Deep-STORM3D Documentation Release 1.0.0

March 10, 2020

Contents

Demo 1 – Learning a localization model

This demo is intended to get you started using Deep-STORM3D. Given a new optical setup, new phase mask, 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 mask

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

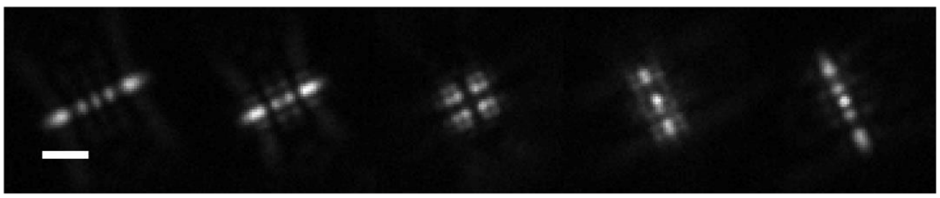


Figure. 1 – 5 slices taken from a z-stack of a fluorescent microsphere that was used to calibrate the PSF in the STORM experiment (main text Fig. 3). Scale bar is 2 microns.

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

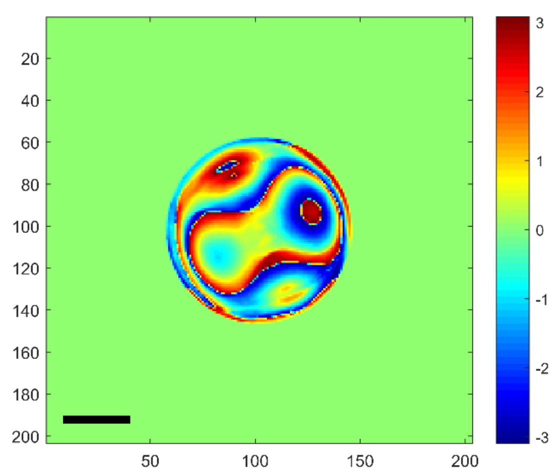


Figure. 2 - phase mask retrieved by VIPR[1]. SLM pixel size is 30 microns. Scale bar is 1 mm.

[1] https://github.com/Borisfer/VIPR---Vectorial-Phase-Retrieval-for-microscopy

Experimental stack preprocessing and SNR estimation

Usually in STORM experiments the acquired frames have a non-uniform background component stemming from autofluorescence. The simplest way to partially eliminate this non-uniform background is to subtract the minimal value per pixel from the entire stack (**Fig. 3**).

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 remaining baseline and the background standard deviation we can examine emitter-empty regions in the experimental frames (**Fig. 3b** red square). Similarly, the signal intensity can be estimated from subtracting the mean background counts in emitter-occupied regions (**Fig. 3b** green square).

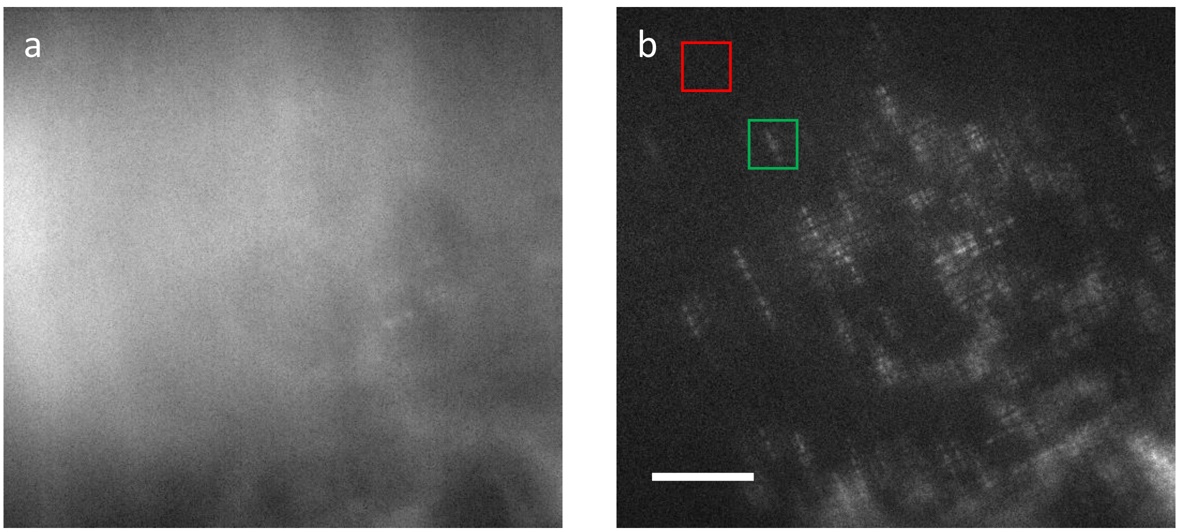


Figure. 3 - background subtraction in Fiji [2,3]. a) minimum frame of the experimental stack. b) Experimental frames after minimum subtraction. Scale bar is 10 microns.

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_demo1.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 a phase mask or learning a localization CNN. In case we are learning a localization CNN, the Boolean flag “learn_mask” (**Fig. 4** blue rectangle) is set to False, and the “initial” mask is set to the mask recovered by VIPR (**Fig. 4** green rectangle).
2. Second section is used to specify the optical system and the imaging assumptions (**Fig. 4** 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

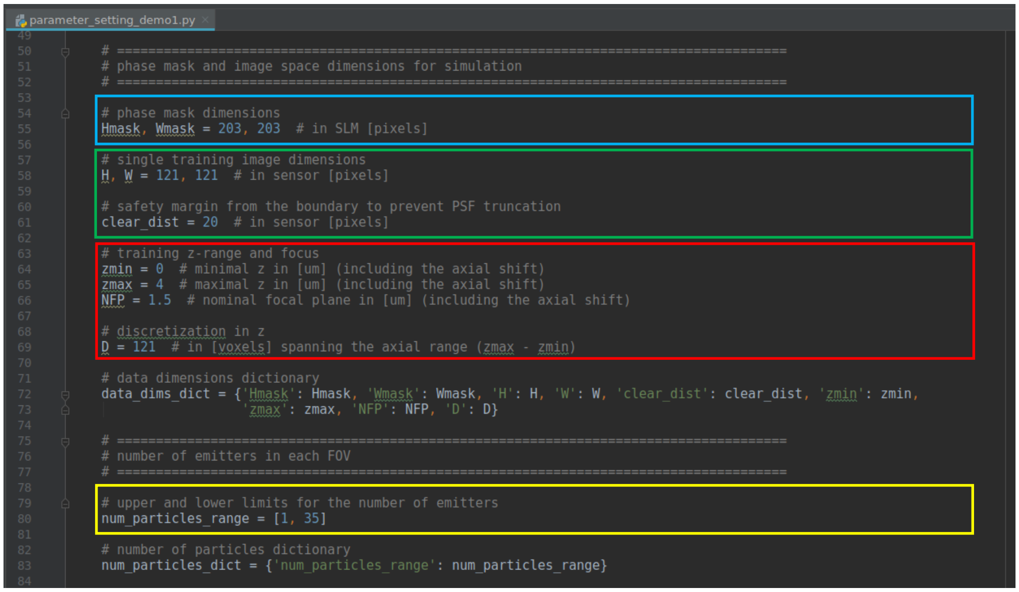
of the spatial light modulator in microns (pixel_size_SLM), the optical magnification (M), and the 4f lenses focal length in microns (f_4f).

parameter_setting_demo1.py

```python
# import needed libraries
from math import pi
import scipy.io as sio
import os


def demo1_parameters():

    # path to current directory
    path_curr_dir = os.getcwd()

    # ===================================================================================
    # initial mask and training mode
    # ===================================================================================

    # boolean that specifies whether we are learning a mask or not
    learn_mask = False

    # initial mask for learning an optimized mask or final mask for training a localization model
    # if learn_mask=True the initial mask is initialized by default to be zero-modulation
    path_mask = path_curr_dir + '/Mat_Files/mask_tetrapod_printed.mat'
    mask_dict = sio.loadmat(path_mask)
    mask_name = list(mask_dict.keys())[3]
    mask_init = mask_dict[mask_name]

    # mask options dictionary
    mask_opts = {'learn_mask': learn_mask, 'mask_init': mask_init}

    # ===================================================================================
    # optics settings: objective, light, sensor properties
    # ===================================================================================

    lamda = 0.67  # mean emission wavelength # in [um] (1e-6*meter)
    NA = 1.45  # numerical aperture of the objective lens
    noil = 1.518  # immersion medium refractive index
    nwater = 1.33  # imaging medium refractive index
    pixel_size_CCD = 11  # sensor pixel size in [um]
    pixel_size_SLM = 30  # SLM pixel size in [um] (after binning of 3 to reduce computational complexity)
    M = 100  # optical magnification
    f_4f = 10e4  # 4f lenses focal length in [um]

    # optical settings dictionary
    optics_dict = {'lamda': lamda, 'NA': NA, 'noil': noil, 'nwater': nwater, 'pixel_size_CCD': pixel_size_CCD,
                   'pixel_size_SLM': pixel_size_SLM, 'M': M, 'f_4f': f_4f}
```

Figure. 4 –Training mode, initial mask, and optical setup parameters.

3. Third section is used to control the dimensions of the Fourier space (**Fig. 5** blue rectangle), the image space (**Fig. 5** green rectangle), and the discretization in the z axis (**Fig. 5** 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 $\Delta_z = \frac{z_{max}-z_{min}}{D}\ [\mu m]$.
4. Fourth 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. 5** yellow rectangle). Normally, a large range during training makes the model more robust to density variations in the experimental data.

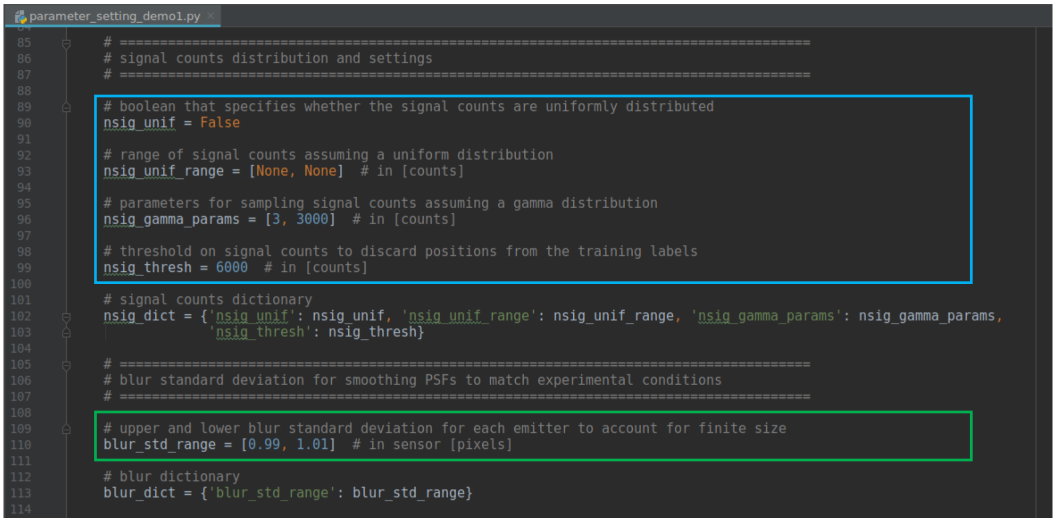


parameter_setting_demo1.py

```python
# ====================================================================================
# phase mask and image space dimensions for simulation
# ====================================================================================

# phase mask dimensions
Hmask, Wmask = 203, 203  # in SLM [pixels]

# single training image dimensions
H, W = 121, 121  # in sensor [pixels]

# safety margin from the boundary to prevent PSF truncation
clear_dist = 20  # in sensor [pixels]

# training z-range and focus
zmin = 0  # minimal z in [um] (including the axial shift)
zmax = 4  # maximal z in [um] (including the axial shift)
NFP = 1.5  # nominal focal plane in [um] (including the axial shift)

# discretization in z
D = 121  # in [voxels] spanning the axial range (zmax - zmin)

# data dimensions dictionary
data_dims_dict = {'Hmask': Hmask, 'Wmask': Wmask, 'H': H, 'W': W, 'clear_dist': clear_dist, 'zmin': zmin,
                  'zmax': zmax, 'NFP': NFP, 'D': D}

# ====================================================================================
# number of emitters in each FOV
# ====================================================================================

# upper and lower limits for the number of emitters
num_particles_range = [1, 35]

# number of particles dictionary
num_particles_dict = {'num_particles_range': num_particles_range}
```

Figure. 5 – Dimensions, discretization in z, and emitter density.

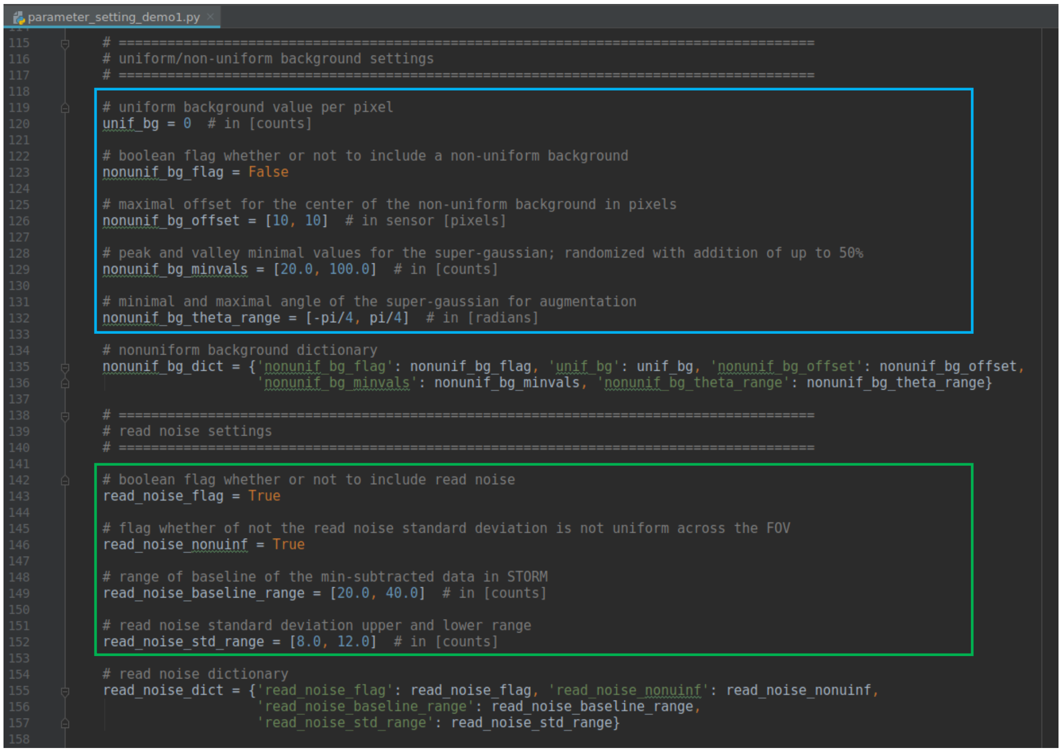
5. Fifth section of the parameters is used to control the distribution of the signal counts and threshold on the minimal detectable signal for the given phase mask (**Fig. 6** blue rectangle). The Boolean flag “nsig_unif” specifies whether the counts distribution for different emitters is uniform or 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”. Finally, 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”.
6. Sixth section of the parameters control the range of the standard deviation of the Gaussian blur used to smooth the simulated PSFs (**Fig. 6** 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, 1.25] to account for variable emitter size.



```python
# ==========================================================================================
# signal counts distribution and settings
# ==========================================================================================

# boolean that specifies whether the signal counts are uniformly distributed
nsig_unif = False

# range of signal counts assuming a uniform distribution
nsig_unif_range = [None, None]  # in [counts]

# parameters for sampling signal counts assuming a gamma distribution
nsig_gamma_params = [3, 3000]  # in [counts]

# threshold on signal counts to discard positions from the training labels
nsig_thresh = 6000  # in [counts]

# signal counts dictionary
nsig_dict = {'nsig_unif': nsig_unif, 'nsig_unif_range': nsig_unif_range, 'nsig_gamma_params': nsig_gamma_params,
             'nsig_thresh': nsig_thresh}

# ==========================================================================================
# blur standard deviation for smoothing PSFs to match experimental conditions
# ==========================================================================================

# upper and lower blur standard deviation for each emitter to account for finite size
blur_std_range = [0.99, 1.01]  # in sensor [pixels]

# blur dictionary
blur_dict = {'blur_std_range': blur_std_range}
```

Figure 6 – Signal distribution and blur standard deviation.

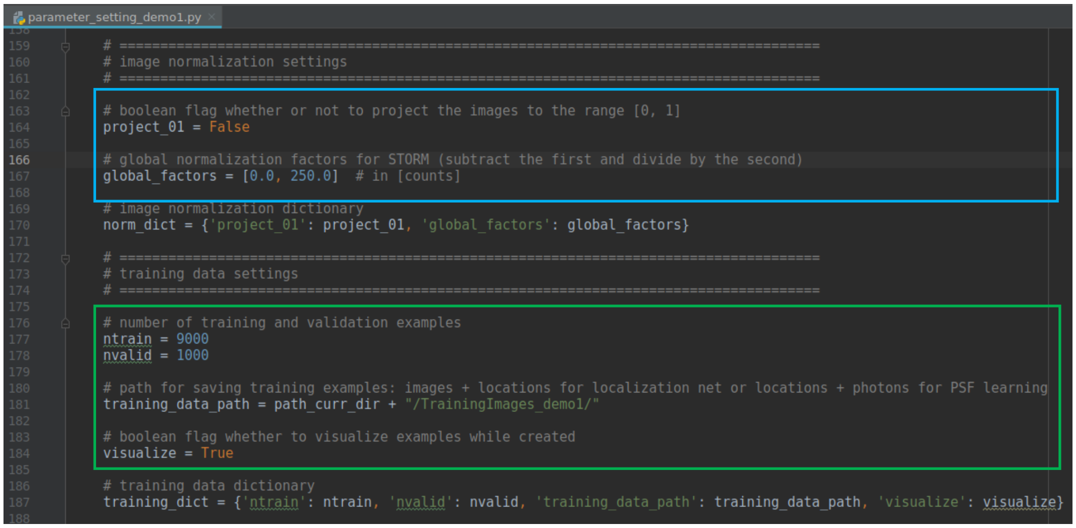
7. Seventh section of the parameters control the Poisson background settings, namely, uniform/non-uniform (**Fig. 7** blue rectangle). The parameter “unif_bg” specifies the uniform background component in counts. The Boolean flag “nonunif_bg_flag” specifies whether to include the non-uniform background modelled by a super-Gaussian. The supper-Gaussian center is randomly shifted within the range specified by “nonunif_bg_offset”, and the peak and valey 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.
8. Eight section of the parameters control the read noise settings (**Fig. 7** 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 7. 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”.



```python
# ====================================================================================
# uniform/non-uniform background settings
# ====================================================================================

# uniform background value per pixel
unif_bg = 0  # in [counts]

# boolean flag whether or not to include a non-uniform background
nonunif_bg_flag = False

# maximal offset for the center of the non-uniform background in pixels
nonunif_bg_offset = [10, 10]  # in sensor [pixels]

# peak and valley minimal values for the super-gaussian; randomized with addition of up to 50%
nonunif_bg_minvals = [20.0, 100.0]  # in [counts]

# minimal and maximal angle of the super-gaussian for augmentation
nonunif_bg_theta_range = [-pi/4, pi/4]  # in [radians]

# nonuniform background dictionary
nonunif_bg_dict = {'nonunif_bg_flag': nonunif_bg_flag, 'unif_bg': unif_bg, 'nonunif_bg_offset': nonunif_bg_offset,
                   'nonunif_bg_minvals': nonunif_bg_minvals, 'nonunif_bg_theta_range': nonunif_bg_theta_range}

# ====================================================================================
# read noise settings
# ====================================================================================

# boolean flag whether or not to include read noise
read_noise_flag = True

# flag whether of not the read noise standard deviation is not uniform across the FOV
read_noise_nonuinf = True

# range of baseline of the min-subtracted data in STORM
read_noise_baseline_range = [20.0, 40.0]  # in [counts]

# read noise standard deviation upper and lower range
read_noise_std_range = [8.0, 12.0]  # in [counts]

# read noise dictionary
read_noise_dict = {'read_noise_flag': read_noise_flag, 'read_noise_nonuinf': read_noise_nonuinf,
                   'read_noise_baseline_range': read_noise_baseline_range,
                   'read_noise_std_range': read_noise_std_range}
```

Figure 7 – Non-uniform background and read noise.

9. Nineth section of the parameters control the image normalization for training (**Fig. 8** blue rectangle). The Boolean flag “project_01” specifies whether each training image should be normalized to the range [0,1] using the transformation: $I_{[0,1]} = \frac{I-I_{min}}{I_{max}-I_{mn}}$. This is helpful especially when the SNR is changing from frame to frame (e.g. Fig. S32 in supplementary information, or Fig. 6 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.
10. Tenth section of the parameters control the size of the training and validation sets (**Fig. 8** 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_demo1.py

# ==========================================================================================
# image normalization settings
# ==========================================================================================

# boolean flag whether or not to project the images to the range [0, 1]
project_01 = False

# global normalization factors for STORM (subtract the first and divide by the second)
global_factors = [0.0, 250.0]  # in [counts]

# image normalization dictionary
norm_dict = {'project_01': project_01, 'global_factors': global_factors}

# ==========================================================================================
# training data settings
# ==========================================================================================

# number of training and validation examples
ntrain = 9000
nvalid = 1000

# path for saving training examples: images + locations for localization net or locations + photons for PSF learning
training_data_path = path_curr_dir + "/TrainingImages_demo1/"

# boolean flag whether to visualize examples while created
visualize = True

# training data dictionary
training_dict = {'ntrain': ntrain, 'nvalid': nvalid, 'training_data_path': training_data_path, 'visualize': visualize}
```

Figure. 8 – Image normalization and training/validation size.

11. Eleventh section of the parameters control the learning settings (**Fig. 9** blue rectangle). The parameter "results_path" defines the path were 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.
12. Last section allows the user to resume training a model from a previously saved checkpoint in case of system crash/ Transfer learning (**Fig. 9** green rectangle). This feature is advanced and not recommended for new users.

parameter_setting_demo1.py

```python
# ==================================================================================
# learning settings
# ==================================================================================

# results folder to save the trained model
results_path = path_curr_dir + "/Results_demo1/"

# maximal dilation flag when learning a localization CNN (set to None if learn_mask=True as we use a different CNN)
dilation_flag = True  # if set to 1 then dmax=16 otherwise dmax=4

# batch size for training a localization model (set to 1 for mask learning as examples are generated 16 at a time)
batch_size = 4

# maximal number of epochs
max_epochs = 50

# initial learning rate for adam
initial_learning_rate = 0.0005

# scaling factor for the loss function
scaling_factor = 800.0

# learning dictionary
learning_dict = {'results_path': results_path, 'dilation_flag': dilation_flag, 'batch_size': batch_size,
                 'max_epochs': max_epochs, 'initial_learning_rate': initial_learning_rate,
                 'scaling_factor': scaling_factor}

# ==================================================================================
# resuming from checkpoint settings
# ==================================================================================

# boolean flag whether to resume training from checkpoint
resume_training = False

# number of epochs to resume training
num_epochs_resume = None

# saved checkpoint to resume from
checkpoint_path = None

# checkpoint dictionary
checkpoint_dict = {'resume_training': resume_training, 'num_epochs_resume': num_epochs_resume,
                   'checkpoint_path': checkpoint_path}
```

Figure. 9 – Learning settings and resuming from checkpoint.

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. 10**). Note that it takes approximately 35 hours on a Titan Xp GPU to learn a localization CNN with the specified parameters in “Demos/parameter_setting_demo1.py”.

demo1.py

```python
# ======================================================================================================================
# Demo 1 demonstrates how to generate a training set and learn a localization model for a given optical setup. The
# parameters assumed in this demo are matched to our STORM experiment (Fig. 3 main text). To understand how we set the
# parameters in the function demo1_parameters() please refer to the documentation in <Docs/demo1_documentation>
# ======================================================================================================================

# import the data generation and localization net learning functions
from Demos.parameter_setting_demo1 import demo1_parameters
from DeepSTORM3D.GenerateTrainingExamples import gen_data
from DeepSTORM3D.Training_Localization_Model import learn_localization_cnn

# specified training parameters
setup_params = demo1_parameters()

# generate training data
gen_data(setup_params)

# learn a localization cnn
learn_localization_cnn(setup_params)
```

Figure. 10 – demo 1.

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. 11**). The function "test_model" accepts 4 input parameters:

1. "path_results" which is the path to the saved training results.
2. "postprocessing_params" which is a dictionary with 2 entries: "thresh" and "radius" specifying the postprocessing threshold (in the range [0, "scaling_factor"]) and the radius for local maxima finding in recovery voxels. These 2 parameters ultimately control the Jaccard Index and the RMSE of the network. Usually "radius"=4 and "thresh" in the range [0.05, 0.2]*"scaling_factor" are a good guess for most scenarios.
3. "path_exp_data" which is the path to the experimental data for testing the model. In case the folder only contains a single experimental frame (as in "demo5.py"), the function will plot the recovered 3D positions and the regenerated input image for visual comparison with the experimental image. Otherwise, if the folder contains several frames (such as in STORM experiments) the function will save a csv file in the experimental folder named "localizations.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.
4. "seed" is the seed used to generate the testing image in case no experimental data is supplied. You can change with this parameter to randomize the testing image in "demo3.py" (**Fig. 12**).

demo2.py

```python
# ==========================================================================================================
# Demo 2 demonstrates how to use a model trained on simulations to localize experimental data. Again, the experimental
# frames are taken from our STORM experiment (Fig. 3 main text). Note that the frames saved in the folder
# <Experimental_Data/Tetrapod_demo2/> are those after minimum subtraction.
# ==========================================================================================================

# import related script and packages to load the exp. image
from DeepSTORM3D.Testing_Localization_Model import test_model
import matplotlib.pyplot as plt
import os

# pre-trained weights on simulations
path_curr = os.getcwd()
path_results = path_curr + '/Demos/Results_Tetrapod_demo2/'

# path to experimental images
path_exp_data = path_curr + '/Experimental_Data/Tetrapod_demo2/'

# postprocessing parameters
postprocessing_params = {'thresh': 40, 'radius': 4}

# test the model by comparing the regenerated image alongside the 3D positions
xyz_rec, conf_rec = test_model(path_results, postprocessing_params, path_exp_data)

# show all plots
plt.show()
```

Figure. 11 – demo 2.

demo3.py

```python
# ====================================================================================================================
# Demo 3 demonstrates how to use a model trained on simulations to localize a simulated example with non-uniform
# background and emitter size. The parameters assumed for this demo match our Telomere simulations (Fig. 4 main text).
# This script also prints quantitative metrics and compare the result to the GT simulated positions.
# ====================================================================================================================

# test a Tetrapod pre-trained model
from DeepSTORM3D.Testing_Localization_Model import test_model
import matplotlib.pyplot as plt
import os

# path to the learning results
path_curr_dir = os.getcwd()
path_results = path_curr_dir + '/Demos/Results_Tetrapod_demo3/'

# set the postprocessing parameters
postprocessing_params = {'thresh': 80, 'radius': 4}

# model testing
seed = 11  # you can change this to randomize the sampled example
test_model(path_results, postprocessing_params, None, seed)

# show all plots
plt.show()
```

Figure. 12 – demo 3.

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.