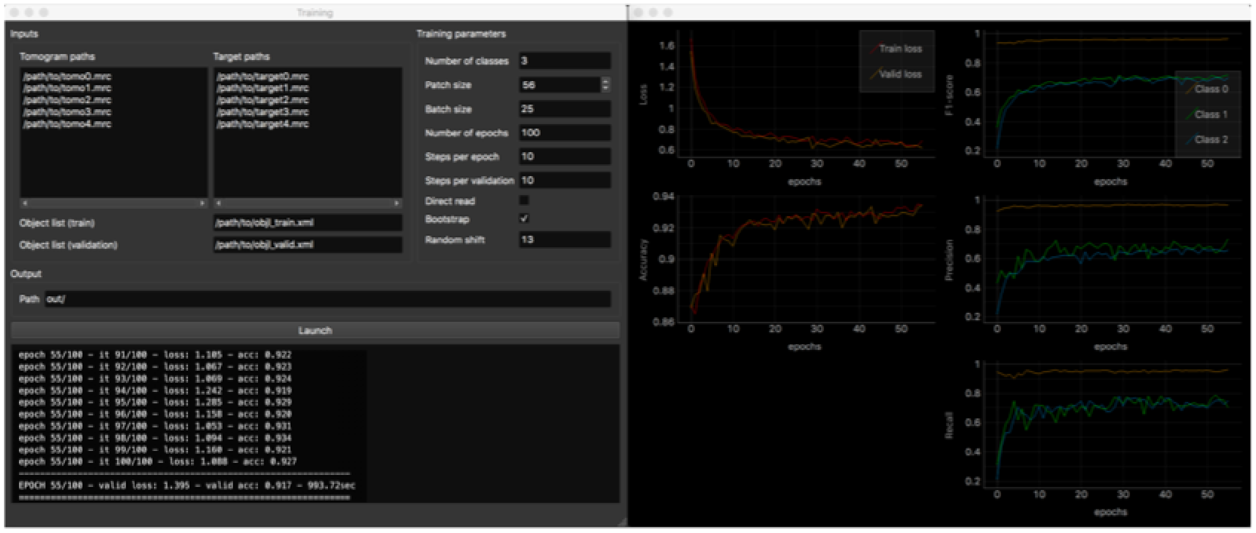
Before running the training procedure, it is good practice to define a validation set, which is a subset of your training set. Then, this validation set will not be used for training, but for computing metrics to evaluate training performance. This is helpful for checking for **overfitting**. Intuitively, overfitting happens when instead of learning discriminating features of objects, the network learns them by heart. Consequently, like a bad student, the network is unable to generalize its knowledge to new data and produces a classification of poor quality. You can detect overfitting by comparing training loss and validation loss curves (or accuracy curves). If they have similar values, then training is efficient. If they diverge, then there is overfitting.

You can define which of your annotated objects you want to use for training and for validation by storing them in separate object lists (see image below). Ideally, the validation objects should originate from a different tomogram than the training objects. If this is not possible, try to choose validation objects that are not too close to training objects. The minimum size of validation set should be **at least** few dozen objects per class, **ideally** a few hundreds.

- **Tomogram and target paths**: list here the paths to the tomograms an their corresponding targets. Should correspond line per line
- **Object list paths**: tomogram index in these object lists should correspond to the order of above listed tomo/target pairs.
- **Output path**: where network weights and training metrics will be saved.

Training parameters:

- **Number of classes** (background class included)



Tomogram paths:
/path/to/tomo0.mrc
/path/to/tomo1.mrc
/path/to/tomo2.mrc
/path/to/tomo3.mrc
/path/to/tomo4.mrc

Target paths:
/path/to/target0.mrc
/path/to/target1.mrc
/path/to/target2.mrc
/path/to/target3.mrc
/path/to/target4.mrc

Object list (train): /path/to/objl_train.xml
Object list (validation): /path/to/objl_valid.xml

Number of classes: 3
Patch size: 56
Batch size: 25
Number of epochs: 100
Steps per epoch: 10
Steps per validation: 10
Direct read: ☐
Bootstrap: ✓
Random shift: 13

Path: out/

```
epoch 55/100 - it 91/100 - loss: 1.105 - acc: 0.922
epoch 55/100 - it 92/100 - loss: 1.067 - acc: 0.923
epoch 55/100 - it 93/100 - loss: 1.069 - acc: 0.924
epoch 55/100 - it 94/100 - loss: 1.242 - acc: 0.919
epoch 55/100 - it 95/100 - loss: 1.285 - acc: 0.929
epoch 55/100 - it 96/100 - loss: 1.158 - acc: 0.920
epoch 55/100 - it 97/100 - loss: 1.053 - acc: 0.931
epoch 55/100 - it 98/100 - loss: 1.094 - acc: 0.934
epoch 55/100 - it 99/100 - loss: 1.160 - acc: 0.921
epoch 55/100 - it 100/100 - loss: 1.088 - acc: 0.927
-------------------------------------------------------------
EPOCH 55/100 - valid loss: 1.395 - valid acc: 0.917 - 993.72sec
=============================================================
```

Fig. 4: Training GUI

- **Patch size** (in voxels): must be a multiple of 4, due to the network architecture.
- **Batch size**: number of patches used to compute average loss.
- **Number of epochs**: at the end of each epoch, evaluation on validation set is performed (usefull to check if network overfits).
- **Steps per epoch**: number of batches trained on per epoch. In the end, the total number of training iterations is [number of epochs]x[steps per epoch].
- **Steps per validation**: number of batches used for validation.
- **Direct read**: if checked, only the current batch is loaded into memory, instead of the whole dataset. Usefull when running out of memory. Transmission speed between dataset storage and GPU should be high enough.
- **Bootstrap**: if checked, applies re-sampling to batch generation, so that each class has an equal chance to be sampled. Usefull when in presence of unbalanced classes. Can remain checked.
- **Random shift** (in voxels): applied to positions in object list when sampling patches. Enhances network robustness. Make sure that objects are still contained in patches after the shift is applied.

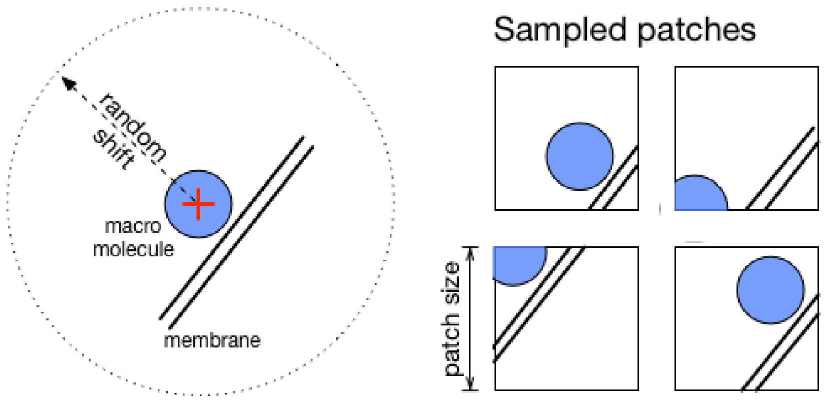


Fig. 5: Patch sampling and random shift

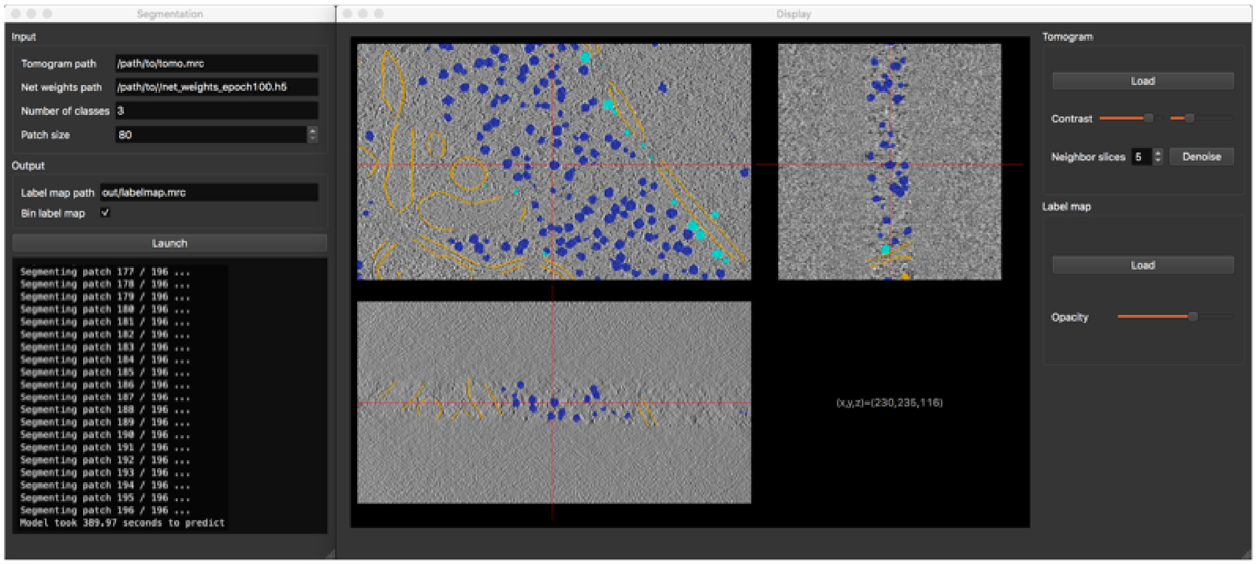
Once you filled out all required fields, hit the ‘Launch’ button. A second window will appear, displaying training metrics in real time, allowing you to monitor the progress of the procedure. Metrics per class are computed for the validation set (F1-score, precision, recall). It is common to obtain per-class score values around 0.6, which for our datasets was enough for satisfying localization. Indeed, even if macromolecules are segmented only partially, it is enough to find them in the Clustering step.

Note: If your machine runs **out of memory**, you can reduce patch size and batch size values.

Note: Every 10 epochs, the network weights are saved at the output path. If your training procedure is interrupted for any reason, this allows you to resume the training at last saved network state, instead of starting over from scratch.

2.4 Segmentation

Now that your network is trained, it is time to apply it to segment new tomograms. As a tomogram is too large to be processed in one take, the procedure splits the volume in smaller overlapping 3D patches. You can adapt the patch size to the available memory on your machine.


Input
Tomogram path: /path/to/tomo.mrc
Net weights path: /path/to//net_weights_epoch100.h5
Number of classes: 3
Patch size: 80
Output
Label map path: out/labelmap.mrc
Bin label map: ✓

```
Segmenting patch 177 / 196 ...
Segmenting patch 178 / 196 ...
Segmenting patch 179 / 196 ...
Segmenting patch 180 / 196 ...
Segmenting patch 181 / 196 ...
Segmenting patch 182 / 196 ...
Segmenting patch 183 / 196 ...
Segmenting patch 184 / 196 ...
Segmenting patch 185 / 196 ...
Segmenting patch 186 / 196 ...
Segmenting patch 187 / 196 ...
Segmenting patch 188 / 196 ...
Segmenting patch 189 / 196 ...
Segmenting patch 190 / 196 ...
Segmenting patch 191 / 196 ...
Segmenting patch 192 / 196 ...
Segmenting patch 193 / 196 ...
Segmenting patch 194 / 196 ...
Segmenting patch 195 / 196 ...
Segmenting patch 196 / 196 ...
Model took 389.97 seconds to predict
```

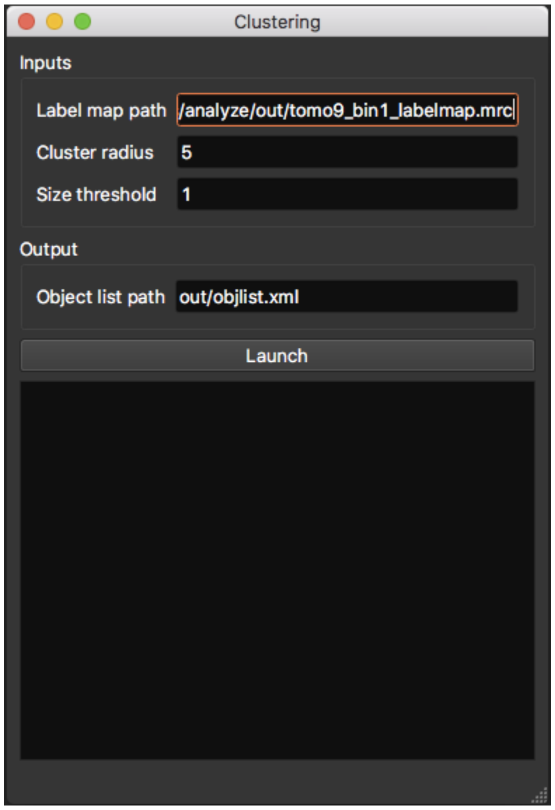
Fig. 6: Segmentation GUI

- **Tomogram path**
- **Net weights path**: path to the .h5 file containing the network weights obtained by the training procedure.
- **Number of classes** (background class included)
- **Patch size** (in voxels): must be a multiple of 4, due to the network architecture.
- **Label map path**: where the segmented tomogram should be saved.
- **Bin label map**: when checked, also saves a sub-sampled version of the label map. Smaller label maps reduces computing time of clustering step.

Once the segmentation is achieved, a display window appears, allowing you to check the consistency of the result.

2.5 Clustering

This procedure analyzes the segmented tomogram (i.e. label map), identifies individual macromolecules and outputs their coordinates, stored as an object list. This analysis is achieved with the mean-shift clustering algorithm.



Label map path /analyze/out/tomo9_bin1_labelmap.mrc
Cluster radius 5
Size threshold 1
Object list path out/objlist.xml

Fig. 7: Clustering GUI

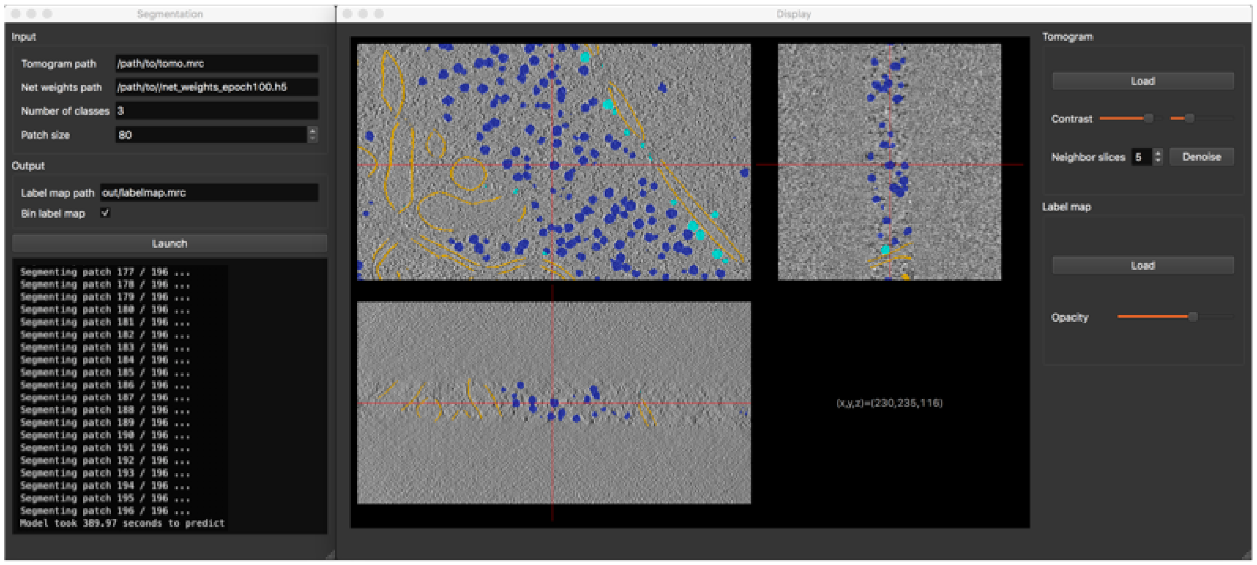
- **Label map path**: path to input label map.
- **Cluster radius** (in voxels): parameter for clustering algorithm. Corresponds to average object radius.
- **Object list path**: where the output object list should be saved.

CHAPTER 3

Tutorial

This part describes how to reproduce the segmentations obtained in our paper, using pre-trained weights. First, please follow installation instructions .

Next, launch the segmentation GUI by typing following command into the terminal: `/path/to/deep-finder/bin/segment`



Tomogram path: /path/to/tomo.mrc
Net weights path: /path/to//net_weights_epoch100.h5
Number of classes: 3
Patch size: 80
Label map path: out/labelmap.mrc
Bin label map: ✓
Neighbor slices: 5

```
Segmenting patch 177 / 196 ...
Segmenting patch 178 / 196 ...
Segmenting patch 179 / 196 ...
Segmenting patch 180 / 196 ...
Segmenting patch 181 / 196 ...
Segmenting patch 182 / 196 ...
Segmenting patch 183 / 196 ...
Segmenting patch 184 / 196 ...
Segmenting patch 185 / 196 ...
Segmenting patch 186 / 196 ...
Segmenting patch 187 / 196 ...
Segmenting patch 188 / 196 ...
Segmenting patch 189 / 196 ...
Segmenting patch 190 / 196 ...
Segmenting patch 191 / 196 ...
Segmenting patch 192 / 196 ...
Segmenting patch 193 / 196 ...
Segmenting patch 194 / 196 ...
Segmenting patch 195 / 196 ...
Segmenting patch 196 / 196 ...
Model took 389.97 seconds to predict
```

Fig. 1: Segmentation GUI

In the segmentation window (left), please fill out requested fields as follows:

3.1 SHREC'19 dataset

- **Tomogram path**: `/path/to/deep-finder/examples/analyze/in/tomo9.mrc`

- **Net weights path**: `/path/to/deep-finder/examples/analyze/in/net_weights_FINAL.h5`
- **Number of classes**: 13

3.2 Chlamydomonas dataset

Download the example tomogram here.

- **Tomogram path**: `/path/to/download/emd_3967.map`
- **Net weights path**: `/path/to/deep-finder/examples/training/out/net_weights_chlamydomonas.h5`
- **Number of classes**: 4

After setting the patch size and your output path, click on button **Launch**. Progress about computation should be printed in the box below the button. Once computation is finished, the display window should pop up, showing the obtained segmentation, super-imposed with the tomogram, allowing you to inspect the result. For more details about segmentation and display windows, please see our *Guide*.

CHAPTER 4

Command line tools

This page gives instructions on how to launch DeepFinder steps from the terminal.

4.1 Set up

First, add DeepFinder to your path with following command: `export PATH="/path/to/deep-finder/bin:$PATH"`

You can add this command to your `~/.bash_profile`

Note: Running these commands without any argument will launch the graphical user interface.

4.2 Annotation

Usage:

```
annotate -t /path/to/tomogram.mrc
         -o /path/to/output/object_list.xml
```

4.3 Target generation

For `generate_target` and `train`, it is not possible to pass all necessary parameters as terminal arguments. Therefore, they have to be passed as an xml file.

Usage:

```
generate_target -p /path/to/parameters.xml
```

Parameter file:

```
<paramsGenerateTarget>
  <path_objl path="/path/to/objl.xml"/>
  <path_initial_vol path=""/>
  <tomo_size>
    <X size="400"/>
    <Y size="400"/>
    <Z size="200"/>
  </tomo_size>
  <strategy strategy="spheres"/>
  <radius_list>
    <class1 radius="1"/>
    <class2 radius="2"/>
    <class3 radius="3"/>
  </radius_list>
  <path_mask_list>
    <class1 path=""/>
  </path_mask_list>
  <path_target path="/path/to/target.mrc"/>
</paramsGenerateTarget>
```

Note: You can find classes with methods for automatically reading and writing these parameter files in utils/params.py

4.4 Training

Usage:

```
train -p /path/to/parameters.xml
```

Parameter file:

```
<paramsTrain>
  <path_out path="./"/>
  <path_tomo>
    <tomo0 path="/path/to/tomo0.mrc"/>
    <tomo1 path="/path/to/tomo1.mrc"/>
    <tomo2 path="/path/to/tomo2.mrc"/>
  </path_tomo>
  <path_target>
    <target0 path="/path/to/target0.mrc"/>
    <target1 path="/path/to/target1.mrc"/>
    <target2 path="/path/to/target2.mrc"/>
  </path_target>
  <path_objl_train path="/path/to/objl_train.xml"/>
  <path_objl_valid path="/path/to/objl_valid.xml"/>
  <number_of_classes n="3"/>
  <patch_size n="48"/>
  <batch_size n="20"/>
  <number_of_epochs n="100"/>
  <steps_per_epoch n="100"/>
  <steps_per_validation n="10"/>
  <flag_direct_read flag="False"/>
  <flag_bootstrap flag="True"/>
```

(continues on next page)

(continued from previous page)

```
  <random_shift shift="13"/>
</paramsTrain>
```

4.5 Segmentation

Usage:

```
segment -t /path/tomogram.mrc
        -w /path/net_weights.h5
        -c NCLASS
        -p PSIZE
        -o /path/output/segmentation.mrc
```

With NCLASS and PSIZE integer values. See *Guide* for parameter description.

4.6 Clustering

Usage:

```
cluster -l /path/to/segmentation.mrc
        -r clusterRadius
        -o /path/to/output/object_list.xml
```

CHAPTER 5

API

5.1 DeepFinder

Each step of the DeepFinder workflow is coded as a class. The parameters of each method are stored as class attributes and are given default values in the constructor. These parameters can easily be given custom values as follows:

```
from deepfinder.training import Train
trainer = Train(Ncl=5, dim_in=56) # initialize training task, where default batch_
↪size=25
trainer.batch_size = 16 # customize batch_size value
```

Each class has a main method called 'launch' to execute the procedure. These classes all inherit from a mother class 'DeepFinder' that possesses features useful for communicating with the GUI.

5.1.1 Training

class `deepfinder.training.`**`TargetBuilder`**

`generate_with_shapes`(*objl*, *target_array*, *ref_list*)
Generates segmentation targets from object list. Here macromolecules are annotated with their shape.

Parameters

- **`objl`** (*`list of dictionaries`*) – Needs to contain [phi,psi,the] Euler angles for orienting the shapes.
- **`target_array`** (*`3D numpy array`*) – array that initializes the training target. Allows to pass an array already containing annotated structures like membranes. index order of array should be [z,y,x]
- **`ref_list`** (*`list of 3D numpy arrays`*) – These reference arrays are expected to be cubic and to contain the shape of macromolecules ('1' for 'is object' and '0' for 'is

not object') The references order in list should correspond to the class label. For ex: 1st element of list -> reference of class 1; 2nd element of list -> reference of class 2 etc.

Returns Target array, where '0' for background class, {'1','2',…} for object classes.

Return type 3D numpy array

`generate_with_spheres` (*objl*, *target_array*, *radius_list*)

Generates segmentation targets from object list. Here macromolecules are annotated with spheres. This method does not require knowledge of the macromolecule shape nor Euler angles in the objl. On the other hand, it can be that a network trained with 'sphere targets' is less accurate than with 'shape targets'.

Parameters

- **`objl`** (*`list of dictionaries`*) –
- **`target_array`** (*`3D numpy array`*) – array that initializes the training target. Allows to pass an array already containing annotated structures like membranes. index order of array should be [z,y,x]
- **`radius_list`** (*`list of int`*) – contains sphere radii per class (in voxels). The radii order in list should correspond to the class label. For ex: 1st element of list -> sphere radius for class 1, 2nd element of list -> sphere radius for class 2 etc.

Returns Target array, where '0' for background class, {'1','2',…} for object classes.

Return type 3D numpy array

`class` `deepfinder.training.`**`Train`** (*Ncl*, *dim_in*)

`launch` (*path_data*, *path_target*, *objlist_train*, *objlist_valid*)

This function launches the training procedure. For each epoch, an image is plotted, displaying the progression with different metrics: loss, accuracy, f1-score, recall, precision. Every 10 epochs, the current network weights are saved.

Parameters

- **`path_data`** (*`list of string`*) – contains paths to data files (i.e. tomograms)
- **`path_target`** (*`list of string`*) – contains paths to target files (i.e. annotated volumes)
- **`objlist_train`** (*`list of dictionaries`*) – contains information about annotated objects (e.g. class, position) In particular, the tomo_idx should correspond to the index of 'path_data' and 'path_target'. See utils/objl.py for more info about object lists. During training, these coordinates are used for guiding the patch sampling procedure.
- **`objlist_valid`** (*`list of dictionaries`*) – same as 'objlist_train', but objects contained in this list are not used for training, but for validation. It allows to monitor the training and check for over/under-fitting. Ideally, the validation objects should originate from different tomograms than training objects.

Note:

The function saves following files at regular intervals: net_weights_epoch*.h5: contains current network weights

net_train_history.h5: contains arrays with all metrics per training iteration

net_train_history_plot.png: plotted metric curves

5.1.2 Inference

class `deepfinder.inference.`**`Segment`**(*Ncl*, *path_weights*, *patch_size=192*)

`launch`(*dataArray*)
This function enables to segment a tomogram. As tomograms are too large to be processed in one take, the tomogram is decomposed in smaller overlapping 3D patches.

Parameters

- **`dataArray`** (*`3D numpy array`*) – the volume to be segmented
- **`weights_path`** (*`str`*) – path to the .h5 file containing the network weights obtained by the training procedure

Returns contains predicted score maps. Array with index order [class,z,y,x]

Return type numpy array

class `deepfinder.inference.`**`Cluster`**(*clustRadius*)

`launch`(*labelmap*)
This function analyzes the segmented tomograms (i.e. labelmap), identifies individual macromolecules and outputs their coordinates. This is achieved with a clustering algorithm (meanshift).

Parameters

- **`labelmap`** (*`3D numpy array`*) – segmented tomogram
- **`clustRadius`** (*`int`*) – parameter for clustering algorithm. Corresponds to average object radius (in voxels)

Returns the object list with coordinates and class labels of identified macromolecules

Return type list of dict

5.2 Utilities

5.2.1 Common utils

`deepfinder.utils.common.`**`bin_array`**(*array*)
Subsamples a 3D array by a factor 2. Subsampling is performed by averaging voxel values in 2x2x2 tiles.

Parameters **`array`** (*`numpy array`*) –

Returns binned array

Return type numpy array

`deepfinder.utils.common.`**`plot_volume_orthoslices`**(*vol*, *filename*)
Writes an image file containing ortho-slices of the input volume. Generates same visualization as matlab function ‘tom_volxyz’ from TOM toolbox. If volume type is int8, the function assumes that the volume is a labelmap, and hence plots in color scale. Else, it assumes that the volume is tomographic data, and plots in gray scale.

Parameters

- **`vol`** (*`3D numpy array`*) –
- **`filename`** (*`str`*) – ‘/path/to/file.png’

`deepfinder.utils.common.`**`read_array`**(*filename*, *dset_name='dataset'*)

Reads arrays. Handles .h5 and .mrc files, according to what extension the file has.

Parameters

- **`filename`** (*`str`*) – '/path/to/file.ext' with '.ext' either '.h5' or '.mrc'
- **`dset_name`** (*`str, optional`*) – h5 dataset name. Not necessary to specify when reading .mrc

Returns numpy array

`deepfinder.utils.common.`**`write_array`**(*array*, *filename*, *dset_name='dataset'*)

Writes array. Can write .h5 and .mrc files, according to the extension specified in filename.

Parameters

- **`array`** (*`numpy array`*) –
- **`filename`** (*`str`*) – '/path/to/file.ext' with '.ext' either '.h5' or '.mrc'
- **`dset_name`** (*`str, optional`*) – h5 dataset name. Not necessary to specify when reading .mrc

5.2.2 Object list utils

`deepfinder.utils.objl.`**`above_thr`**(*objlIN*, *thr*)

Parameters

- **`objl`** (*`list of dict`*) –
- **`thr`** (*`float`*) – threshold

Returns contains only objects with cluster size >= thr

Return type list of dict

`deepfinder.utils.objl.`**`disp`**(*objlIN*)

Prints objl in terminal

`deepfinder.utils.objl.`**`get_class`**(*objlIN*, *label*)

Get all objects of specified class.

Parameters

- **`objl`** (*`list of dict`*) –
- **`label`** (*`int`*) –

Returns contains only objects from class 'label'

Return type list of dict

`deepfinder.utils.objl.`**`get_labels`**(*objlIN*)

Returns a list with different (unique) labels contained in input objl

`deepfinder.utils.objl.`**`get_obj`**(*objl*, *obj_id*)

Get objects with specified object ID.

Parameters

- **`objl`** (*`list of dict`*) – input object list
- **`obj_id`** (*`list of int`*) – object ID of wanted object(s)

Returns contains object(s) with obj ID 'obj_id'

Return type list of dict

`deepfinder.utils.objl.`**`get_tomo`**`(`*objlIN*, *tomo_idx*`)`

Get all objects originating from tomo 'tomo_idx'.

Parameters

- **`objlIN`** (*`list of dict`*) – contains objects from various tomograms
- **`tomo_idx`** (*`int`*) – tomogram index

Returns contains objects from tomogram 'tomo_idx'

Return type list of dict

`deepfinder.utils.objl.`**`read`**`(`*filename*`)`

Reads object list. Handles .xml and .xlsx files, according to what extension the file has.

Parameters **`filename`** (*`str`*) – '/path/to/file.ext' with '.ext' either '.xml' or '.xlsx'

Returns list of dict

`deepfinder.utils.objl.`**`remove_class`**`(`*objl*, *label_list*`)`

Removes all objects from specified classes.

Parameters

- **`objl`** (*`list of dict`*) – input object list
- **`label_list`** (*`list of int`*) – label of objects to remove

Returns same as input object list but with objects from classes 'label_list' removed

Return type list of dict

`deepfinder.utils.objl.`**`remove_obj`**`(`*objl*, *obj_id*`)`

Removes objects by object ID.

Parameters

- **`objl`** (*`list of dict`*) – input object list
- **`obj_id`** (*`list of int`*) – object ID of wanted object(s)

Returns same as input object list but with object(s) 'obj_id' removed

Return type list of dict

`deepfinder.utils.objl.`**`scale_coord`**`(`*objlIN*, *scale*`)`

Scales coordinates by specified factor. Useful when using binned (sub-sampled) volumes, where coordinates need to be multiplied or divided by 2.

Parameters

- **`objlIN`** (*`list of dict`*) –
- **`scale`** (*`float, int or tuple`*) – if float or int, same scale is applied to all dim

Returns object list with scaled coordinates

Return type list of dict

`deepfinder.utils.objl.`**`write`**`(`*objl*, *filename*`)`

Writes object list. Can write .xml and .xlsx files, according to the extension specified in filename.

Parameters

- **`objl`** (*`list of dict`*) –

- **`filename`** (*`str`*) – '/path/to/file.ext' with '.ext' either '.xml' or '.xlsx'

5.2.3 Scoremap utils

`deepfinder.utils.smap.`**`bin`**(*scoremaps*)

Subsamples the scoremaps by a factor 2. Subsampling is performed by averaging voxel values in 2x2x2 tiles.

Parameters **`scoremaps`** (*`4D numpy array`*) – array with index order [class,z,y,x]

Returns 4D numpy array

`deepfinder.utils.smap.`**`read_h5`**(*filename*)

Reads scormaps stored in .h5 file.

Parameters **`filename`** (*`str`*) – path to file This .h5 file has one dataset per class (dataset '/class*' contains scoremap of class *)

Returns scoremaps array with index order [class,z,y,x]

Return type 4D numpy array

`deepfinder.utils.smap.`**`to_labelmap`**(*scoremaps*)

Converts scoremaps into a labelmap.

Parameters **`scoremaps`** (*`4D numpy array`*) – array with index order [class,z,y,x]

Returns array with index order [z,y,x]

Return type 3D numpy array

`deepfinder.utils.smap.`**`write_h5`**(*scoremaps*, *filename*)

Writes scoremaps in .h5 file

Parameters

- **`scoremaps`** (*`4D numpy array`*) – array with index order [class,z,y,x]
- **`filename`** (*`str`*) – path to file This .h5 file has one dataset per class (dataset '/class*' contains scoremap of class *)

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index