

DeepNebulae Documentation Release 1.0.0

November 15, 2024

Contents

| | |
|---|----|
| Demo 1 – Learning a localization model | 2 |
| Recovering the experimental phase masks | 2 |
| Experimental SNR estimation | 3 |
| Setting the simulation parameters | 4 |
| Training a localization CNN | 11 |
| Localization using a trained CNN | 11 |
| References | 13 |

Demo 1 – Learning a localization model

This demo is intended to get you started using DeepNebulae. Given a new optical setup, new phase masks, fluorescent dye, objective lens, etc. you need to train a new neural network for localization. In order to do so, you need to create a training set that can be fed into the training process. The involved steps are listed below with accompanying snapshots to ease the process.

Recovering the experimental phase masks

To calibrate the experimental phase masks, you will need to acquire a z-stack of a bead on the coverslip, with each mask, in a 3D aligned fashion. These z-stacks should cover the entire z-range that the user intends to recover in the actual experiment. An example of such a z-stack with the Tetrapod PSF is given in **Fig. 1**.



Figure. 1 – 5 slices taken from a z-stack of a fluorescent microsphere that was used to calibrate the Tetrapod PSF in one of our experiments. Scale bar is 2 microns.

Next, to retrieve the phase using the measurements in **Fig. 1**, the user should use the VIPR [1] phase retrieval method for each z-stack. The code of VIPR is written in MATLAB 2019a, and is publicly available¹. For more details on using this software given the acquired z-stacks (**Fig. 1**) please refer to its own documentation. The expected output from VIPR for each z-stack is a phase mask modelling the experimental system's output in the corresponding imaging arm, as shown in **Fig. 2**.

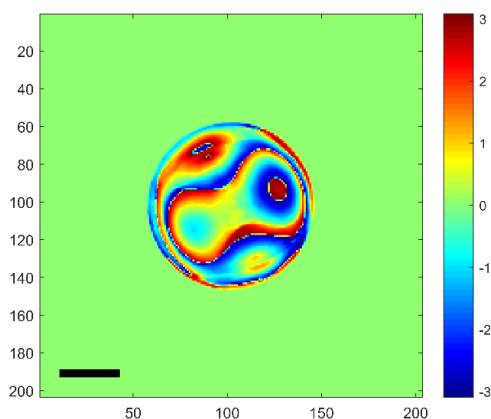


Figure. 2 - phase mask retrieved by VIPR[1]. SLM pixel size is 30 microns. Scale bar is 1 mm.

¹ <https://github.com/Borisfer/VIPR---Vectorial-Phase-Retrieval-for-microscopy>

Experimental SNR estimation

To train a net to localize the experimental data we need to match the signal to noise ratio in simulations. To get a rough estimate of the baseline and the background standard deviation we can examine emitter-empty regions in the experimental frames (**Fig. 3a,b** red square). Similarly, the signal intensity can be estimated from subtracting the mean background counts in emitter-occupied regions (**Fig. 3a,b** green square/rectangle). The ratio between the green rectangles is used later on to determine the efficiency of signal splitting with polarizing beam splitter in our simulation parameters.

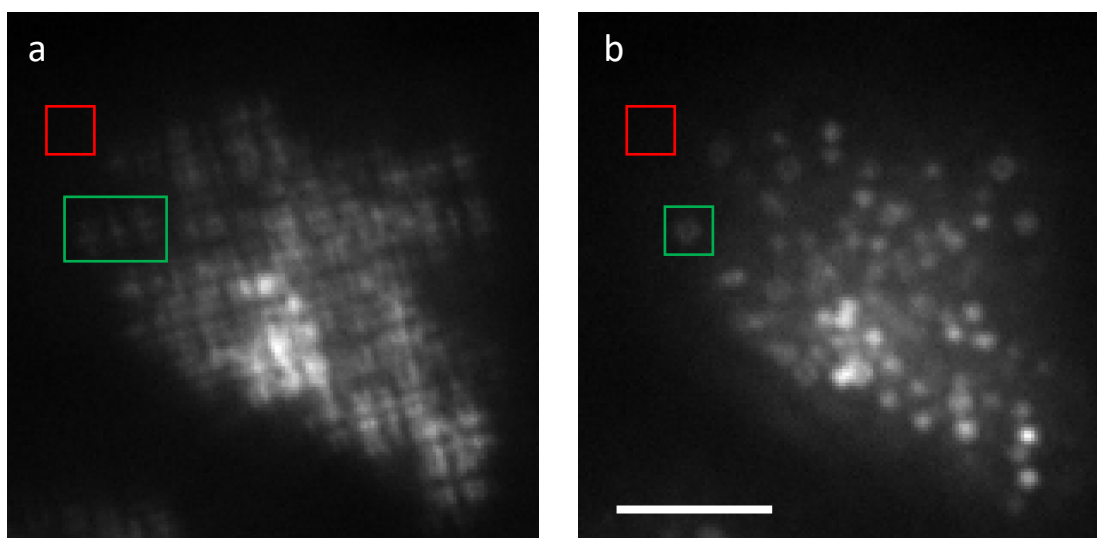


Figure. 3 – SNR determination in Fiji [2,3]. a) Experimental image from camera 1 with the Tetrapod PSF. b) Experimental image from camera 2 with the EDOF PSF. Scale bar is 5 microns.

Channel registration

For registration of both imaging channels we image a bead sample, localize the results in ThunderSTORM [4], and then estimate an affine transformation mapping camera 1 to camera 2 (**Fig. 4**). See further discussion in the README file.

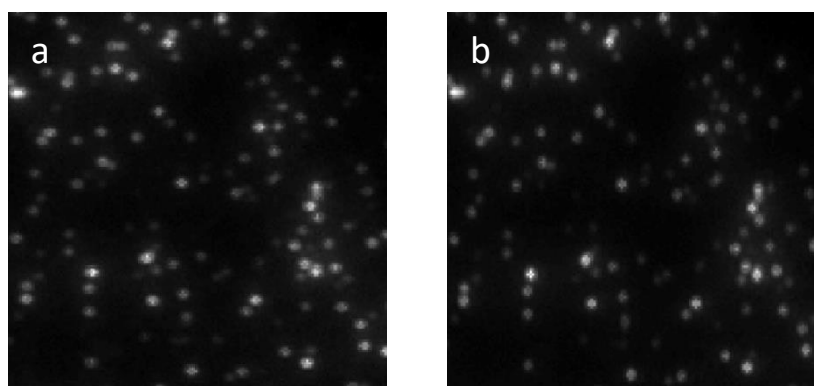


Figure. 4 – Transform calibration. a-b) Experimental images of a bead sample in camera 1/2 respectively.

Setting the simulation parameters

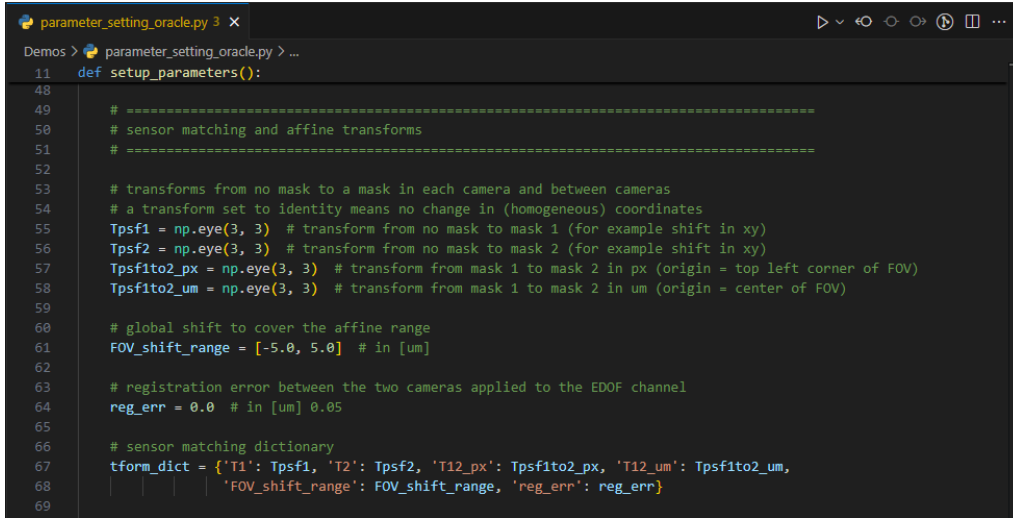
Now that we have characterized the experimental conditions, we can train a localization CNN. The simulation parameters needed to simulate the training data and learn the localization CNN are encapsulated in the script “Demos/parameter_setting_oracle.py”. Next, let us go over the list of parameters and discuss their role:

1. First section is used to set the mode of training: either learning optimized phase masks or learning a localization CNN. In case we are learning a localization CNN, the Boolean flags in “learn_mask” (**Fig. 5** blue rectangle) are set to False, and the “initial” masks is set to the masks recovered by VIPR (**Fig. 5** green rectangle).
2. Second section is used to specify the optical system and the imaging assumptions (**Fig. 5** red rectangle). These include: the fluorophore mean emission wavelength (lamda) in microns, the objective numerical aperture (NA), the immersion oil refractive index (noil), the imaging medium refractive index (nwater), the pixel size of the sensor in microns (pixel_size_CCD), the pixel size of the spatial light modulator in microns (pixel_size_SLM), the optical magnification (M), and the 4f lenses focal length in microns (f_4f).

```
parameter_setting_oracle.py 3 x
Demos > parameter_setting_oracle.py > ...
4 import os
5 import numpy as np
6 from math import pi
7 import scipy.io as sio
8 import torch
9
10
11 def setup_parameters():
12     # path to current directory
13     path_curr_dir = os.getcwd()
14
15     # =====
16     # initial masks and training mode
17     # =====
18
19
20     # boolean flags to specify whether we are learning the masks or not
21     learn_mask = [False, False]
22
23
24     # initial masks for learning optimized masks or final masks for training a localization model
25     # if learn_mask=True the initial mask is initialized by default to be zero-modulation
26     path_masks = path_curr_dir + '/Mat_Files/masks_oracle_vipr.mat'
27     mask_dict = sio.loadmat(path_masks)
28     mask_init = [mask_dict['mask1'], mask_dict['mask2']]
29
30     # mask options dictionary
31     mask_opts = {'learn_mask': learn_mask, 'mask_init': mask_init}
32
33     # =====
34     # optics settings: objective, light, sensor properties
35     # =====
36
37     lamda = 0.605 # mean emission wavelength # in [um] (1e-6*meter)
38     NA = 1.49 # numerical aperture of the objective lens
39     noil = 1.518 # immersion medium refractive index
40     nwater = 1.33 # imaging medium refractive index
41     pixel_size_CCD = 11 # sensor pixel size in [um] (including binning)
42     pixel_size_SLM = 24 # SLM pixel size in [um] (after binning of 3 to reduce computational complexity)
43     M = 100 # optical magnification
44     f_4f = 15e4 # 4f lenses focal length in [um]
45
46     # optical settings dictionary
47     optics_dict = {'lamda': lamda, 'NA': NA, 'noil': noil, 'nwater': nwater, 'pixel_size_CCD': pixel_size_CCD,
48                   'pixel_size_SLM': pixel_size_SLM, 'M': M, 'f_4f': f_4f}
```

Figure. 5 –Training mode, initial mask, and optical setup parameters.

- Third section is used to specify the transforms mapping from camera 1 to camera 2, as well as intrinsic shifts due to the application of a phase mask, compared to the unmodulated system (**Fig. 6**). Normally, the transforms from the unmodulated system to the system with a mask are already taken care of by VIPR in phase retrieval, in which case they should be set to an identity matrix (i.e. (homogeneous) coordinates do not change). The transforms include: transform from no mask to a phase mask in channel 1 (Tpsf1), transform from no mask to a phase mask in channel 2 (Tpsf2), transform from phase mask in channel 1 to a phase mask in channel 2 in pixels with an origin in the top left corner of the FOV (Tpsf1to2_px), and transform from phase mask in channel 1 to a phase mask in channel 2 in microns with an origin in the center of the FOV (Tpsf1to2_um).



```

parameter_setting_oracle.py 3 x
Demos > parameter_setting_oracle.py > ...
11 def setup_parameters():
48
49 # =====
50 # sensor matching and affine transforms
51 # =====
52
53 # transforms from no mask to a mask in each camera and between cameras
54 # a transform set to identity means no change in (homogeneous) coordinates
55 Tpsf1 = np.eye(3, 3) # transform from no mask to mask 1 (for example shift in xy)
56 Tpsf2 = np.eye(3, 3) # transform from no mask to mask 2 (for example shift in xy)
57 Tpsf1to2_px = np.eye(3, 3) # transform from mask 1 to mask 2 in px (origin = top left corner of FOV)
58 Tpsf1to2_um = np.eye(3, 3) # transform from mask 1 to mask 2 in um (origin = center of FOV)
59
60 # global shift to cover the affine range
61 FOV_shift_range = [-5.0, 5.0] # in [um]
62
63 # registration error between the two cameras applied to the EDOF channel
64 reg_err = 0.0 # in [um] 0.05
65
66 # sensor matching dictionary
67 tform_dict = {'T1': Tpsf1, 'T2': Tpsf2, 'T12_px': Tpsf1to2_px, 'T12_um': Tpsf1to2_um,
68              'FOV_shift_range': FOV_shift_range, 'reg_err': reg_err}
69

```

Figure. 6 – Channel registration.

- Fourth section is used to control the dimensions of the Fourier space (**Fig. 7** blue rectangle), the image space (**Fig. 7** green rectangle), and the discretization in the z axis (**Fig. 7** red rectangle). The z-values are set like expected in the experimental sample. If the imaged emitters are directly on the coverslip, then “zmin” = 0. Otherwise, the distance of the lowest emitter from the coverslip needs to be calibrated for accurate depth recovery. The nominal focus position “NFP” should be set such that the PSF approximately spans the entire axial range. The number of voxels in z (“D”) is what sets the axial discretization. Smaller voxels require more training time but will lead to more accurate depth recovery. Normally, for an axial range of 4 microns this is set to be within the range [81,121]. The resulting voxel size in z is $\Delta_z = \frac{z_{max}-z_{min}}{D} [\mu m]$.
- Fifth section is used to control the training density. The parameter “num_particles_range” defines the upper and lower limit on the number of particles that will be simulated within the field of view (FOV) (**Fig. 7** yellow rectangle). Normally, a large range during training makes the model more robust to density variations in the experimental data.

```

11 def setup_parameters():
69
70 # =====
71 # phase mask and image space dimensions for simulation
72 # =====
73
74 # phase mask dimensions
75 Hmask, Wmask = 343, 343 # in SLM [pixels]
76
77 # single training image dimensions
78 H, W = 121, 121 # in sensor [pixels]
79
80 # safety margin from the boundary to prevent PSF truncation
81 clear_dist = 20 # in sensor [pixels]
82
83 # training z-range and focus
84 zmin = 1 # minimal z in [um] (including the axial shift)
85 zmax = 4.5 # maximal z in [um] (including the axial shift)
86 NFP = 3.15 # nominal focal plane in [um] (including the axial shift)
87
88 # discretization in z
89 D = 81 # in [voxels] spanning the axial range (zmax - zmin)
90
91 # data dimensions dictionary
92 data_dims_dict = {'Hmask': Hmask, 'Wmask': Wmask, 'H': H, 'W': W, 'clear_dist': clear_dist, 'zmin': zmin,
93                  'zmax': zmax, 'NFP': NFP, 'D': D}
94
95 # =====
96 # number of emitters in each FOV
97 # =====
98
99 # upper and lower limits for the number of emitters
100 num_particles_range = [1, 50]
101
102 # number of particles dictionary
103 num_particles_dict = {'num_particles_range': num_particles_range}
104

```

Figure. 7 – Dimensions, discretization in z, and emitter density.

6. Sixth section of the parameters is used to control the distribution of the signal counts and threshold on the minimal detectable signal for the given phase mask (**Fig. 8** blue rectangle). The Boolean flag “nsig_unif” specifies whether the counts distribution for different emitters is uniform or gamma distributed. In case it is True, then the user needs to define the upper and lower limit of the uniform distribution in the parameter “nsig_unif_range”. Otherwise, the counts distribution will be Gamma with a shape and scale parameters as defined in “nsig_gamma_params”. For gamma distributed counts, the user can set a threshold on the detectable number of counts given the SNR conditions. This is controlled by the parameter “nsig_thresh”. Finally, to account for intensity scaling between imaging channels the parameter “nsig_ratio_range” specifies the upper and lower limits for the intensity scaling factor in channel 2, where we assume channel 2 have a reduced signal efficiency compared to channel 1.
7. Seventh section of the parameters control the range of the standard deviation of the Gaussian blur used to smooth the simulated PSFs (**Fig. 8** green rectangle). For true point sources, the parameter “blur_std_range” can be set to the estimated parameter in VIPR. Otherwise, for finite size emitters (like in the telomere sample), this parameter is set to the range [0.75, 1.0] to account for variable emitter size.

```

11 def setup_parameters():
105     # ===== You, 4 days ago *
106     # signal counts distribution and settings
107     # =====
108
109     # boolean that specifies whether the signal counts are uniformly distributed
110     nsig_unif = True
111
112     # range of signal counts assuming a uniform distribution
113     nsig_unif_range = [20000, 120000] # in [counts]
114
115     # parameters for sampling signal counts assuming a gamma distribution
116     nsig_gamma_params = None # in [counts]
117
118     # threshold on signal counts to discard positions from the training labels
119     nsig_thresh = None # in [counts]
120
121     # range to randomize the fraction of counts in the second sensor (if Nc=2)
122     nsig_ratio_range = [0.75, 0.95] # in [counts]
123
124     # signal counts dictionary
125     nsig_dict = {'nsig_unif': nsig_unif, 'nsig_unif_range': nsig_unif_range, 'nsig_gamma_params': nsig_gamma_params,
126                 'nsig_thresh': nsig_thresh, 'nsig_ratio_range': nsig_ratio_range}
127
128     # =====
129     # blur standard deviation for smoothing PSFs to match experimental conditions
130     # =====
131
132     # upper and lower blur standard deviation for each emitter to account for finite size
133     blur_std_range = [0.75, 1] # in sensor [pixels]
134
135     # blur dictionary
136     blur_dict = {'blur_std_range': blur_std_range}

```

Figure 8 – Signal distribution and blur standard deviation.

8. Eight section of the parameters control the Poisson background settings, namely, uniform/non-uniform (**Fig. 9** blue rectangle). The parameter “unif_bg” specifies the uniform background component in counts. The Boolean flag “nonunif_bg_flag” specifies whether to include the non-uniform background modelled by a super-Gaussian. The super-Gaussian center is randomly shifted within the range specified by “nonunif_bg_offset”, and the peak and valley of the super-Gaussian are given in “non_unif_minvals”. These values are perturbed by 50% randomly during training data generation. Finally, the rotation angle of the super-Gaussian is randomly chosen within the range given in “nonunif_bg_theta_range”. Note that even if the flag “nonunif_bg_flag” is set to False, the remaining values are required as these are also potentially used in defining the read noise spatial distribution discussed next.
9. Ninth section of the parameters control the read noise settings (**Fig. 9** green rectangle). The Boolean flag “read_noise_flag” is used to decide whether to add read noise. The Boolean flag “read_noise_nonunif” is used to decide whether the spatial distribution of the read noise is non-uniform across the FOV (higher in the middle). If set to True, the standard deviation of the read-noise across the FOV is assumed to be distributed using the super-Gaussian from section 8. The upper and lower limits for the read noise standard deviation are given in the parameter “read_noise_std_range”. Finally, after subtracting the minimal frame from the experimental data, normally we are left with a varying “baseline” across the FOV. To account for this, during data generation we add a random baseline in the range “read_noise_baseline_range”.

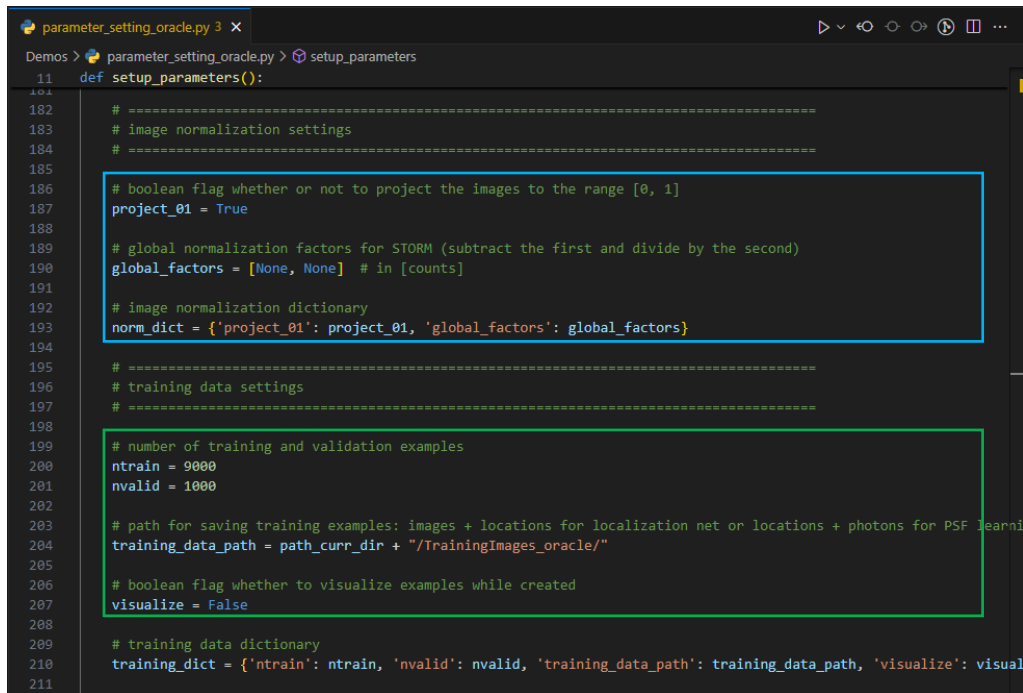
```

11 def setup_parameters():
12     # =====
13     # uniform/non-uniform background settings
14     # =====
15
16     # uniform background value per pixel
17     unif_bg = 0 # in [counts]
18
19     # boolean flag whether or not to include a non-uniform background
20     nonunif_bg_flag = True
21
22     # maximal offset for the center of the non-uniform background in pixels
23     nonunif_bg_offset = [10, 10] # in sensor [pixels]
24
25     # peak and valley minimal values for the super-gaussian; randomized with addition of up to 50%
26     nonunif_bg_minvals = [20.0, 150.0] # in [counts]
27
28     # minimal and maximal angle of the super-gaussian for augmentation
29     nonunif_bg_theta_range = [-pi/4, pi/4] # in [radians]
30
31     # nonuniform background dictionary
32     nonunif_bg_dict = {'nonunif_bg_flag': nonunif_bg_flag, 'unif_bg': unif_bg, 'nonunif_bg_offset': nonunif_bg_offset,
33                       'nonunif_bg_minvals': nonunif_bg_minvals, 'nonunif_bg_theta_range': nonunif_bg_theta_range}
34
35     # =====
36     # read noise settings
37     # =====
38
39     # boolean flag whether or not to include read noise
40     read_noise_flag = True
41
42     # flag whether or not the read noise standard deviation is not uniform across the FOV
43     read_noise_nonunif = False
44
45     # range of baseline of the min-subtracted data in STORM
46     read_noise_baseline_range = [200.0, 250.0] # in [counts]
47
48     # read noise standard deviation upper and lower range
49     read_noise_std_range = [20.0, 24.0] # in [counts]
50
51     # read noise dictionary
52     read_noise_dict = {'read_noise_flag': read_noise_flag, 'read_noise_nonunif': read_noise_nonunif,
53                       'read_noise_baseline_range': read_noise_baseline_range,
54                       'read_noise_std_range': read_noise_std_range}
55

```

Figure 9 – Non-uniform background and read noise.

10. Tenth section of the parameters control the image normalization for training (**Fig. 10** blue rectangle). The Boolean flag “project_01” specifies whether each training image should be normalized to the range [0,1] using the transformation: $I_{[0,1]} = \frac{I - I_{min}}{I_{max} - I_{min}}$. This is helpful especially when the SNR is changing from frame to frame (e.g. Fig. 10 in the main text). If “project_01” is set to False, the parameter “global_factors” is used to normalize the training images via the transformation: $I_{norm} = \frac{I - global_factors[1]}{global_factors[2]}$. This is normally useful when the SNR doesn’t change much in between frames.
11. Eleventh section of the parameters control the size of the training and validation sets (**Fig. 10** green rectangle). “ntrain” defines the number of examples for training, and “nvalid” defines the number of examples for validation. The parameter “training_data_path” defines the path where the generated training examples will be saved. The Boolean flag “visualize” controls whether the images are shown during data generation.



```
parameter_setting_oracle.py 3 x
Demos > parameter_setting_oracle.py > setup_parameters
11 def setup_parameters():
12     # =====
13     # image normalization settings
14     # =====
15
16     # boolean flag whether or not to project the images to the range [0, 1]
17     project_01 = True
18
19     # global normalization factors for STORM (subtract the first and divide by the second)
20     global_factors = [None, None] # in [counts]
21
22     # image normalization dictionary
23     norm_dict = {'project_01': project_01, 'global_factors': global_factors}
24
25     # =====
26     # training data settings
27     # =====
28
29     # number of training and validation examples
30     ntrain = 9000
31     nvalid = 1000
32
33     # path for saving training examples: images + locations for localization net or locations + photons for PSF learning
34     training_data_path = path_curr_dir + "/TrainingImages_oracle/"
35
36     # boolean flag whether to visualize examples while created
37     visualize = False
38
39     # training data dictionary
40     training_dict = {'ntrain': ntrain, 'nvalid': nvalid, 'training_data_path': training_data_path, 'visualize': visualize}
```

Figure. 10 – Image normalization and training/validation size.

12. Twelfth section of the parameters control the learning settings (**Fig. 11** blue rectangle). The parameter “results_path” defines the path where the learning results (including the CNN weights) will be saved. The Boolean flag “dilation_flag” controls whether the maximal dilation will be $d_{max} = 16$ or $d_{max} = 4$. This controls the network receptive field as explained in Fig. S2 in the supplementary information. The parameter “batch_size” controls the batch size used for training the CNN. Normally this is set to the biggest number that can still fit the entire model on GPU for training (in this case 4). The parameter “max_epochs” defines the maximal number of epochs for training the CNN, and the parameter “initial_learning_rate” defines the initial learning rate for the ADAM optimizer. Finally, the parameter “scaling_factor” is used to control the scaling factor of the emitters in the loss function. This parameter is used to mitigate the class imbalance between empty voxels and voxels containing emitters in the prediction grid.
13. Thirteenth section allows the user to resume training a model from a previously saved checkpoint in case of system crash/ Transfer learning (**Fig. 11** green rectangle). This feature is advanced and not recommended for new users.
14. Last section lets the user decide which GPU to use, assuming you have multiple GPUs on your system. Otherwise, if there is only one GPU (cuda:0) or only CPU, this will be set for you automatically if you do not change the existing setting (**Fig. 12**)

```
parameter_setting_oracle.py 3
Demos > parameter_setting_oracle.py > setup_parameters
11 def setup_parameters():
212 # =====
213 # learning settings
214 # =====
215
216 # results folder to save the trained model
217 results_path = path_curr_dir + "/Results_oracle/"
218
219 # maximal dilation flag when learning a localization CNN (set to None if learn_mask=True as we use a different CNN
220 dilation_flag = True # if set to 1 then dmax=16 otherwise dmax=4
221
222 # batch size for training a localization model (set to 1 for mask learning as examples are generated 16 at a time)
223 batch_size = 4
224
225 # maximal number of epochs
226 max_epochs = 50
227
228 # initial learning rate for adam
229 initial_learning_rate = 0.0005
230
231 # scaling factor for the loss function
232 scaling_factor = 800.0
233
234 # learning dictionary
235 learning_dict = {'results_path': results_path, 'dilation_flag': dilation_flag, 'batch_size': batch_size,
236                 'max_epochs': max_epochs, 'initial_learning_rate': initial_learning_rate,
237                 'scaling_factor': scaling_factor}
238
239 # =====
240 # resuming from checkpoint settings
241 # =====
242
243 # boolean flag whether to resume training from checkpoint
244 resume_training = False
245
246 # number of epochs to resume training
247 num_epochs_resume = None
248
249 # saved checkpoint to resume from
250 checkpoint_path = None
251
252 # checkpoint dictionary
253 checkpoint_dict = {'resume_training': resume_training, 'num_epochs_resume': num_epochs_resume,
254                   'checkpoint_path': checkpoint_path}
255
```

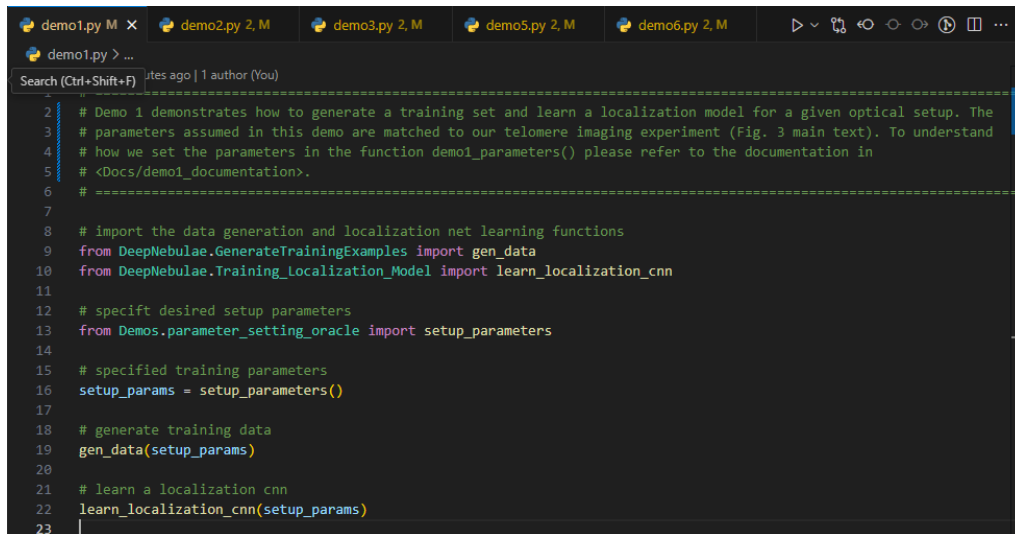
Figure. 11 – Learning settings and resuming from checkpoint.

```
parameter_setting_oracle.py 3
Demos > parameter_setting_oracle.py > setup_parameters
11 def setup_parameters():
256 # =====
257 # device to use for training/validation (optimally should be a cuda device)
258 # =====
259
260 # device to train/evaluate on
261 device_id = 0
262 device = torch.device("cuda:" + str(device_id) if torch.cuda.is_available() else "cpu")
263
264 # device dictionary
265 device_dict = {'device': device}
266
267 # =====
268 # final resulting dictionary including all parameters
269 # =====
270
271 settings = {'**mask_opts', **num_particles_dict, **nsig_dict, **blur_dict, **nonunif_bg_dict, **read_noise_dict,
272            '**norm_dict, **optics_dict, **data_dims_dict, **training_dict, **learning_dict, **checkpoint_dict,
273            '**tform_dict, **device_dict}
274
275 return settings
276
```

Figure. 12 – GPU id for training.

Training a localization CNN

Now, after setting the parameters for data generation and network training the user needs to generate the training data and learn a localization CNN. This is achieved by using the function “gen_data” from the module “GenerateTrainingExamples.py”, and the function “learn_localization_cnn” from the module “Training_Localization_Model”. Both steps are demonstrated in the script “demo1.py” (**Fig. 13**). Note that it takes approximately 35 hours on a Titan Xp GPU to learn a localization CNN with the specified parameters in “parameter_setting_oracle.py”.



```
demo1.py M x demo2.py 2, M demo3.py 2, M demo5.py 2, M demo6.py 2, M
demo1.py > ...
Search (Ctrl+Shift+F) 1tes ago | 1 author (You)
2 # Demo 1 demonstrates how to generate a training set and learn a localization model for a given optical setup. The
3 # parameters assumed in this demo are matched to our telomere imaging experiment (Fig. 3 main text). To understand
4 # how we set the parameters in the function demo1_parameters() please refer to the documentation in
5 # <Docs/demo1_documentation>.
6 # -----
7
8 # import the data generation and localization net learning functions
9 from DeepNebulae.GenerateTrainingExamples import gen_data
10 from DeepNebulae.Training_Localization_Model import learn_localization_cnn
11
12 # specifit desired setup parameters
13 from Demos.parameter_setting_oracle import setup_parameters
14
15 # specified training parameters
16 setup_params = setup_parameters()
17
18 # generate training data
19 gen_data(setup_params)
20
21 # learn a localization cnn
22 learn_localization_cnn(setup_params)
23
```

Figure. 13 – demo1.

Localization using a trained CNN

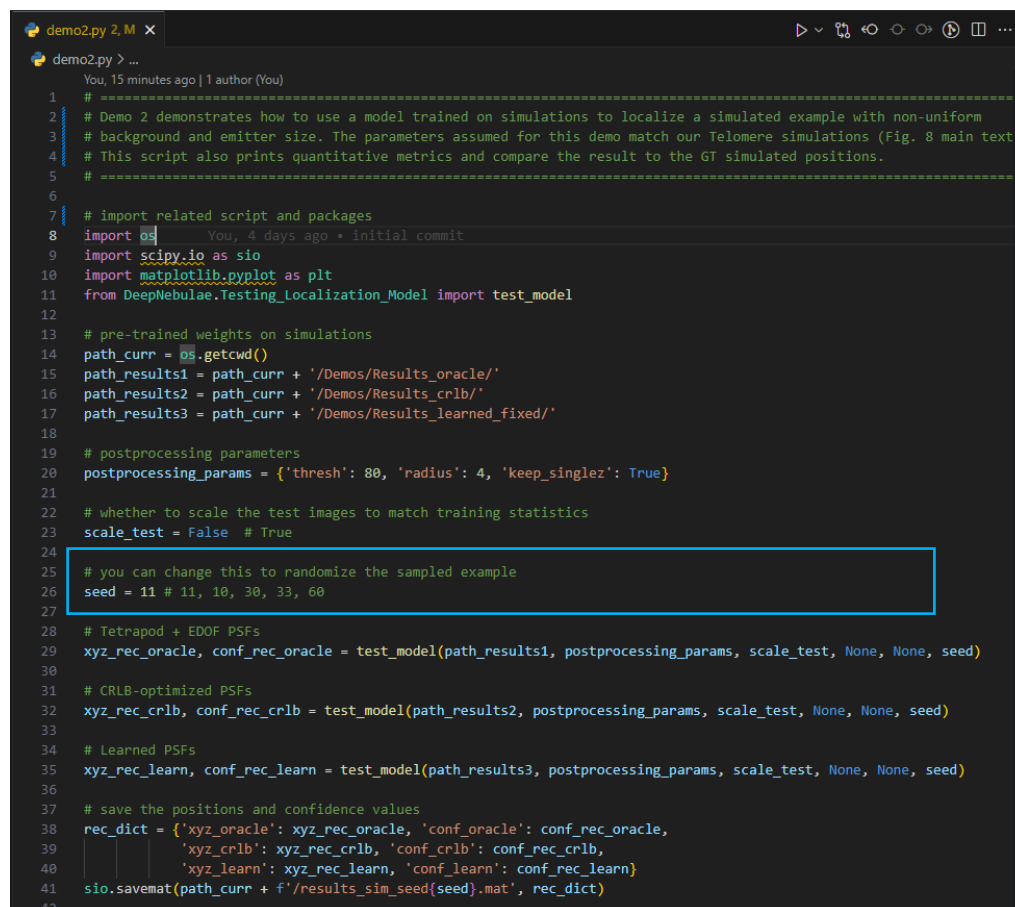
After learning a localization CNN, the model can be tested using the function “test_model” from the module “Testing_Localization_Model.py” as demonstrated in “demo2.py” (**Fig. 14**). The function “test_model” accepts 6 input parameters:

1. “path_results” which is the path to the saved training results.
2. “postprocessing_params” which is a dictionary with 3 entries: “thresh”, “radius”, and “keep_onez” specifying the postprocessing threshold (in the range [0, “scaling_factor”]), the radius for local maxima finding in recovery voxels, and whether more than one recovery is allowed in the same recovery sub-pixel at different zs. These 3 parameters ultimately control the Jaccard Index and the RMSE of the network. Usually “radius”=4, “thresh” in the range [0.05, 0.2]*“scaling_factor”, and “keep_onez=True” are a good guess for most scenarios.
3. “Scale_test” which is a Boolean flag specifying whether to renormalize test data to have the same mean and standard deviation as training data prior to CNN processing. In most cases, this flag is better kept “False”.
4. “path_exp_data1” which is the path to the first channel experimental data for testing the model. In case the folder only contains a single experimental frame (as in “demo3.py”), the function will plot the recovered 3D positions and the regenerated input image for visual comparison with the experimental images. Otherwise, if the folder contains several frames (such as in “demo5.py”) the function will save a csv file in the experimental folder named

“localizations_<data>_<time>.csv”. This file will have 5 columns (“frame”, “x [nm]”, “y [nm]”, “z [nm]”, and “intensity [au]”), and can be used afterwards for drift correction and visualization in ThunderSTORM [4] or ZOLA-3D [5]. If the number of images in the folder is below 100, the script will also plot the images with localizations on top marked by red crosses.

Finally, if this parameter is set to “None” the function will generate a random testing image and show the recovered 3D positions alongside the GT positions and calculate quantitative performance metrics. These include the Jaccard Index, the lateral RMSE and the axial RMSE. In addition, the input image will be compared to the regenerated image visually.

5. “path_exp_data2” which is the path to the second channel experimental data for testing the model. The number of images in this folder must match the number of images in the folder “path_exp_data1” and have matching names!
6. “seed” is the seed used to generate the testing image in case no experimental data is supplied. You can play around with this parameter to randomize the testing image in demo 2/6 (**Fig. 14**, blue rectangle).



```

demo2.py 2, M x
demo2.py > ...
You, 15 minutes ago | 1 author (You)
1 # =====
2 # Demo 2 demonstrates how to use a model trained on simulations to localize a simulated example with non-uniform
3 # background and emitter size. The parameters assumed for this demo match our Telomere simulations (Fig. 8 main text).
4 # This script also prints quantitative metrics and compare the result to the GT simulated positions.
5 # =====
6
7 # import related script and packages
8 import os
9 import scipy.io as sio
10 import matplotlib.pyplot as plt
11 from DeepNebulae.Testing_Localization_Model import test_model
12
13 # pre-trained weights on simulations
14 path_curr = os.getcwd()
15 path_results1 = path_curr + '/Demos/Results_oracle/'
16 path_results2 = path_curr + '/Demos/Results_crlb/'
17 path_results3 = path_curr + '/Demos/Results_learned_fixed/'
18
19 # postprocessing parameters
20 postprocessing_params = {'thresh': 80, 'radius': 4, 'keep_single': True}
21
22 # whether to scale the test images to match training statistics
23 scale_test = False # True
24
25 # you can change this to randomize the sampled example
26 seed = 11 # 11, 10, 30, 33, 60
27
28 # Tetrapod + EDOF PSFs
29 xyz_rec_oracle, conf_rec_oracle = test_model(path_results1, postprocessing_params, scale_test, None, None, seed)
30
31 # CRLB-optimized PSFs
32 xyz_rec_crlb, conf_rec_crlb = test_model(path_results2, postprocessing_params, scale_test, None, None, seed)
33
34 # Learned PSFs
35 xyz_rec_learn, conf_rec_learn = test_model(path_results3, postprocessing_params, scale_test, None, None, seed)
36
37 # save the positions and confidence values
38 rec_dict = {'xyz_oracle': xyz_rec_oracle, 'conf_oracle': conf_rec_oracle,
39            'xyz_crlb': xyz_rec_crlb, 'conf_crlb': conf_rec_crlb,
40            'xyz_learn': xyz_rec_learn, 'conf_learn': conf_rec_learn}
41 sio.savemat(path_curr + f'/results_sim_seed{seed}.mat', rec_dict)
42

```

Figure. 14 – demo2.

References

- [1] B. Ferdman, E. Nehme, L. E. Weiss, R. Orange, O. Alalouf, and Y. Shechtman, "VIPR: Vectorial Implementation of Phase Retrieval for fast and accurate microscopic pixel-wise pupil estimation," *bioRxiv*, 2020.
- [2] J. Schindelin, C. T. Rueden, M. C. Hiner, and K. W. Eliceiri, "The ImageJ ecosystem: An open platform for biomedical image analysis," *Mol. Reprod. Dev.*, vol. 82, no. 7–8, pp. 518–529, 2015.
- [3] J. Schindelin *et al.*, "Fiji: An open-source platform for biological-image analysis," *Nat. Methods*, vol. 9, no. 7, pp. 676–682, 2012.
- [4] M. Ovesný, P. Křížek, J. Borkovec, Z. Švindrych, and G. M. Hagen, "ThunderSTORM: A comprehensive ImageJ plug-in for PALM and STORM data analysis and super-resolution imaging," *Bioinformatics*, vol. 30, no. 16, pp. 2389–2390, 2014.
- [5] A. Aristov, B. Lelandais, E. Rensen, and C. Zimmer, "ZOLA-3D allows flexible 3D localization microscopy over an adjustable axial range," *Nat. Commun.*, vol. 9, no. 1, pp. 1–8, 2018.