PyTorch Connectomics Documentation

version latest

Zudi Lin and Donglai Wei

December 27, 2020

Contents

PyTorch Connectomics documentation

PyTorch Connectomics is a deep learning framework for automatic and semi-automatic annotation of connectomics datasets, powered by PyTorch.

> ***Note***
>
> This package is under development and should not be considered as formally released.

The field of connectomics aims to reconstruct the wiring diagram of the brain by mapping the neural connections at the level of individual synapses. Recent advances in electronic microscopy (EM) have enabled the collection of a large number of image stacks at nanometer resolution, but the annotation requires expertise and is super time-consuming.

PyTorch Connectomics consists of various deep learning based object detection, semantic segmentation and instance segmentation methods for the annotation and analysis of 3D image stacks. In addition, it consists of an easy-to-use data augmentation interface, tutorials on several common benchmark datasets, and helpful image stack processing functions, both for reproducing state-of-the-art results on benchmark datasets, and labelling large-scale volumes.

Installation

The code is developed and tested on a machine with 8 NVIDIA GPUs with the CentOS Linux 7.4 (Core) operation system.

> ***Note***
>
> We do not recommend installation as root user on your system python. Please setup an Anaconda/Miniconda environment and add the required packages to the environment.

Please follow the steps below for a successful installation:

1. Create a new conda environment:

```
$ conda create -n py3_torch python=3.8
$ source activate py3_torch
$ conda install pytorch torchvision cudatoolkit=10.2 -c pytorch
```

2. Ensure that at least PyTorch 1.5.1 is installed:

```
$ python -c 'import torch; print(torch.__version__)'
>>> 1.5.1
```

3. Ensure CUDA is setup correctly (optional):

    1. Check if PyTorch is installed with CUDA support:

```
$ python -c 'import torch; print(torch.cuda.is_available())'
>>> True
```

    2. Add CUDA to `$PATH` and `$CPATH` (note that your actual CUDA path may vary from `/usr/local/cuda`):

```
$ PATH=/usr/local/cuda/bin:$PATH
$ echo $PATH
>>> /usr/local/cuda/bin:...

$ CPATH=/usr/local/cuda/include:$CPATH
```

```
$ echo $CPATH
>>> /usr/local/cuda/include:...
```

3. Verify that `nvcc` is accessible from terminal:

```
$ nvcc --version
>>> 10.2
```

4. Ensure that PyTorch and system CUDA versions match:

```
$ python -c 'import torch; print(torch.version.cuda)'
>>> 10.2

$ nvcc --version
>>> 10.2
```

The codebased is mainly developed and tested on the Harvard FASRC cluster. Please load required CUDA modules from the RC server module list during running and development on the RC server.

4. Download and install the package:

```
$ git clone https://github.com/zudi-lin/pytorch_connectomics.git
$ cd pytorch_connectomics
$ pip install --upgrade pip
$ pip install -r requirements.txt
$ pip install --editable .
```

We install the package in editable mode by default so that there is no need to re-install it when making changes to the code.

Note

If you meet compilation errors, please open an issue and describe the steps to reproduce the errors. It is highly recommended to first play with the demo Jupyter notebooks to make sure that the installation is correct and also have an intial taste of the modules.

Configurations

PyTorch Connectomics uses a key-value based configuration system that can be adjusted to carry out standard and commonly used tasks. The configuration system is built with YACS that uses YAML, a human-readable data-serialization language, to manage options.

Note

The system has `_C.KEY:VALUE` field, which will use a pre-defined configurations first. Values in the pre-defined config will be overwritten in sub-configs, if there are any according to the user requirements. We provided several base configs for standard tasks in connectomics research as `<task>.yaml` files at pytorch_connectomics/configs/.

We do not expect all features in the package to be available through configs, which will make it too complicated. If you need to add options that are not available in the current version, please modify the keys-value pairs in `/connectomics/config/config.py`

Basic Usage

Some basic usage of the `CfgNode` object in YACS is shown below:

```
from yacs.config import CfgNode as CN
_C = CN()            # config definition
_C.SYSTEM = CN()     # config definition for GPU and CPU
_C.MODEL = CN()      # Model architectures defined in the package
_C.MODEL.ARCHITECTURE = 'unet_residual_3d'
```

The configs in PyTorch Connectomics also accepts command line configuration overwrite, i.e.: Key-value pairs provided in the command line will overwrite the existing values in the config file. For example, we can add arguments when executing `scripts/main.py`:

```
python -u scripts/main.py \
--config-file configs/Lucchi-Mitochondria.yaml SOLVER.ITERATION_TOTAL 30000
```

To see a list of available configs in PyTorch Connectomics and what they mean, check Config References.

Multiple Losses for a Single Learning Target

Sometimes training with a single loss function does not produce favorable predictions. Thus we provide a simple way to specify multiple loss functions for training the segmentation models. For example, to use the weighted binary cross-entropy loss (`WeightedBCE`) and the soft Sørensen–Dice loss (`DiceLoss`) at the same time, we can change the key-value pairs of `LOSS_OPTION` in the `config.yaml` file by doing:

```
MODEL:
  LOSS_OPTION: [['WeightedBCE', 'DiceLoss']]
  LOSS_WEIGHT: [[1.0, 0.5]]
  WEIGHT_OPT: [['1', '0']]
```

`LOSS_OPTION`: the loss criterions to be used during training. `LOSS_WEIGHT`: the relative weight of each loss function. `WEIGHT_OPT`: the option for generating pixel-wise loss mask (set to '0' disable).

If you only want to use weighted binary cross-entropy loss, do:

```
MODEL:
  LOSS_OPTION: [['WeightedBCE']]
  LOSS_WEIGHT: [[1.0]]
  WEIGHT_OPT: [['1']]
```

Multitask Learning

To conduct multitask learning, which predicts multiple targets given a image volume, we can further adjust the `TARGET_OPT` option. For example, to conduct instance segmentation of mitochondria, we can predict not only the binary foreground mask but also the instance boundary to distinguish closely touching objects. Specifically, we can use the following options:

```
MODEL:
  TARGET_OPT: ['0', '4-2-1']
  LOSS_OPTION: [['WeightedBCE', 'DiceLoss'], ['WeightedBCE']]
  LOSS_WEIGHT: [[1.0, 1.0], [1.0]]
  WEIGHT_OPT: [['1', '0'], ['1']]
```

`TARGET_OPT`: a list of the targets to learn.

Currently five kinds of `TARGET_OPT` are supported:

- `'0'`: binary foreground mask (used in the mitochondria segmentation tutorial).
- `'1'`: synaptic polarity mask (used in the synaptic polairty tutorial).

- `'2'`: affinity map (used in the neuron segmentation tutorial).
- `'3'`: masks of small objects only.
- `'4'`: instance boundaries (used in the mitochondria segmentation tutorial).
- `'9'`: generic segmantic segmentation. Supposing there are 12 classes (including one background class) to predict, we need to set `MODEL.OUT_PLANES: 12` and `MODEL.TARGET_OPT: ['9-12']`. Here 9 represent the multi-class semantic segmentation task, while 12 in `['9-12']` represents the 12 semantic classes.

More options will be provided soon!

Inference

Most of the config options are shared by training and inference. However, there are several options to be adjusted at inference time by the `update_inference_cfg` function:

```
def update_inference_cfg(cfg):
   r"""Update configurations (cfg) when running mode is inference.

   Note that None type is not supported in current release of YACS (0.1.7), but will be
   supported soon according to this pull request: https://github.com/rbgirshick/yacs/pull/18
   Therefore a re-organization of the configurations using None type will be done when YACS
   0.1.8 is released.
   """
   # Dataset configurations:
   if len(cfg.INFERENCE.INPUT_PATH) != 0:
      cfg.DATASET.INPUT_PATH = cfg.INFERENCE.INPUT_PATH
   cfg.DATASET.IMAGE_NAME = cfg.INFERENCE.IMAGE_NAME
   cfg.DATASET.OUTPUT_PATH = cfg.INFERENCE.OUTPUT_PATH
   if len(cfg.INFERENCE.PAD_SIZE) != 0:
      cfg.DATASET.PAD_SIZE = cfg.INFERENCE.PAD_SIZE

   # Model configurations:
   if len(cfg.INFERENCE.INPUT_SIZE) != 0:
      cfg.MODEL.INPUT_SIZE = cfg.INFERENCE.INPUT_SIZE
   if len(cfg.INFERENCE.OUTPUT_SIZE) != 0:
      cfg.MODEL.OUTPUT_SIZE = cfg.INFERENCE.OUTPUT_SIZE
```

There are also several options exclusive for inference. For example:

```
INFERENCE:
  AUG_MODE: 'mean' # options for test augmentation
  AUG_NUM: 4
  BLENDING: 'gaussian' # blending function for overlapping inference
  STRIDE: (4, 128, 128) # sampling stride for inference
  SAMPLES_PER_BATCH: 16 # batchsize for inference
```

Data Loading

Data Augmentation

Since many semi-supervised and unsupervised learning tasks do not require labels, the only key required in our data augmentor is `'image'`. Let's look at an example for using an augmentation pipeline on input images:

```
from connectomics.data.augmentation import *
tranforms = [
    Rescale(p=0.8),
```

```
    MisAlignment(displacement=16,
                 rotate_ratio=0.5,
                 p=0.5),
    CutBlur(length_ratio=0.6,
            down_ratio_min=4.0,
            down_ratio_max=8.0,
            p=0.7),
]
augmentor = Compose(tranforms,
                    input_size = (8, 256, 256))

sample = {'image': image}
augmented = augmentor(sample)
```

Then the augmented data can be retrived using the corresponding key. Our augmentor can also apply the same set of transformations to the input images and all other specified targets. For example, under the supervised segmentation setting, an label image/volume contains the segmentation masks and a valid mask indicating the annotated regions are required. We provide the `additional_targets` option to handle those targets:

```
from connectomics.data.augmentation import *
additional_targets = {'label': 'mask',
                      'valid_mask': 'mask'}

tranforms = [
    Rescale(p=0.8,
            additional_targets=additional_targets),
    MisAlignment(displacement=16,
                 rotate_ratio=0.5,
                 p=0.5,
                 additional_targets=additional_targets),
    CutBlur(length_ratio=0.6,
            down_ratio_min=4.0,
            down_ratio_max=8.0,
            p=0.7,
            additional_targets=additional_targets),
]
augmentor = Compose(tranforms,
                    input_size = (8, 256, 256),
                    additional_targets=additional_targets)

sample = {'image': image,
          'label': label,
          'valid_mask': valid_mask}
augmented = augmentor(sample)
```

Note

Each addition target need to be specified with a name (*e.g.*, `'valid_mask'`) and a target type (`'img'` or `'mask'`). Some augmentations are only applied to `'img'`, and augmentations for both `'img'` and `'mask'` will use different interpolation modes for them.

The `'label'` key in `'mask'` target type is used by default in the configuration file as most of the tutorial examples belong to the supervised training category. For model training with partially annotated dataset under the supervised setting, we need to add:

```
AUGMENTOR:
  ADDITIONAL_TARGETS_NAME: ['label', 'valid_mask']
  ADDITIONAL_TARGETS_TYPE: ['mask', 'mask']
```

Rejection Sampling

TileDataset

Neuron Segmentation

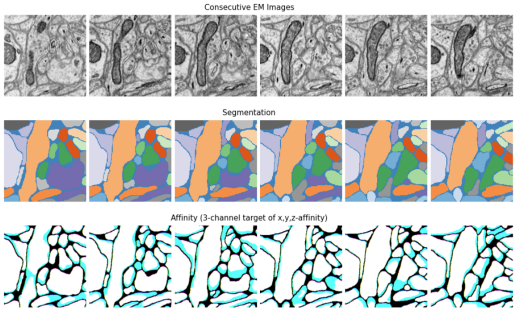
This tutorial provides step-by-step guidance for neuron segmentation with SENMI3D benchmark datasets. Dense neuron segmentation in electronic microscopy (EM) images belongs to the category of **instance segmentation**. The methodology is to first predict the affinity map (the connectivity of each pixel to neighboring pixels) with an encoder-decoder ConvNets and then generate the segmentation map using a standard segmentation algorithm (e.g., watershed).

The evaluation of segmentation results is based on the Rand Index and Variation of Information.

Note

Before running neuron segmentation, please take a look at the demos to get familiar with the datasets and available utility functions in this package.

The main script to run the training and inference is `pytorch_connectomics/scripts/main.py`. The pytorch target affinity generation is **`connectomics.data.dataset.VolumeDataset`**.



Examples of EM images, segmentation and affinity map from the SNEMI3D dataset. Since the affinity map has 3 channels, we can visualize them as RGB images.

1. Get the dataset:

```
wget http://rhoana.rc.fas.harvard.edu/dataset/snemi.zip
```

For description of the data please check this page.

Note

The affinity value of a pixel (voxel) can be 1 even at the segment boundary. Thus we usually widen the instance border (erode the instance mask) to let the model make more conservative predictions to prevent merge error.

2. Provide the `yaml` configuration file to run training:

```
$ source activate py3_torch
$ python scripts/main.py --config-file configs/SNEMI-Neuron.yaml
```

The configuration file for training is in `configs/SNEMI-Neuron.yaml`. We usually create a `datasets/` folder under `pytorch_connectomics` and put the SNEMI dataset there. Please modify the following options according to your system configuration and data storage:

- `IMAGE_NAME`: Name of the volume file (HDF5 or TIFF).
- `LABEL_NAME`: Name of the label file (HDF5 or TIFF).
- `INPUT_PATH`: Path to both input files above.
- `OUTPUT_PATH`: Path to store outputs (checkpoints and Tensorboard events).
- `NUM_GPUS`: Number of GPUs to use.
- `NUM_CPUS`: Number of CPU cores to use (for data loading).

3. (*Optional*) To run training starting from pretrained weights, add a checkpoint file:

```
$ source activate py3_torch
$ python scripts/main.py --config-file configs/SNEMI-Neuron.yaml \
$ --checkpoint /path/to/checkpoint/checkpoint_xxxxx.pth.tar
```

4. Visualize the training progress:

```
$ tensorboard --logdir outputs/SNEMI3D/
```

5. Run inference on image volumes (add `--inference`). During inference the model can use larger batch sizes or take bigger inputs. Test-time augmentation is also applied by default:

```
$ python scripts/main.py --config-file configs/SNEMI-Neuron.yaml
--inference --checkpoint outputs/SNEMI3D/checkpoint_xxxxx.pth
```

6. Generate segmentation and run evaluation:

    1. Download the `waterz` package:

    ```
    $ git clone git@github.com:zudi-lin/waterz.git
    $ cd waterz
    $ pip install --editable .
    ```

    2. Download the `zwatershed` package:

    ```
    $ git clone git@github.com:zudi-lin/zwatershed.git
    $ cd zwatershed
    $ pip install --editable .
    ```

    3. Generate 3D segmentation and report Rand and VI score using `waterz`. Please see examples at https://github.com/zudi-lin/waterz.

Mitochondria Segmentation

Introduction

Mitochondria are the primary energy providers for cell activities, thus essential for metabolism. Quantification of the size and geometry of mitochondria is not only crucial to basic neuroscience research, but also informative to clinical studies including, but not limited to, bipolar disorder and diabetes.

This tutorial has two parts. In the first part, you will learn how to make **pixel-wise class prediction** on the widely used benchmark dataset released by Lucchi et al. in 2012. In the second part, you will learn how to predict the **instance masks** of individual mitochondrion from the large-scale MitoEM dataset released by Wei et al. in 2020.

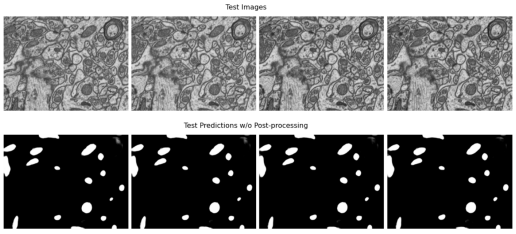
Semantic Segmentation

This section provides step-by-step guidance for mitochondria segmentation with the EM benchmark datasets released by Lucchi et al.. We consider the task as a **semantic segmentation** task and predict the mitochondria pixels with encoder-decoder ConvNets similar to the models used in affinity prediction in neuron segmentation. The evaluation of the mitochondria segmentation results is based on the F1 score and Intersection over Union (IoU).

Note

Different from other EM connectomics datasets used in the tutorials, the dataset released by Lucchi et al. is an isotropic dataset, which means the spatial resolution along all three axes is the same. Therefore a completely 3D U-Net and data augmentation along z-x and z-y planes besides x-y planes are preferred.

All the scripts needed for this tutorial can be found at `pytorch_connectomics/scripts/`. Need to pass the argument `--config-file configs/Lucchi-Mitochondria.yaml` during training and inference to load the required configurations for this task. The pytorch dataset class of lucchi data is **`connectomics.data.dataset.VolumeDataset`**.



Qualitative results of the model prediction on the mitochondria segmentation dataset released by Lucchi et al., without any post-processing.

1. Get the dataset:

   Download the dataset from our server:

   ```
   wget http://rhoana.rc.fas.harvard.edu/dataset/lucchi.zip
   ```

   For description of the data please check the author page.

2. Run the training script:

   ```
   $ source activate py3_torch
   $ CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 python -u scripts/main.py \
     --config-file configs/Lucchi-Mitochondria.yaml
   ```

3. Visualize the training progress:

   ```
   $ tensorboard --logdir runs
   ```

4. Run inference on test image volume:

   ```
   $ source activate py3_torch
   $ CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 python -u scripts/main.py \
     --config-file configs/Lucchi-Mitochondria.yaml --inference \
     --checkpoint outputs/Lucchi_mito_baseline/volume_100000.pth.tar
   ```

5. Since the ground-truth label of the test set is public, we can run the evaluation locally:

   ```
   from connectomics.utils.evaluation import get_binary_jaccard
   pred = pred / 255. # output is casted to uint8 with range [0,255].
   gt = (gt!=0).astype(np.uint8)
   thres = [0.4, 0.6, 0.8] # evaluate at multiple thresholds.
   scores = get_binary_jaccard(pred, gt, thres)
   ```

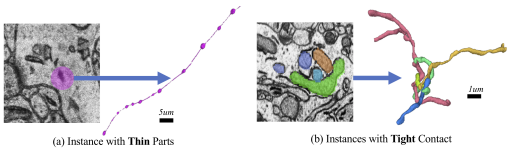
   The prediction can be further improved by conducting median filtering to remove noise:

```
from connectomics.utils.evaluation import get_binary_jaccard
from connectomics.utils.processing import binarize_and_median
pred = pred / 255. # output is casted to uint8 with range [0,255].
pred = binarize_and_median(pred, size=(7,7,7), thres=0.8)
gt = (gt!=0).astype(np.uint8)
scores = get_binary_jaccard(pred, gt) # prediction is already binarized
```

Our pretained model achieves a foreground IoU and IoU of **0.892** and **0.943** on the test set, respectively. The results are better or on par with state-of-the-art approaches. Please check BENCHMARK.md for detailed performance comparison and the pre-trained models.

Instance Segmentation

This section provides step-by-step guidance for mitochondria segmentation with our benchmark datasets MitoEM. We consider the task as 3D **instance segmentation** task and provide three different confiurations of the model output. The model is `unet_res_3d`, similar to the one used in neuron segmentation. The evaluation of the segmentation results is based on the AP-75 (average precision with an IoU threshold of 0.75).



Complex mitochondria in the MitoEM dataset:(a) mitochondria-on-a-string (MOAS), and (b) dense tangle of touching instances. Those challenging cases are prevalent but not covered in previous datasets.

Note

The MitoEM dataset has two sub-datasets **Rat** and **Human** based on the source of the tissues. Three training configuration files on **MitoEM-Rat** are provided in `pytorch_connectomics/configs/MitoEM/` for different learning targets of the model.

Note

Since the dataset is very large and can not be directly loaded into memory, we use the **`connectomics.data.dataset.TileDataset`** dataset class that only loads part of the whole volume by opening involved PNG images.

1. Introduction to the dataset:

   On the Harvard RC cluster, the datasets can be found at:

   ```
   /n/pfister_lab2/Lab/vcg_connectomics/mitochondria/miccai2020/rat
   ```

   and

   ```
   /n/pfister_lab2/Lab/vcg_connectomics/mitochondria/miccai2020/human
   ```

   For the public link of the dataset, check the project page.

   Dataset description:

   - `im`: includes 1,000 single-channel `*.png` files (**4096x4096**) of raw EM images (with a spatial resolution of **30x8x8** nm).
   - `mito`: includes 1,000 single-channel `*.png` files (**4096x4096**) of instance labels.
   - `*.json`: **`Dict`** contains paths to `*.png` files

2. Configure `.yaml` files for different learning targets.

    - `MitoEM-R-A.yaml`: output 3 channels for affinty prediction.
    - `MitoEM-R-AC.yaml`: output 4 channels for both affinity and instance contour prediction.
    - `MitoEM-R-BC.yaml`: output 2 channels for both binary mask and instance contour prediction.

3. Run the training script.

> ***Note***
>
> By default the path of images and labels are not specified. To run the training scripts, please revise the `DATASET.IMAGE_NAME`, `DATASET.LABEL_NAME`, `DATASET.OUTPUT_PATH` and `DATASET.INPUT_PATH` options in `configs/MitoEM-R-*.yaml`. The options can also be given as command-line arguments without changing of the `yaml` configuration files.

```
$ source activate py3_torch
$ python -u scripts/main.py --config-file configs/MitoEM-R-A.yaml
```

4. Visualize the training progress. More info here:

```
$ tensorboard --logdir ``OUTPUT_PATH/<EXP_DIR_NAME>``
```

> ***Note***
>
> Our utility functions will create a subdir in OUTPUT_PATH to save the Tensorboard event files. Substitute **<EXP_DIR_NAME>** with your subdir name.

5. Run inference on image volumes:

```
$ source activate py3_torch
$ python -u scripts/main.py \
  --config-file configs/MitoEM-R-A.yaml --inference \
  --checkpoint OUTPUT_PATH/checkpoint_<ITER_NUM>.pth.tar
```

> ***Note***
>
> Please change the `INFERENCE.IMAGE_NAME` `INFERENCE.OUTPUT_PATH` `INFERENCE.OUTPUT_NAME` options in `configs/MitoEM-R-A.yaml`.

Synapse Detection

Introduction

A synapse is an essential structure in the nervous system that allows an electric or chemical signal to be passed to another neuron or an effector cell (*e.g.*, muscle fiber). Identification of synapses is important for reconstructing the wiring diagram of neurons to enable new insights into the workings of the brain, which is the long-term goal of the connectomics area. Signal flows in one direction at a synapse, therefore each synapse usually consists of a pre-synaptic region and a post-synaptic region.

This tutorial has two parts. In the first part, you will learn how to detect **synaptic clefts** by predicting the synaptic cleft pixels on the CREMI Challenge dataset from adult *Drosophila melanogaster* brain tissue. This dataset is released in 2016. In the second part, you will learn how to predict the **synaptic polarity masks** to demonstrate the signal flow between neurons using the dataset released by Lin et al. in 2020. The brain sample is collected from Layer II/III in the primary visual cortex of an adult rat.

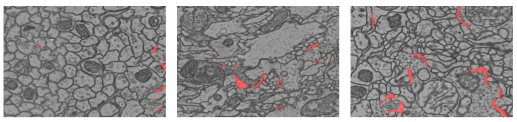
Synaptic Cleft Detection

This tutorial provides step-by-step guidance for synaptic cleft detection with CREMI benchmark datasets. We consider the task as a semantic segmentation task and predict the synapse pixels with encoder-decoder ConvNets similar to the models used in affinity prediction in neuron segmentation. The evaluation of the synapse detection results is based on the F1 score and average distance. See CREMI metrics for more details.

Note

We preform re-alignment of the original CREMI image stacks and also remove the crack artifacts. Please reverse the alignment before submitting the test prediction to the CREMI challenge.

Script needed for this tutorial can be found at `pytorch_connectomics/scripts/`. The *YAML* configuration files can be found at `pytorch_connectomics/configs/`, which stores the common settings for model training and inference. Other default configuration options can be found at `pytorch_connectomics/connectomics/config/`. The pytorch dataset class of the synaptic cleft detection task is **`connectomics.data.dataset.VolumeDataset`**.



Qualitative results of the synaptic cleft prediction (red segments) on the CREMI challenge test volumes. The three images from left to right are cropped from volume A+, B+, and C+, respectively.

1. Get the dataset:

   Download the dataset from the Harvard RC server:

   ```
   wget http://rhoana.rc.fas.harvard.edu/dataset/cremi.zip
   ```

   For description of the data please check this page.

   ### Note

   If you use the original CREMI challenge datasets or the data processed by yourself, the file names can be different from the default ones. In such case, please change the corresponding entries, including `IMAGE_NAME`, `LABEL_NAME` and `INPUT_PATH` in the CREMI config file.

2. Run the main.py script for training. This script can take a list of volumes and conduct training/inference at the same time.

   ```
   $ source activate py3_torch
   $ CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 python -u scripts/main.py \
     --config-file configs/CREMI-Synaptic-Cleft.yaml
   ```

   - `config-file`: configuration setting for the current experiment.

3. Visualize the training progress:

   ```
   $ tensorboard --logdir runs
   ```

4. Run the main.py script for inference:

```
$ CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 python -u scripts/main.py \
  --config-file configs/CREMI-Synaptic-Cleft.yaml \
  --checkpoint outputs/CREMI_syn_baseline/volume_50000.pth.tar \
  --inference
```

   - `config-file`: configuration setting for current experiments.
   - `inference`: will run inference when given, otherwise will run training instead.
   - `checkpoint`: the pre-trained checkpoint file for inference.

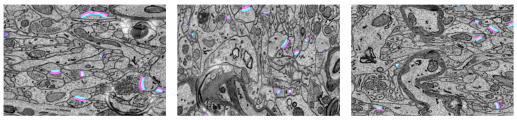
Synaptic Polarity Detection

This tutorial provides step-by-step guidance for synaptic polarity detection with the EM-R50 dataset released by Lin et al. in 2020. This task is different from the synaptic cleft detection task in two aspects. First, this one requires distinguishing different synapses, while the cleft detection task only needs the binary foreground mask for evaluation. Second, the polarity detection task also requires separated pre-synaptic and post-synaptic masks. The evaluation metric of the synaptic polarity detection results is an IoU-based F1 score. The sparsity and diversity of synapses make the task challenging.

Note

We tackle the task using a bottom-up approach that first generates the segmentation masks of synaptic regions and then apply post-processing algorithms like connected component labeling to separate individual synapses. Our segmentation model uses a model target of three channels. The three channels are **pre-synaptic region**, **post-synaptic region** and **synaptic region** (union of the first two channels), respectively.

All the scripts needed for this tutorial can be found at `pytorch_connectomics/scripts/`. The pytorch dataset class of synaptic partners is **`connectomics.data.dataset.VolumeDataset`**.



Qualitative results of the synaptic polarity prediction on the EM-R50 dataset. The three-channel outputs that consist of pre-synaptic region, post-synaptic region and their union (synaptic region) are visualizd in color on the EM images. The single flows from the magenta sides to the cyan sides between neurons.

1. Get the dataset:

   Download the example dataset for synaptic polarity detection from our server:

   ```
   wget http://rhoana.rc.fas.harvard.edu/dataset/jwr15_synapse.zip
   ```

2. Run the training script. The training and inference script can take a list of volumes (separated by '@') in either the yaml config file or by command-line arguments.

Note

By default the path of images and labels are not specified. To run the training scripts, please revise the `IMAGE_NAME`, `LABEL_NAME` and `INPUT_PATH` options in `configs/Synaptic-Polarity.yaml`. The options can also be given as command-line arguments without changing of the `yaml` configuration files.

```
$ source activate py3_torch
$ CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 python -u scripts/main.py \
  --config-file configs/Synaptic-Polarity.yaml
```


```
$ source activate py3_torch
$ CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 python -u scripts/main.py \
  --config-file configs/Synaptic-Polarity.yaml
```


> ***Note***
>
> We add **higher weights** to the foreground pixels and apply **rejection sampling** to reject samples without synapes during training to heavily penalize false negatives. This is beneficial for down-stream proofreading and analysis as correcting false positives is much easier than finding missing synapses in the vast volumes.

3. Visualize the training progress. More info here:

```
$ tensorboard --logdir outputs/synaptic_polarity
```

4. Run inference on image volumes:

```
$ source activate py3_torch
$ CUDA_VISIBLE_DEVICES=0,1,2,3,4,5,6,7 python -u scripts/main.py \
  --config-file configs/Synaptic-Polarity.yaml --inference \
  --checkpoint outputs/synaptic_polarity/volume_xxxxx.pth.tar
```

> ***Note***
>
> By default the path of images for inference are not specified. Please change the `INFERENCE.IMAGE_NAME` option in `configs/Synaptic-Polarity.yaml`.

5. Apply post-processing algorithms. Use this function to convert the probability map into instance/semantic segmentation masks based on the application.

connectomics.data.datasets

class `connectomics.data.dataset.`**`TileDataset`** (chunk_num: List[int] = [2, 2, 2], chunk_num_ind: Optional[list] = None, chunk_iter: int = - 1, chunk_stride: bool = True, volume_json: str = 'path/to/image', label_json: Optional[str] = None, valid_mask_json: Optional[str] = None, valid_ratio: float = 0.5, sample_volume_size: tuple = (8, 64, 64), sample_label_size: Optional[tuple] = None, sample_stride: tuple = (1, 1, 1), augmentor: Optional[connectomics.data.augmentation.composition.Compose] = None, target_opt: List[str] = ['0'], weight_opt: List[List[str]] = [['1']], mode: str = 'train', do_2d: bool = False, label_erosion: int = 0, pad_size: List[int] = [0, 0, 0], reject_size_thres: int = 0, reject_p: float = 0.95)

Dataset class for large-scale tile-based datasets. Large-scale volumetric datasets are usually stored as individual tiles. Directly loading them as a single array for training and inference is infeasible. This class reads the paths of the tiles and construct smaller chunks for processing.

Parameters:

- **chunk_num** (*list*) – volume spliting parameters in (z, y, x) order. Default: [2, 2, 2]
- **chunk_num_ind** (*list*) – predefined list of chunks. Default: None
- **chunk_iter** (*int*) – number of iterations on each chunk. Default: -1
- **chunk_stride** (*bool*) – allow overlap between chunks. Default: True
- **volume_json** (*str*) – json file for input image. Default: `'path/to/image'`
- **label_json** (*str, optional*) – json file for label. Default: None
- **valid_mask_json** (*str, optional*) – json file for valid mask. Default: None
- **valid_ratio** (*float*) – volume ratio threshold for valid samples. Default: 0.5
- **sample_volume_size** (*tuple, int*) – model input size.
- **sample_label_size** (*tuple, int*) – model output size.
- **sample_stride** (*tuple, int*) – stride size for sampling.
- **augmentor** (*connectomics.data.augmentation.composition.Compose, optional*) – data augmentor for training. Default: None
- **target_opt** (*list*) – list the model targets generated from segmentation labels.
- **weight_opt** (*list*) – list of options for generating pixel-wise weight masks.
- **mode** (*str*) – `'train'`, `'val'` or `'test'`. Default: `'train'`
- **do_2d** (*bool*) – load 2d samples from 3d volumes. Default: False
- **label_erosion** (*int*) – label erosion parameter to widen border. Default: 0
- **pad_size** (*list*) – padding parameters in (z, y, x) order. Default: [0, 0, 0]
- **reject_size_thres** (*int*) – threshold to decide if a sampled volumes contains foreground objects. Default: 0
- **reject_p** (*float*) – probability of rejecting non-foreground volumes. Default: 0.95

class `connectomics.data.dataset.`**`VolumeDataset`** (volume: list, label: Optional[list] = None, valid_mask: Optional[list] = None, valid_ratio: float = 0.5, sample_volume_size: tuple = (8, 64, 64), sample_label_size: tuple = (8, 64, 64), sample_stride: tuple = (1, 1, 1), augmentor: Optional[connectomics.data.augmentation.composition.Compose] = None, target_opt: List[str] = ['1'], weight_opt: List[List[str]] = [['1']], mode: str = 'train', do_2d: bool = False, iter_num: int = - 1, reject_size_thres: int = 0, reject_p: float = 0.95)

Dataset class for volumetric image datasets. At training time, subvolumes are randomly sampled from all the large input volumes with (optional) rejection sampling to increase the frequency of foreground regions in a batch. At inference time, subvolumes are yielded in a sliding-window manner with overlap to counter border artifacts.

Parameters:

- **volume** (*list*) – list of image volumes.
- **label** (*list, optional*) – list of label volumes. Default: None
- **valid_mask** (*list, optional*) – list of valid masks. Default: None
- **valid_ratio** (*float*) – volume ratio threshold for valid samples. Default: 0.5
- **sample_volume_size** (*tuple, int*) – model input size.
- **sample_label_size** (*tuple, int*) – model output size.
- **sample_stride** (*tuple, int*) – stride size for sampling.
- **augmentor** (*connectomics.data.augmentation.composition.Compose, optional*) – data augmentor for training. Default: None
- **target_opt** (*list*) – list the model targets generated from segmentation labels.
- **weight_opt** (*list*) – list of options for generating pixel-wise weight masks.
- **mode** (*str*) – `'train'`, `'val'` or `'test'`. Default: `'train'`
- **do_2d** (*bool*) – load 2d samples from 3d volumes. Default: False
- **iter_num** (*int*) – total number of training iterations (-1 for inference). Default: -1
- **reject_size_thres** (*int*) – threshold to decide if a sampled volumes contains foreground objects. Default: 0
- **reject_p** (*float*) – probability of rejecting non-foreground volumes. Default: 0.95

Note

For relatively small volumes, the total number of possible subvolumes can be smaller than the total number of samples required in training (the product of total iterations and mini-natch size), which raises *StopIteration*. Therefore the dataset length is also decided by the training settings.

connectomics.data.augmentation

class `connectomics.data.augmentation.`**`Compose`**(transforms: list = [], input_size: tuple = (8, 256, 256), smooth: bool = True, keep_uncropped: bool = False, keep_non_smoothed: bool = False, additional_targets: Optional[dict] = None)

Composing a list of data transforms.

The sample size of the composed augmentor can be larger than the specified input size of the model to ensure that all pixels are valid after center-crop.

Parameters:

- **transforms** (*list*) – list of transformations to compose.
- **input_size** (*tuple*) – input size of model in (z, y, x) order. Default: $(8, 256, 256)$
- **smooth** (*bool*) – smoothing the object mask with Gaussian filtering. Default: True
- **keep_uncropped** (*bool*) – keep uncropped image and label. Default: False
- **keep_non_smooth** (*bool*) – keep the non-smoothed object mask. Default: False
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

Examples::

```
>>> # specify addtional targets besides 'image'
>>> kwargs = {'additional_targets': {'label': 'mask'}}
>>> augmentor = Compose([Rotate(p=1.0, **kwargs),
>>>                      Flip(p=1.0, **kwargs),
>>>                      Elastic(alpha=12.0, p=0.75, **kwargs),
```

```
>>>                     Grayscale(p=0.75, **kwargs),
>>>                     MissingParts(p=0.9, **kwargs)],
>>>                     input_size = (8, 256, 256), **kwargs)
>>> sample = {'image':input, 'label':label}
>>> augmented = augmentor(data)
>>> out_input, out_label = augmented['image'], augmented['label']
```

class connectomics.data.augmentation.**CutBlur**(length_ratio: float = 0.25, down_ratio_min: float = 2.0, down_ratio_max: float = 8.0, downsample_z: bool = False, p: float = 0.5, additional_targets: Optional[dict] = None)

3D CutBlur data augmentation, adapted from https://arxiv.org/abs/2004.00448.

Randomly downsample a cuboid region in the volume to force the model to learn super-resolution when making predictions. This augmentation is only applied to images.

Parameters:

- **length_ratio** (*float*) – the ratio of the cuboid length compared with volume length.
- **down_ratio_min** (*float*) – minimal downsample ratio to generate low-res region.
- **down_ratio_max** (*float*) – maximal downsample ratio to generate low-res region.
- **downsample_z** (*bool*) – downsample along the z axis (default: False).
- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params()

Calculate the appropriate sample size with data augmentation.

Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**CutNoise**(length_ratio: float = 0.25, mode: str = 'uniform', scale: float = 0.2, p: float = 0.5, additional_targets: Optional[dict] = None)

3D CutNoise data augmentation.

Randomly add noise to a cuboid region in the volume to force the model to learn denoising when making predictions. This augmentation is only applied to images.

Parameters:

- **length_ratio** (*float*) – the ratio of the cuboid length compared with volume length.
- **mode** (*string*) – the distribution of the noise pattern. Default: `'uniform'`.
- **scale** (*float*) – scale of the random noise. Default: 0.2.
- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params()

Calculate the appropriate sample size with data augmentation.

Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**DataAugment**(p: float = 0.5, additional_targets: Optional[dict] = None)

DataAugment interface. A data augmentor needs to conduct the following steps:

1. Set **`sample_params`** at initialization to compute required sample size.
2. Randomly generate augmentation parameters for the current transform.
3. Apply the transform to a pair of images and corresponding labels.

All the real data augmentations (except mix-up augmentor and test-time augmentor) should be a subclass of this class.

Parameters:

- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params ()

Calculate the appropriate sample size with data augmentation.
Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**Elastic** (alpha: float = 16.0, sigma: float = 4.0, p: float = 0.5, additional_targets: Optional[dict] = None)

Elastic deformation of images as described in [Simard2003] (with modifications). The implementation is based on https://gist.github.com/erniejunior/601cdf56d2b424757de5. This augmentation is applied to both images and masks.

Simard2003 Simard, Steinkraus and Platt, “Best Practices for Convolutional Neural Networks applied to Visual Document Analysis”, in Proc. of the International Conference on Document Analysis and Recognition, 2003.

Parameters:

- **alpha** (*float*) – maximum pixel-moving distance of elastic deformation. Default: 10.0
- **sigma** (*float*) – standard deviation of the Gaussian filter. Default: 4.0
- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params ()

Calculate the appropriate sample size with data augmentation.
Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**Flip** (do_ztrans: int = 0, p: float = 0.5, additional_targets: Optional[dict] = None)

Randomly flip along *z*-, *y*- and *x*-axes as well as swap *y*- and *x*-axes for anisotropic image volumes. For learning on isotropic image volumes set **do_ztrans** to 1 to swap *z*- and *x*-axes (the inputs need to be cubic). This augmentation is applied to both images and masks.

Parameters:

- **do_ztrans** (*int*) – set to 1 to swap z- and x-axes for isotropic data. Default: 0
- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params ()

Calculate the appropriate sample size with data augmentation.
Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**Grayscale** (contrast_factor: float = 0.3, brightness_factor: float = 0.3, mode: str = 'mix', invert: bool = False, invert_p: float = 0.0, p: float = 0.5, additional_targets: Optional[dict] = None)

Grayscale intensity augmentation, adapted from ELEKTRONN (http://elektronn.org/).
Randomly adjust contrast/brightness, randomly invert the color space and apply gamma correction. This augmentation is only applied to images.

Parameters:

- **contrast_factor** (*float*) – intensity of contrast change. Default: 0.3
- **brightness_factor** (*float*) – intensity of brightness change. Default: 0.3
- **mode** (*string*) – one of `'2D'`, `'3D'` or `'mix'`. Default: `'mix'`
- **invert** (*bool*) – whether to invert the images. Default: False
- **invert_p** (*float*) – probability of inverting the images. Default: 0.0
- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params ()

Calculate the appropriate sample size with data augmentation.
Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**MisAlignment** (displacement: int = 16, rotate_ratio: float = 0.0, p: float = 0.5, additional_targets: Optional[dict] = None)

Mis-alignment data augmentation of image stacks. This augmentation is applied to both images and masks.

Parameters:

- **displacement** (*int*) – maximum pixel displacement in *xy*-plane. Default: 16
- **rotate_ratio** (*float*) – ratio of rotation-based mis-alignment. Default: 0.0
- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params ()

Calculate the appropriate sample size with data augmentation.
Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**MissingParts** (deformation_strength: int = 0, iterations: int = 40, p: float = 0.5, additional_targets: Optional[dict] = None)

Missing-parts augmentation of image stacks. This augmentation is only applied to images.

Parameters:

- **deformation_strength** (*int*) – Default: 0
- **iterations** (*int*) – Default: 40
- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params ()

Calculate the appropriate sample size with data augmentation.
Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**MissingSection** (num_sections: int = 2, p: float = 0.5, additional_targets: Optional[dict] = None)

Missing-section augmentation of image stacks. This augmentation is applied to both images and masks.

Parameters:

- **num_sections** (*int*) – number of missing sections. Default: 2
- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params ()
Calculate the appropriate sample size with data augmentation.
Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**MixupAugmentor** (min_ratio: float = 0.7, max_ratio: float = 0.9, num_aug: int = 2)
Mixup augmentor (experimental). Conduct linear interpolation between two image samples. The segmentation mask of the sample with higher weight should be used with the augmented output.
The input can be a *numpy.ndarray* or *torch.Tensor* of shape (B, C, Z, Y, X).

Parameters:
- **min_ratio** (*float*) – minimal interpolation ratio of the target volume. Default: 0.7
- **max_ratio** (*float*) – maximal interpolation ratio of the target volume. Default: 0.9
- **num_aug** (*int*) – number of volumes to be augmented in a batch. Default: 2

Examples::

```
>>> from connectomics.data.augmentation import MixupAugmentor
>>> mixup_augmentor = MixupAugmentor(num_aug=2)
>>> volume = mixup_augmentor(volume)
>>> pred = model(volume)
```

class connectomics.data.augmentation.**MotionBlur** (sections: int = 2, kernel_size: int = 11, p: float = 0.5, additional_targets: Optional[dict] = None)
Motion blur data augmentation of image stacks. This augmentation is only applied to images.

Parameters:
- **sections** (*int*) – number of sections along z dimension to apply motion blur. Default: 2
- **kernel_size** (*int*) – kernel size for motion blur. Default: 11
- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params ()
Calculate the appropriate sample size with data augmentation.
Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**Rescale** (low: float = 0.8, high: float = 1.25, fix_aspect: bool = False, p: float = 0.5, additional_targets: Optional[dict] = None)
Rescale augmentation. This augmentation is applied to both images and masks.

Parameters:
- **low** (*float*) – lower bound of the random scale factor. Default: 0.8
- **high** (*float*) – higher bound of the random scale factor. Default: 1.2
- **fix_aspect** (*bool*) – fix aspect ratio or not. Default: False
- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params ()
Calculate the appropriate sample size with data augmentation.
Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**Rotate** (rot90: bool = True, p: float = 0.5, additional_targets: Optional[dict] = None)

Continuous rotatation of the *xy*-plane.
If the rotation degree is arbitrary, the sample size for *x*- and *y*-axes should be at least $\sqrt{2}$ times larger than the input size to ensure there is no non-valid region after center-crop. This augmentation is applied to both images and masks.

Parameters:
- **rot90** (*bool*) – rotate the sample by only 90 degrees. Default: True
- **p** (*float*) – probability of applying the augmentation. Default: 0.5
- **additional_targets** (*dict, optional*) – additional targets to augment. Default: None

set_params ()
Calculate the appropriate sample size with data augmentation.
Some data augmentations (wrap, misalignment, etc.) require a larger sample size than the original, depending on the augmentation parameters that are randomly chosen. This function takes the data augmentation parameters and returns an updated data sampling size accordingly.

class connectomics.data.augmentation.**TestAugmentor** (mode='mean', num_aug=4)
Test-time augmentor.
Our test-time augmentation includes horizontal/vertical flips over the *xy*-plane, swap of *x* and *y* axes, and flip in *z*-dimension, resulting in 16 variants. Considering inference efficiency, we also provide the option to apply only *x*-*y* swap and *z*-flip, resulting in 4 variants. By default the test-time augmentor returns the pixel-wise mean value of the predictions.

Parameters:
- **mode** (*str*) – one of `'min'`, `'max'` or `'mean'`. Default: `'mean'`
- **num_aug** (*int*) – number of data augmentation variants: 0, 4 or 16. Default: 4

Examples::

```
>>> from connectomics.data.augmentation import TestAugmentor
>>> test_augmentor = TestAugmentor(mode='mean', num_aug=16)
>>> output = test_augmentor(model, inputs) # output is a numpy.ndarray on CPU
```

connectomics.engine

class connectomics.engine.**Trainer** (cfg, device, mode, checkpoint=None)

Parameters:
- **cfg** (*yacs.config.CfgNode*) – YACS configuration options.
- **device** (*torch.device*) – by default all training and inference are conducted on GPUs.
- **mode** (*str*) – running mode of the trainer (`'train'` or `'test'`).
- **checkpoint** (*optional*) – the checkpoint file to be loaded (default: *None*)

test ()
Inference function.

train ()
Training function.

connectomics.model

Building Blocks

Model Zoo

connectomics.utils

Post-processing

`connectomics.utils.processing.`**`bc_connected`** (volume, thres1=0.8, thres2=0.5, thres_small=128, scale_factors=(1.0, 1.0, 1.0), dilation_struct=(1, 5, 5), remove_small_mode='background')

From binary foreground probability map and instance contours to instance masks via connected-component labeling.

Note

The instance contour provides additional supervision to distinguish closely touching objects. However, the decoding algorithm only keep the intersection of foreground and non-contour regions, which will systematically result in imcomplete instance masks. Therefore we apply morphological dilation (check **`dilation_struct`**) to enlarge the object masks.

Parameters:

- **volume** (*numpy.ndarray*) – foreground and contour probability of shape `(C, Z, Y, X)`.
- **thres1** (*float*) – threshold of foreground. Default: 0.8
- **thres2** (*float*) – threshold of instance contours. Default: 0.5
- **thres_small** (*int*) – size threshold of small objects to remove. Default: 128
- **scale_factors** (*tuple*) – scale factors for resizing in `(Z, Y, X)` order. Default: `(1.0, 1.0, 1.0)`
- **dilation_struct** (*tuple*) – the shape of the structure for morphological dilation. Default: `(1, 5, 5)`
- **remove_small_mode** (*str*) – `'background'` or `'neighbor'`. Default: `'background'`

`connectomics.utils.processing.`**`bc_watershed`** (volume, thres1=0.9, thres2=0.8, thres3=0.85, thres_small=128, scale_factors=(1.0, 1.0, 1.0), remove_small_mode='background')

From binary foreground probability map and instance contours to instance masks via watershed segmentation algorithm.

Parameters:

- **volume** (*numpy.ndarray*) – foreground and contour probability of shape `(C, Z, Y, X)`.
- **thres1** (*float*) – threshold of seeds. Default: 0.9
- **thres2** (*float*) – threshold of instance contours. Default: 0.8
- **thres3** (*float*) – threshold of foreground. Default: 0.85
- **thres_small** (*int*) – size threshold of small objects to remove. Default: 128
- **scale_factors** (*tuple*) – scale factors for resizing in `(Z, Y, X)` order. Default: `(1.0, 1.0, 1.0)`
- **remove_small_mode** (*str*) – `'background'` or `'neighbor'`. Default: `'background'`

`connectomics.utils.processing.`**`binary_connected`** (volume, thres=0.8, thres_small=128, scale_factors=(1.0, 1.0, 1.0), remove_small_mode='background')

From binary foreground probability map to instance masks via connected-component labeling.

Parameters:

- **volume** (*numpy.ndarray*) – foreground probability of shape (C, Z, Y, X).
- **thres** (*float*) – threshold of foreground. Default: 0.8
- **thres_small** (*int*) – size threshold of small objects to remove. Default: 128
- **scale_factors** (*tuple*) – scale factors for resizing in (Z, Y, X) order. Default: $(1.0, 1.0, 1.0)$
- **remove_small_mode** (*str*) – `'background'` or `'neighbor'`. Default: `'background'`

`connectomics.utils.processing.`**`binary_watershed`** (volume, thres1=0.98, thres2=0.85, thres_small=128, scale_factors=(1.0, 1.0, 1.0), remove_small_mode='background')

From binary foreground probability map to instance masks via watershed segmentation algorithm.

Parameters:

- **volume** (*numpy.ndarray*) – foreground probability of shape (C, Z, Y, X).
- **thres1** (*float*) – threshold of seeds. Default: 0.98
- **thres2** (*float*) – threshold of foreground. Default: 0.85
- **thres_small** (*int*) – size threshold of small objects to remove. Default: 128
- **scale_factors** (*tuple*) – scale factors for resizing in (Z, Y, X) order. Default: $(1.0, 1.0, 1.0)$
- **remove_small_mode** (*str*) – `'background'` or `'neighbor'`. Default: `'background'`

`connectomics.utils.processing.`**`polarity2instance`** (volume, thres=0.5, thres_small=128, scale_factors=(1.0, 1.0, 1.0), semantic=False)

From synaptic polarity prediction to instance masks via connected-component labeling. The input volume should be a 3-channel probability map of shape (C, Z, Y, X) where $C = 3$, representing pre-synaptic region, post-synaptic region and their union, respectively.

Note

For each pair of pre- and post-synaptic segmentation, the decoding function will annotate pre-synaptic region as $2n - 1$ and post-synaptic region as $2n$, for $n > 0$. If **`semantic=True`**, all pre-synaptic pixels are labeled with while all post-synaptic pixels are labeled with 2. Both kinds of annotation are compatible with the `TARGET_OPT: ['1']` configuration in training.

Note

The number of pre- and post-synaptic segments will be reported when setting **`semantic=False`**. Note that the numbers can be different due to either incomplete syanpses touching the volume borders, or errors in the prediction. We thus make a conservative estimate of the total number of synapses by using the relatively small number among the two.

Parameters:

- **volume** (*numpy.ndarray*) – 3-channel probability map of shape $(3, Z, Y, X)$.
- **thres** (*float*) – probability threshold of foreground. Default: 0.5
- **thres_small** (*int*) – size threshold of small objects to remove. Default: 128
- **scale_factors** (*tuple*) – scale factors for resizing the output volume in (Z, Y, X) order. Default: $(1.0, 1.0, 1.0)$
- **semantic** (*bool*) – return only the semantic mask of pre- and post-synaptic regions. Default: False

Examples::

```
>>> from connectomics.data.utils import readvol, savevol
>>> from connectomics.utils.processing import polarity2instance
>>> volume = readvol(input_name)
>>> instances = polarity2instance(volume)
>>> savevol(output_name, instances)
```

Indices and Tables

- **genindex**
- **modindex**

Index

V

Python Module Index

c