omnipose

Release 1.0.6-26-g260e4d3

Kevin Cutler

May 10, 2024

BASICS

Omnipose is a general image segmentation tool that builds on Cellpose in a number of ways described in our paper. It works for both 2D and 3D images and on any imaging modality or cell shape, so long as you train it on representative images. We have several pre-trained models for:

- **bacterial phase contrast**: trained on a diverse range of bacterial species and morphologies.
- **bacterial fluorescence**: trained on the subset of the phase data that had a membrane or cytosol tag.
- **C. elegans**: trained on a couple OpenWorm videos and the BBBC010 alive/dead assay. We are working on expanding this significantly with the help of other labs contributing ground-truth data.
- **cyto2**: trained on user data submitted through the Cellpose GUI. Very diverse data, but not necessarily the best quality. This model can be a good starting point for users making their own ground-truth datasets.

Here we provide both the documentation for Omnipose and our fork of Cellpose. Please note this documentation is actively in development. For support, submit an issue on the Omnipose repo. For more on the workings of cellpose, check out our twitter thread and read the paper.

1. Install an Anaconda distribution of Python. Note you might need to use an anaconda prompt if you did not add anaconda to the path. Alternatives like miniconda also work just as well.
2. Open an anaconda prompt / command prompt with `conda` for **python 3** in the path.
3. To create a new environment for CPU only, run

```
conda create -n omnipose 'python==3.10.12' pytorch
```

For users with NVIDIA GPUs, add these additional arguments:

```
torchvision pytorch-cuda=11.8 -c pytorch -c nvidia
```

See *GPU support* for more details. Python 3.10 is not a strict requirement; see *Python compatibility* for more about choosing your python version.

4. To activate this new environment, run

```
conda activate omnipose
```

5. To install the latest PyPi release of Omnipose, run

```
pip install omnipose
```

or, for the most up-to-date development version,

```
git clone https://github.com/kevinjohncutler/omnipose.git
cd omnipose
pip install -e .
```

Warning: If you previously installed Omnipose, please run

```
pip uninstall cellpose_omni && pip cache remove cellpose_omni
```

to prevent version conflicts. See *project structure* for more details.

CHAPTER

ONE

PYTHON COMPATIBILITY

We have tested Omnipose extensively on Python version 3.8.5 and have encountered issues on some lower versions. Versions up to 3.10.11 have been confirmed compatible, but we have encountered bugs with the GUI dependencies on 3.11+. For those users with system or global pyenv python3 installations, check your python version by running `python -V` before making your conda environment and choose a different version. That way, there is no crosstalk between pip-installed packages inside and outside your environment. So if you have 3.x.y installed via pyenv etc., install your environment with 3.x.z instead.

CHAPTER

TWO

PYENV VERSUS CONDA

Pyenv also works great for creating an environment for installing Omnipose (and it also works a lot better for installing Napari alongside it, in my experience). Simply set your global version anywhere from 3.8.5-3.10.11 and run `pip install omnipose`. I've had no problems with GPU compatibility with this method on Linux, as pip collects all the required packages. Conda is much more reproducible, but often finicky. You can use pyenv on Windows and macOS too, but you will need a conda environment for Apple Silicon GPU support (PyPi still lacks many package versions built for Apple Silicon).

CHAPTER

THREE

GPU SUPPORT

Omnipose runs on CPU on macOS, Windows, and Linux. PyTorch has historically only supported NVIDIA GPUs, but has more more recently begun supporting Apple Silicon GPUs. It looks AMD support may be avaiable these days (ROCm), but I have not tested that out. Windows and Linux installs are straightforward:

Your PyTorch version (>=1.6) needs to be compatible with your NVIDIA driver. Older cards may not be supported by the latest drivers and thus not supported by the latest PyTorch version. See the official documentation on installing both the most recent and previous combinations of CUDA and PyTorch to suit your needs. Accordingly, you can get started with CUDA 11.8 by making the following environment:

```
conda create -n omnipose 'python==3.10.12' pytorch torchvision pytorch-cuda=11.8 \
-c pytorch -c nvidia
```

Note that the official PyTorch command includes torchaudio, but that is not needed for Omnipose. (*torchvision appears to be necessary these days*). If you are on older drivers, you can get started with an older version of CUDA, *e.g.* 10.2:

```
conda create -n omnipose pytorch=1.8.2 cudatoolkit=10.2 -c pytorch-lts
```

For Apple Silicon, download omnipose_mac_environment.yml and install the environment:

```
conda env create -f <path_to_environment_file>
conda activate omnipose
```

You may edit this yml to change the name or python version etc. For more notes on Apple Silicon development, see this thread. On all systems, remember that you may need to use ipykernel to use the omnipose environment in a notebook.

CHAPTER

FOUR

WHERE ARE MODELS STORED?

To maintain compatibility with Cellpose, the pretrained Omnipose models are also downloaded to `$HOME/.cellpose/models/`. This path on linux is `/home/USERNAME/.cellpose/`, on macOS `/Users/USERNAME/.cellpose/`, and on Windows `C:\Users\USERNAME\.cellpose\models\`. These models are downloaded the first time you try to use them, either on the command line, in the GUI, or in a notebook.

If you would like to download the models to a different directory and are using the command line or the GUI, you will need to always set the environment variable `CELLPOSE_LOCAL_MODELS_PATH` before you run `python -m omnipose ...` (thanks Chris Roat for implementing this!).

To set the environment variable in the command line/Anaconda prompt on windows run the following command modified for your path: `set CELLPOSE_LOCAL_MODELS_PATH=C:/PATH_FOR_MODELS/`. To set the environment variable in the command line on linux, run `export CELLPOSE_LOCAL_MODELS_PATH=/PATH_FOR_MODELS/`.

To set this environment variable when running Omnipose in a jupyter notebook, run this code at the beginning of your notebook before you import Omnipose:

```
import os
os.environ["CELLPOSE_LOCAL_MODELS_PATH"] = "/PATH_FOR_MODELS/"
```

CHAPTER

FIVE

COMMON ISSUES

If you receive the error: `Illegal instruction (core dumped)`, then likely mxnet does not recognize your MKL version. Please uninstall and reinstall mxnet without mkl:

```
pip uninstall mxnet-mkl
pip uninstall mxnet
pip install mxnet==1.4.0
```

If you receive the error: `No module named PyQt5.sip`, then try uninstalling and reinstalling pyqt5

```
pip uninstall pyqt5 pyqt5-tools
pip install pyqt5 pyqt5-tools pyqt5.sip
```

If you have errors related to OpenMP and libiomp5, then try

```
conda install nomkl
```

If you receive an error associated with **matplotlib**, try upgrading it:

```
pip install matplotlib --upgrade
```

If you receive the error: `ImportError: _arpack DLL load failed`, then try uninstalling and reinstalling scipy

```
pip uninstall scipy
pip install scipy
```

If you are having issues with the graphical interface, make sure you have **python 3.8.5** installed. Higher versions *should* also work.

If you are on macOS Yosemite or earlier, PyQt does not work and you won't be able to use the GUI. More recent versions of macOS are fine. The software has been heavily tested on Windows 10 and Ubuntu 18.04, and less well tested on macOS. Please post an issue if you have installation problems.

CHAPTER

SIX

GUI

The Omnipose GUI is an expansion and refinement of that from Cellpose. It defaults to the `bact_phase_omni` model and corresponding model parameters. Additionally, we pre-load a small bacterial phase contrast image for demonstration purposes. Masks are also represented in *N-color* format by default, which is handy for visualizing and editing. Be sure to untick the `ncolor` box to switch to standard label format before saving your masks if that format is what you need (what you see is what you get).

Note: The GUI only segments one image at a time, so it is really only intended for users to try out Omnipose and find the best model and optimal segmentation parameters with minimal setup. If you want to segment multiple images in a directory or train a model, use Omnipose in the *command line* or a *jupyter notebook*. The GUI prints out the current parameters for you in the bottom left.

6.1 Starting the GUI

The quickest way to start is to open the GUI from a command line terminal. You might need to open an anaconda prompt if you did not add anaconda to the path. Activate your omnipose conda environment and run `omnipose` (or `python -m omnipose`).

The first time Omnipose runs, it will ask you to download the GUI dependencies. When it finishes, run the launch command again. The terminal will remain open and you can see model download progress, error messages, etc. as you interact with the GUI.

You can **drag and drop** images (.tif, .png, .jpg, .gif) into the GUI and run Cellpose, and/or manually segment them. Omnipose waits to download a model until the first time you use it. When the GUI is processing, you will see the progress bar fill up and during this time you cannot click on anything in the GUI. For more information about what the GUI is doing you can look at the terminal/prompt with which you launched the GUI. For best accuracy and runtime performance, resize images so cells are less than 100 pixels across.

For multi-channel, multi-Z tiffs, the expected format is ZCYX.

6.2 Using the GUI

Main GUI mouse controls (works in all views):

- Pan = left-click + drag
- Zoom = scroll wheel (or +/= and - buttons)
- Full view = double left-click
- Select mask = left-click on mask
- Delete mask = Ctrl (or Command on Mac) + left-click
- Merge masks = Alt + left-click (will merge last two)
- Start draw mask = right-click
- End draw mask = right-click, or return to circle at beginning

Overlaps in masks are NOT allowed. If you draw a mask on top of another mask, it is cropped so that it doesn't overlap with the old mask. Masks in 2D should be single strokes (if *single_stroke* is checked).

If you want to draw masks in 3D, then you can turn *single_stroke* option off and draw a stroke on each plane with the cell and then press ENTER. 3D labeling will fill in unlabelled z-planes so that you do not have to as densely label.

Note: The GUI automatically saves after you draw a mask but NOT after segmentation and NOT after 3D mask drawing (too slow). Save in the file menu or with Ctrl+S. The output file is in the same folder as the loaded image with `_seg.npy` appended.

Keyboard shortcuts	Description
CTRL+H	help
=/+ // -	zoom in // zoom out
CTRL+Z	undo previously drawn mask/stroke
CTRL+0	clear all masks
CTRL+L	load image (can alternatively drag and drop image)
CTRL+S	SAVE MASKS IN IMAGE to `_seg.npy` file
CTRL+P	load `_seg.npy` file (note: it will load automatically with image if it exists)
CTRL+M	load masks file (must be same size as image with 0 for NO mask, and 1,2,3... for masks)
CTRL+N	load numpy stack (NOT WORKING ATM)
A/D or LEFT/RIGHT	cycle through images in current directory
W/S or UP/DOWN	change color (RGB/gray/red/green/blue)
PAGE-UP / PAGE-DOWN	change to flows and cell prob views (if segmentation computed)
, / .	increase / decrease brush size for drawing masks
X	turn masks ON or OFF
Z	toggle outlines ON or OFF
C	cycle through labels for image type (saved to `_seg.npy`)

6.3 Segmentation options

`SIZE`: you can manually enter the approximate diameter for your cells, or press "calibrate" to let the SizeModel() estimate it. The size can be visualized by a disk at the bottom of the view window (can turn this disk on by checking "scale disk on"). Size defaults to 0 for bacterial models, which disables image resizing.

`use GPU`: this will be grayed out for conda envoronemts / machines not configured for running pytorch on GPU.

`MODEL`: choose among several pretrained models

`CHAN TO SEG`: this is the channel in which the cytoplasm or nuclei exist

`CHAN2 (OPT)`: if *cyto** model is chosen, then choose the nuclear channel for this option

CHAPTER

SEVEN

INPUTS

Omnipose automatically detects TIFs, PNGs, or JPEGs. Under the hood, *cellpose_omni.io* uses `tifffile` for loading TIFs and cv2 for PNG and JPEG. We are considering adding direct support for other bioformats types such as ND2, but for now all input must be exported to the above image formats prior to running Omnipose.

7.1 Channel formatting

Single-plane, multichannel images can be formatted as (**nY**,**nX**,**nChan**) or (**nChan**,**nY**,**nX**), the latter CYX formatting being more conventional and easier to work with (*e.g.*, in Napari). The *channels* settings will take care of reshaping the input appropriately for the network if we can safely assume that the smallest axis is the channel axis. For example, a (2,2048,2048) image will automatically have axis 0 set to be the channel axis. The *channel_axis* parameter allows you to override this when necessary.

Note that Omnipose also rescales the input for each channel so that 0 = 0.01st percentile of image values and 1 = 99.99th percentile. These are not yet user-tunable parameters, but they will be in a future release.

7.2 3D segmentation

Multiple-plane and multiple-channel TIFs are supported in the GUI (can drag-and-drop) and are supported when running in a notebook. Multiplane images should be of shape ZCYX or ZYX. You can test this by running in python:

```
import skimage.io
data = skimage.io.imread('img.tif')
print(data.shape)
```

If drag-and-drop of the TIF into the GUI does not work correctly, then it's likely that the shape of the TIF is incorrect. If drag-and-drop works (you can see a TIF with multiple planes), then the GUI will automatically run 3D segmentation and display it in the GUI. Watch the command line for progress. It is recommended to use a GPU to speed up processing.

If drag-and-drop doesn't work because of the shape of your TIF, you need to transpose the TIF and re-save to use the GUI, or use the Napari plugin for Cellpose, or run CLI/notebook and specify the `channel_axis` and/or `z_axis` parameters:

> `channel_axis` and `z_axis` can be used to specify the axis (0-based) of the image which corresponds to the image channels and to the z axis. For example. a 105-plane z-stack image with 2 channels of shape (1024, 1024,2,105,1) can be specified with **`channel_axis`**=2 and **`z_axis`**=3. If **`channel_axis`**=None, cellpose will try to automatically determine the channel axis by choosing the dimension with the minimal size after squeezing. If **`z_axis`**=None cellpose will automatically select the first non-channel axis of the image to be the Z axis (ZYX ordering). These parameters can be specified using the command line with `--`**`channel_axis`** or `--`**`z_axis`** or as inputs to `model.eval` for the `Cellpose` or `CellposeModel` model.

There are two distinct modes of 3D image processing. The first is Cellpose3D, which uses a 2D model on orthogonal slices of the volume to estimate 3D predicitons from 2D network output. To use this in a notebook, set **do_3D**=True. You can give a list of 3D inputs, or a single 3D/4D stack. When running on the command line, add the flag --**do_3D** (it will run all TIFs in the folder as 3D TIFs if possible).

If Cellpose3D segmentation is not working well and there is inhomogeneity in Z, try stitching masks in Z instead of running **do_3D**=True. See details for this option here: stitch_threshold.

The second approach, implemented in Omnipose, is to directly predict 3D flows etc. by training models on 3D datasets. We offer one pretrained model: `plant_omni`. The --**dim** argument allows users to specify the dimensionality of their data/model for training and evaluation, so **dim**=2 corresponds to 2D processing (even in Cellpose3D) and **dim**=3 corresponds to 3D processing. More work is needed to validate functionality of true 3D segmentation in the GUI.

CHAPTER

EIGHT

SETTINGS

The most important settings are described on this page. See *cellpose_omni.models()* for all options.

This is a typical example of using an Omnipose model to segment a list of images in a notebook. Cellpose users need only select an Omnipose model and use `omni=True` to update their existing code.

```
from cellpose_omni import models
import skimage.io
model = models.Cellpose(gpu=False,
                        model_type='bact_phase_omni',
                        nclasses=4,
                        nchan=2,
                        dim=2)

files = ['img0.tif', 'img1.tif']
imgs = [skimage.io.imread(f) for f in files]
masks, flows, styles, diams = model.eval(imgs,
                                         diameter=None,
                                         channels=[0,0],
                                         threshold=0.4,
                                         omni=True)
```

This example shows the same settings used for each image, but you can also pass in a list for `channels` and `diameter` that specifies unique values to apply to each image. See our *example notebooks* for a solid introduction and figure notebooks for more advanced examples.

Tip: Use **pretrained_model**=<**path to model**> in place of **model_type**=<**model name**> when you want to use a model that is not built-in. Specify **nclasses** and **nchan** if you encounter any issues in the model initialization (see *Pretrained models*).

8.1 Channels

Use `channels = [0,0]` for mono-channel images or multi-channel images that you would like converted to grayscale prior to segmentation. `[0,0]` is what we used to train and evaluate our `bact_phase_omni`, `bact_fluor_omni`, `worm_omni`, `worm_high_res_omni`, and `plant_omni` models. If you do want to run segmentation on a specific channel of multi-channel images, use *1-based-indexing* `[i,0]` with `i = 1,2,3,...` for red, green, blue, ..., respectively. For example, you might have blue nuclei that look a lot like fluorescent bacteria, so could use the `bact_fluor_omni` model with `channels = [2,0]`.

You can also use two channels for segmentation: a cytoplasm channel and a nuclear channel. The `cyto2_omni` model was trained with image channels re-ordered to have red cytoplasm and green nucleus (where applicable in the dataset) using `--chan 1 --chan2 2` and therefore was evaluated using `channels = [1,2]`.

See *mono_channel_bact.ipynb* for a monochannel segmentation on bacterial phase contrast images and *multi_channel_cyto.ipynb* for multichannel segmentation of mouse neuron cells.

8.2 Flow threshold

The neural network may predict hallucinate network outputs that do not correspond well to the masks found by the mask reconstruction dynamics. As a consistency check, we can compute the 'true' flow field from the predicted labels and compare this to the network predictions pixel-by-pixel. The `flow_threshold` parameter is the maximum allowed error of the flows averaged over all pixels in a given mask. The default is `flow_threshold=0.4`. Increase this threshold if Omnipose is not returning as many masks as you expect. Decrease this threshold if Omnipose is returning too many spurious masks.

Note: Well-trained models really don't need this and we set `flow_threshold=0.0` for most of our model evaluation. This disables the flow error calculation and will make Omnipose run a lot faster on large datasets.

8.3 Mask threshold

This threshold is applied to the distance transform output of Omnipose (or the cellprob output of Cellpose) to seed cell masks pixels for running dynamics. The default is `mask_threshold=0.0`. Decrease this threshold if you are getting too few masks or if masks do not cover the entire cell.

Tip: The GUI provides sliders that update the Omnipose output for `flow_threshold` and `mask_threshold` in real time, which is very fast even on CPU for small images (~500 x 500 px).

8.4 Diameter

In most Omnipose models, we set `diameter=0` to disable image rescaling. We found that rescaling to a common cell diameter is only necessary when the images for training and evaluation have extreme diffrences in cell size, such as in the cyto2 dataset. Therefore, `cyto2_omni` was trained with a mean diameter of 30px just like the Cellpose `cyto` model. This means that images are rescaled by a factor of `30.0/D` where D is the mean diameter of all cells in the image. See the page on mean cell diameter to see how Omnipose handles this better than Cellpose.

The `worm_high_res_omni` is another example where rescaling was necessary. We suspect that it is the network architecture kernel size and number of down-sampling stages that prevents accurate prediction of boundary-derived output like flow and distance at the centers of objects. For these high-resolution *C. elegans* images, we found 60px to work well, but we did not do more tests to push this higher. To use this model, images should be rescaled by a factor of `60.0/D`.

Tip: At this time, the diameter used for training is not saved with the model parameters and therefore must be specified using `mymodel.diameter=60.0` after initializing `mymodel=models.CellposeModel()`. 30 is the default for models with *cyto* in the name but can be overwritten as shown. Similarly, *nuclei*-named models default to a mean diameter of 17 and *bacteria*-named models default to a mean diameter of 0 (rescaling disabled).

8.5 SizeModel()

In contrast to the `CellposeModel()` class that takes `diameter` as an option for rescaling, the `Cellpose` class includes a `SizeModel()` for automatic diameter estimation. This is a linear regression model trained on the 'style' vector of the network, which you can think of as a 64-dimensional summary of the input image. A SizeModel() for Omnipose was trained on the `cyto2` dataset to predict our own cell diameter from the style vector. To use the `SizeModel()`, we follow a two-step process:

1. Run the image through the cellpose network and obtain the style vector. Predict the size using the linear regression model from the style vector.
2. Resize the image based on the predicted size and run cellpose again, and produce masks. Take the final estimated size as the median diameter of the predicted masks.

For automated estimation in the `Cellpose()` class set `diameter = None` (default). However, if this estimate is incorrect, you will need to set the diameter manually.

Changing the diameter will change the results that the algorithm outputs. When the diameter is set smaller than the true size then Omnipose may over-segment cells. Similarly, if the diameter is set too big then Omnipose may under-segment cells.

8.6 Resample

The cellpose network is run on your rescaled image -- where the rescaling factor is determined by the diameter you input (or determined automatically as above). For instance, if you have an image with 60 pixel diameter cells, the rescaling factor is 30./60. = 0.5. After network predictions are made, the model runs the dynamics. The dynamics can be run at the rescaled size (`resample=False`), or the dynamics can be run on the resampled, interpolated flows at the true image size (`resample=True`). `resample=True` will create smoother masks when the cells are large but will be slower. `resample=False` can produce some jagged mask edges due to nearest-neighbor interpolation. The default our Cellpose fork is `resample==True`.

8.7 3D settings

Volumetric stacks do not always have the same sampling in XY as they do in Z. Therefore you can set an `anisotropy` parameter to allow for differences in sampling, e.g. set to 2.0 if Z is sampled half as dense as X or Y.

There may be additional differences in YZ and XZ slices that make them unable to be used for 3D segmentation. I'd recommend viewing the volume in those dimensions if the segmentation is failing. In those instances, you may want to turn off 3D segmentation (`do_3D=False`) and run instead with `stitch_threshold>0`. Cellpose will create masks in 2D on each XY slice and then stitch them across slices if the IoU between the mask on the current slice and the next slice is greater than or equal to the `stitch_threshold`.

3D segmentation ignores the `flow_threshold` because we did not find that it helped to filter out false positives in our test 3D cell volume. Instead, we found that setting `min_size` is a good way to remove false positives.

CHAPTER

NINE

OUTPUTS

Omnipose uses a generalized version of the Cellpose U-net to predict several output "images" based on an input image. You can use a Cellpose model with Omnipose (**`omni`**=`True`), which just turns on the Omnipose mask reconstruction algorithm to fix the over-segmentation errors that may result form your Cellpose network outputs.

Cellpose models predict 2 outputs: flows and cell probability (cellprob). The predictions the network makes of cellprob are the inputs to a sigmoid centered at zero ($\sigma(x) = \frac{1}{1+e^{-x}}$), so they vary from around -6 to $+6$. The flow field is a vector field and is therefore comprised of N distinct outputs in N dimensions.

The original Omnipose models predict 3 outputs: distance field, flow field, and boundary. The distance field is modified during training to have a background of -5 instead of 0. This helps balance the asymmetry in output range, as the flow components range from -5 to -5 and the boundary field ranges from roughly -6 to $+6$. (same sigmoid input described above).

New Omnipose models no longer require the boundary field to achieve the same accuracy, and thus by default train with just distance and flow (**`nclasses`**=2).

Warning: If you trained a custom model with Omnipose <= version 0.4.0, your defaults were **`nclasses`**=3 and **`nchan`**=2. Use these settings when initializing you model. Moving forward, Omnipose will use **`nclasses`**=2 and **`nchan`**=1 by default. See *Pretrained models* for a table of models and the number of outputs.

9.1 _seg.npy output

`*_seg.npy` files have the following fields:

- *filename* : filename of image
- *img* : image with chosen channels (CYX) (if not multiplane)
- *masks* : masks (0 = NO masks; 1,2,... = mask labels)
- *colors* : colors for masks
- *outlines* : outlines of masks (0 = NO outline; 1,2,... = outline labels)
- *chan_choose* : channels that you chose in GUI (0=gray/none, 1=red, 2=green, 3=blue)
- *ismanual* : element *k* = whether or not mask *k* was manually drawn or computed by Omnipose/Cellpose
- ***flows***
  [flows[0] is XY flow in RGB, flows[1] is the cell probability in range 0-255 instead of 0.0 to 1.0, flows[2] is Z flow in range 0-255 (if it exists, otherwise zeros),] flows[3] is [dY, dX, cellprob] (or [dZ, dY, dX, cellprob] for 3D), flows[4] is pixel destinations (for internal use)
- *est_diam* : estimated diameter (if run on command line)

- *zdraw* : for each mask, which planes were manually labelled (planes in between manually drawn have interpolated masks)

Here is an example of loading in a *_seg.npy file and plotting masks and outlines

```
import numpy as np
from cellpose_omni import plot
dat = np.load('_seg.npy', allow_pickle=True).item()

# plot image with masks overlaid
mask_RGB = plot.mask_overlay(dat['img'], dat['masks'],
                         colors=np.array(dat['colors']))

# plot image with outlines overlaid in red
outlines = plot.outlines_list(dat['masks'])
plt.imshow(dat['img'])
for o in outlines:
    plt.plot(o[:,0], o[:,1], color='r')
```

If you run in a notebook and want to save to a *_seg.npy file, run

```
from cellpose_omni import io
io.masks_flows_to_seg(images, masks, flows, diams, file_name, channels)
```

where each of these inputs is a list (as is the output of *model.eval*)

9.2 PNG output

You can save masks to PNG in the GUI. Be aware that the GUI will save the masks in the format being displayed, which defaults to the N-color representation for easier visualization and editing (4 or 5 repeating colors). Toggle off `ncolor` before saving masks to put them in standard 1,...,N format.

To save masks (and other plots in PNG) using the command line, add the flag `--save_png`. If you want the N-color versions saved, use `--save_ncolor`.

In a notebook, use:

```
from cellpose_omni import io
io.save_to_png(images, masks, flows, image_names)
```

9.3 ROI manager compatible output for ImageJ

You can save the outlines of masks in a text file that is compatible with ImageJ ROI Manager from the GUI File menu.

To save using the command line, add the flag `--save_txt`.

Use the function below if running in a notebook:

```
from cellpose_omni import io, plot

# image_name is file name of image
# masks is numpy array of masks for image
base = os.path.splitext(image_name)[0]
```

(continues on next page)

(continued from previous page)

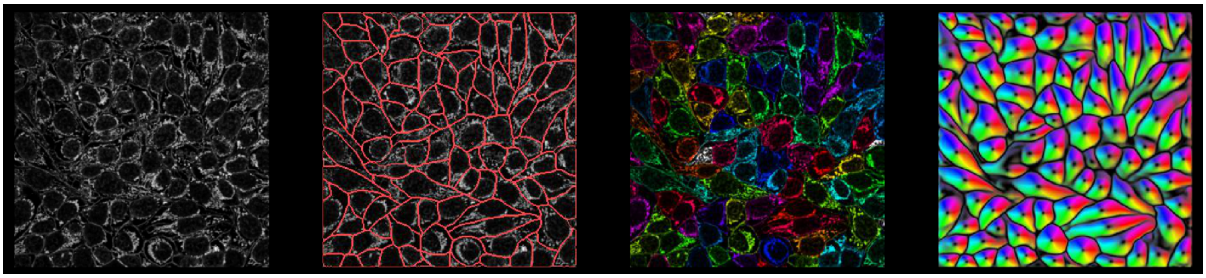
```
outlines = utils.outlines_list(masks)
io.outlines_to_text(base, outlines)
```

To load this `_cp_outlines.txt` file into ImageJ, use the python script provided in Cellpose: `imagej_roi_converter.py`. Run this as a macro after opening your image file. It will ask you to input the path to the `_cp_outlines.txt` file. Input that and the ROIs will appear in the ROI manager.

9.4 Plotting functions

In `plot.py` there are functions, like `show_segmentation`:

```
from cellpose_omni import plot

nimg = len(imgs)
for idx in range(nimg):
    maski = masks[idx]
    flowi = flows[idx][0]

    fig = plt.figure(figsize=(12,5))
    plot.show_segmentation(fig, imgs[idx], maski, flowi, channels=channels[idx])
    plt.tight_layout()
    plt.show()
```



CHAPTER

TEN

TRAINING

Begin a training round in a terminal using the following command template:

```
omnipose --train --use_gpu --dir <training image directory> \
         --img_filter <img_filter> --mask_filter <mask_filter> \
         --nchan <nchan> --all_channels --channel_axis <channel_axis> \
         --pretrained_model None --diameter 0 --nclasses 2 \
         --learning_rate 0.1 --RAdam --batch_size 16 --n_epochs <n_epochs>
```

Note: Training should be done only via CLI. If image preprocessing is required, I highly suggest doing that in a script and saving to a new folder (as opposed to attempting preprocessing + training in one script/notebook).

The main commands here are:

omnipose
: calls `__main__.py` in `cellpose-omni`, which first loads the images in `--dir` and formats them. Then `--train` toggles on the training branch (versus evaluation).

--dir
: points to a folder of image and label pairs. With `--look_one_level_down`, you can let `--dir` point to a folder with subfolders. This can be very useful when training on several distinct subsets of ground truth data.

--diameter
: should be set to `0` (and is now `0` by default) to disable rescaling. Anything else will rescale your images relative to a mean diameter of 30 (see *Cell diameter*), such that `--diameter 15` will **upscale** your image by a factor of 2 along each axis and `--diameter 60` will likewise **downscale** by a factor of 2. If you need automatic diameter estimation, see *Diameter and the Size Model*.

--nchan, --nclasses
: define the number of image channels and the number of prediction classes. These should always be specified for **custom** models, as the defaults are *--nchan 1* (mono-channel images) and `--nclasses 2` (flow and distance field predictions). If you train a model with `--nclasses 3` (add the boundary field) or have multichannel images these will be in the model file name. Use these when running the model, too, both in CLI and in *`cellpose_omni.models.CellposeModel()`*.

--all_channels
: tells Omnipose to use all `nchan` channels for segmentation. The relatively complicated `--chan` and `--chan2` settings from Cellpose are still available, but I never use them. I highly recommend preprocessing your training set to have the channels you want to use (and for evaluation, do the same preprocessing in a script/notebook).

--channel_axis
: lets you specify where your channels are in your arrays. Conventional ordering is CYX for multichannel 2D images, so `--channel_axis` defaults to `0`. RGB images will have `--channel_axis 2`.

Warning: Paths given to `--dir` or `--test_dir` must absolute paths.

10.1 Hyperparameters

It is best for reproducibility to explicitly choose hyperparameters at runtime rather than relying on defaults.

`--RAdam`
: selects the RAdam optimizer (versus the default SGD). I found RAdam to be a bit faster and more stable compared to SGD and other optimizers.

`--learning_rate`
: controls the optimizer step size.

`--batch_size`
: controls the number of images the network sees for each step (with the last batch being smaller if the number of images is not evenly divisible by `batch_size`). A random crop is selected from each image (see `--tyx`). This means that only a portion of each image is seen during a given epoch. Smaller batches can sometimes lead to better generalization. Larger batches can lead to better stability. I have found that it does not make a very large difference in model performance, but larger batches can train faster (see `--dataparallel`).

`--tyx`
: controls the crop size for selecting a sample from each training image (see *Image dimensions*).

`--n_epochs`
: controls how many times the network is shown the full dataset. I usually do 4000.

`--dataloader`
: toggles on parallel dataloading. Preprocessing batches for training is a CPU bottleneck, but the DataParallel library helps a lot with that. Use `--num_workers` to control how many cores will participate. This is only a benefit when you have more images in your training set than cores on your machine.

10.2 Model saving

You can choose how often to save your models with `--save_every <n>`. This overwrites the model every time. To save a new model each `n` epochs, you can use `save_each` (useful for debugging / comparing across epochs).

10.3 Training data

Your training set should consist of at least two tuples of images, labels, and (optionally) label link files.

10.3.1 File naming conventions

Each tuple of images and labels should be formatted as <**base**><**img_filter**>.<**img_ext**>, <**base**><**mask_filter**>.<**mask_ext**>, and (optionally) <**base**>**_links.txt**. **base** can be any string. The **img_filter** defaults to an empty string '' and the **mask_filter** defaults to **_masks**. These can be arranged in a single training folder:

```
folder/
├── A.tif
├── A_masks.tif
├── B.tif
├── B_masks.tif
└── ...
```

Or in subfolders (when using --**look_one_level_down**):

```
folder/
├── subfolder_1/
│   ├── A.tif
│   └── A_masks.tif
├── subfolder_2/
│   ├── B.tif
│   ├── B_masks.tif
│   └── ...
└── ...
```

If you use the --**img_filter** option (--**img_filter img** in this case), the suffix only goes on image files:

```
folder/
├── A_img.tif
├── A_masks.tif
├── B_img.tif
├── B_masks.tif
└── ...
```

10.3.2 File extensions

Microscopy images should generally be saved in a lossless format like PNG or TIF. Instance label matrices may likewise be stored as images in either PNG or TIF. Note that TIF supports up to 32 bits per channel whereas PNG only supports 16. That said, if you have more than $2^{16} - 1 = 65535$ labels in one image, you should definitely be cropping your images into several smaller images.

10.3.3 Image dimensions

You should aim to make training images of roughly size `(512,512)`. During training, the **tyx** parameter (set to `224,224` by default) controls the size of warped image crops in each batch shown to the network. Although the true rectangular patch selected from each image in a batch has randomly expanded or contracted dimensions (within a range `0.5-1.5`), you should aim to have the *tyx* dimensions roughly half that of the images in the training set. If much smaller, then each image will not be sufficiently covered during an epoch (requiring more epochs to converge). Larger `tyx` will just slow down training and possibly hurt generalizability.

If an image dimension is substantially larger than 512 px, subdivide it along that axis. For example, `(2048,2048)` images should be split into 16 `(512,512)` images (4 along each axis). Smaller images are far easier to annotate correctly.

If your image dimensions are substantially smaller than 512 px, you can instead decrease the **`tyx`** parameter. For example, if your training images are around size `(256,256)`, then I would recommend the CLI flag `--tyx 128,128`.

Note: The *tyx* tuple elements must be evenly divisible by 8 (for U-net downsampling).

10.3.4 Object density

As a general rule, you want to train on images with densely packed objects. This is to balance the foreground class to the background class. In other words, we want Omnipose to focus on predicting good output in foreground regions rather than zero output in background regions. If your images have a lot of useless background, *crop out* just the denser regions. This can be done automatically if you can segment clusters/microcolonies of cells. You can use functions in `omnipose.utils` for processing a binary image into crops that you can then join into an ensemble image using a rectangle packing algorithm. Training on these images allows Omnipose to see the same number of cells but a lot faster, as it does not waste time looking at too much background.

10.3.5 Ground truth quality

Garbage in, garbage out. It is better to have fewer images with meticulously crafted, consistent labels than many images with sloppy labels. Your labels should...

1. be based on supplemental channels wherever the primary channel is ambiguous
2. be label matrices, not semantic (binary) masks
3. not miss a single cell
4. extend to cell boundaries
5. meet each other at cell interfaces

You will probably spend 10x more time annotating ground truth images than acquiring them, so it is worth putting in the effort to find a membrane dye that does not conflict with main channel(s) on which your model will be trained. This is purely for the purposes of having a physiological reference for the ground truth of cell extent and cell septation, not for training the segmentation model.

Tip: If using a transmissive modality like phase contrast or brightfield or DIC, use the same filter cube as your fluorescence channel. This usually removes any offset between the channels. Otherwise, be sure to do multimodal registration between the channels.

10.4 Transfer learning

You can use `--pretrained_model None` to train from scratch or `--pretrained_model <model path>` to start from an existing model. Once a model is initialized and trained, you **cannot** change its structure. This is defined by **nchan** (the number of channels used for segmentation), **nclasses** (the number of prediction classes), and **dim** (the dimension of the images). **You must use precisely the same nchan, nclasses, and dim that were used to train the existing model.** See *Models* for a table of the pretrained model parameters.

10.5 Diameter and the Size Model

The Cellpose pretrained models are trained using resized images so that the cells have the same median diameter across all images. If you choose to use a pretrained model, then this fixed median diameter is used. **Omnipose models are generally not trained with rescaling.** `cyto2_omni` is the exception, as its images are extremely diverse in size.

If you choose to train from scratch, you can set the median diameter you want to use for rescaling with the `--diameter` flag, or set it to `0` to disable rescaling. The `cyto`, `cyto2`, and `cyto2_omni` models were trained with a diameter of 30 pixels and the *nuclei* model was trained with a diameter of 17 pixels.

If your target image set varies a lot in cell diameter (i.e., the images you want to segment vary unpredictably in size), you may also want to learn a *SizeModel()* that predicts the diameter from the network style vectors. Add the flag `--train_size` and this model will be trained and saved as an `*.npy` file. **Omnipose models generally do not come with a** *SizeModel()*, with the exception of `cyto2_omni`.

10.6 Examples

To train on cytoplasmic images (green cyto and red nuclei) starting with a pretrained model from cellpose_omni (cyto or nuclei):

```
omnipose --train --dir <train_path> --pretrained_model cyto --chan 2 --chan2 1
```

You can train from scratch as well:

```
omnipose --train --dir <train_path> --pretrained_model None
```

You can also specify the full path to a pretrained model to use:

```
omnipose --dir <train_path> --pretrained_model <model_path> --save_png
```

To train the `bact_phase_omni` model from scratch using the same parameters from the Omnipose paper, download the dataset and run

```
omnipose --train --use_gpu --dir <bacterial_dataset_directory> --mask_filter _masks \
         --n_epochs 4000 --pretrained_model None --learning_rate 0.1 --diameter 0 \
         --batch_size 16  --RAdam --nclasses 3
```

10.7 Training 3D models

To train a 3D model on image volumes, specify the dimension argument: `--dim 3`. You may run out of VRAM on your GPU. In that case, you can specify a smaller crop size, *e.g.*, `--tyx 50,50,50`. The command I used in the paper on the *Arabidopsis thaliana* lateral root primordia dataset was:

```
omnipose --use_gpu --train --dir <path> --mask_filter _masks \
         --n_epochs 4000 --pretrained_model None --learning_rate 0.1 --save_every 50 \
         --save_each  --verbose --look_one_level_down --all_channels --dim 3 \
         --RAdam --batch_size 4 --diameter 0 --nclasses 3
```

CHAPTER

ELEVEN

MODELS

All 2D models originally published in the Cellpose and Omnipose papers use **nchan**=2. This is because Cellpose defaults are set to train models that use two channels for segmentation (usually cytoplasm and nucleus). Images without a second channel are just padded with 0s. I think most users will train Omnipose on mono-channel images, so now **nchan**=1 by default.

Tip: Always specify `nchan` and `nclasses` when training and evaluating models.

Omnipose used to have a boundary prediction, so **nclasses**=3 (flow field, distance field, and boundary field in 2D). The current version of Omnipose no longer needs a boundary prediction, so **nclasses**=2 is the default.

See the table below for named models and their corresponding `nchan`, `nclasses`.

11.1 Pretrained models

model	nchan	nclasses	dim
bact_phase_omni	2	3	2
bact_fluor_omni	2	3	2
cyto2_omni	2	3	2
worm_omni	2	3	2
plant_omni	2	3	3
bact_phase_omni_2	1	2	2

Cellpose models all have **nchan**=2, **nclasses**=2, and **dim**=2 (3D Cellpose uses 2D models to approximate 3D output). This means that if you wanted to, you could train an Omnipose model based on a Cellpose model using these hyperparameters (see *Transfer learning*).

CHAPTER

TWELVE

IN A NOTEBOOK

I have three thorough tutorials on using Omnipose in a Jupyter notebook. The first focuses on mono-channel segmentation of locally saved images. The second shows how to use a second channel to aid in segmentation, also taking the opportunity to demonstrate how to download images on-the-fly in a notebook. The third shows how to do 3D segmentation, dealing with GPU VRAM bottlenecks and visualization strategies along the way.

12.1 The basics in 2D

This notebook demonstrates how to load images, display them, segment them using Omnipose, and visualize both the segmentation results and the intermediate network output. Here we show the details behind the most typical workflow: single-channel segmentation. The bacterial images used are (1) from my own image library and (2-5) from the DeLTA2.0 paper. From the latter, we shall see how to handle images that are intrinsically grayscale but were exported and published as RGB(A) - *i.e.*, there is no extra information in those extra channels to aid segmentation. For two-channel segmentation, see the `multi_channel_cyto` notebook.

Before running this notebook, install the latest version of Omnipose from GitHub.

```
# Import dependencies
import numpy as np
from cellpose_omni import models, core

# This checks to see if you have set up your GPU properly.
# CPU performance is a lot slower, but not a problem if you
# are only processing a few images.
use_GPU = core.use_gpu()
print('>>> GPU activated? {}'.format(use_GPU))

# for plotting
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rcParams['figure.dpi'] = 300
plt.style.use('dark_background')
%matplotlib inline
```

12.1.1 How to load your images

There are several ways to load your image files into a notebook. If you have a specific set of images, put their full paths into a list. For example:

```
# make it a list even if there is only one file
files = ['path_to_image_1']
files = ['path_to_image_1','path_to_image_2']

# you can also add to the list like so:
files = files + ['path_to_image_3']
```

Alternatively, you can load all the images in a directory. Here are a few templates you can use to get the list of directories automatically by searching for image files matching a cetrain file lane with an extension and keywords in the file name.

```
from pathlib import Path

basedir = '<path_to_image_folder>'
# use rglob to search subfolders recursively
files = [str(p) for p in Path(basedir).rglob("*.tif")]
# change the search string to grab only one channel
files = [str(p) for p in Path(basedir).glob("*C1.png")]
# specify a match anywhere in the file name
files = [str(p) for p in Path(basedir).glob("*488*.png")]
```

We can also use the *cellpose_omni.io* library to grab all the images in the test_files folder. This is very handy for grabbing images of different extensions. Here we are using four RGB(A) images from the DeLTA 2.0 training set (on which the bact_phase_omni model has never been trained) as well as an RGB image acquired in the same lab as much of the Omnipose `bact_phase` dataset.

```
from pathlib import Path
import os
from cellpose_omni import io
import omnipose
omnidir = Path(omnipose.__file__).parent.parent
basedir = os.path.join(omnidir,'docs','test_files')
files = io.get_image_files(basedir)
```

Next we read in the images from the file list. It's a good idea to display the images before proceeding. Here I happen to be reading in some RBG tiles of grayscale phase contrast images (such as you might use for figures etc.) as well as some single-channel images. As part of the visualization process, the images are rescaled to be in the range 0-1. Omnipose does this exact thing internally (you don't have to rescale them prior to running segmentation via CLI).

```
from cellpose_omni import io, transforms
from omnipose.utils import normalize99
imgs = [io.imread(f) for f in files]

# print some info about the images.
for i in imgs:
    print('Original image shape:',i.shape)
    print('data type:',i.dtype)
    print('data range: min {}, max {}\n'.format(i.min(),i.max()))
nimg = len(imgs)
print('\nnumber of images:',nimg)
```

(continues on next page)

(continued from previous page)

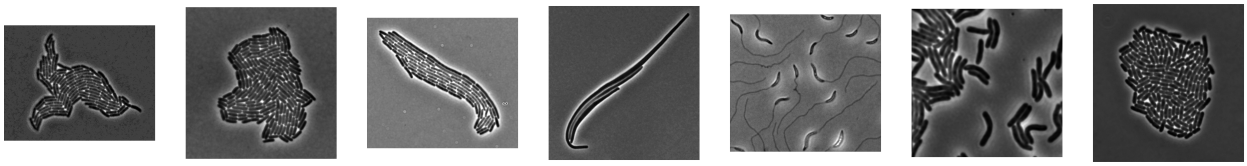
```

fig = plt.figure(figsize=[40]*2,frameon=False) # initialize figure
print('\n')
for k in range(len(imgs)):
    img = transforms.move_min_dim(imgs[k]) # move the channel dimension last
    if len(img.shape)>2:
        # imgs[k] = img[:,:,1] # could pick out a specific channel
        imgs[k] = np.mean(img,axis=-1) # or just turn into grayscale

    imgs[k] = normalize99(imgs[k])
    # imgs[k] = np.pad(imgs[k],10,'edge')
    print('new shape: ', imgs[k].shape)
    plt.subplot(1,len(files),k+1)
    plt.imshow(imgs[k],cmap='gray')
    plt.axis('off')
```

```
Original image shape: (287, 377, 3)
data type: uint8
data range: min 4, max 22

Original image shape: (564, 564, 3)
data type: uint8
data range: min 30, max 203

Original image shape: (783, 908)
data type: uint8
data range: min 0, max 255

Original image shape: (396, 390, 4)
data type: uint8
data range: min 49, max 255

Original image shape: (281, 310)
data type: uint16
data range: min 360, max 64813

Original image shape: (384, 392)
data type: uint16
data range: min 0, max 65535

Original image shape: (334, 321)
data type: uint16
data range: min 2582, max 39614


number of images: 7


new shape:  (287, 377)
new shape:  (564, 564)
new shape:  (783, 908)
new shape:  (396, 390)
```

(continues on next page)

(continued from previous page)

```
new shape:  (281, 310)
new shape:  (384, 392)
new shape:  (334, 321)
```



Note that the first two images are RGB, the third and fifth are mono-channel, and the fourth is RGBA (the alpha channel encodes transparency). Exporting to RGB is usually just done for making diagrams or making images compatible with non-scientific viewing software. Pro tip: Adobe Illustrator *will not* interpolate the pixels in your image if you save it as RGB, but it *will* if you keep it mono-channel. Usually you want exact, non-interpolated (pixelated) images to be presented since it is your raw data, so you can convert it to grayscale by `im_RGB = [im,im,im]` (or more slick, `im_RGB = [im,]*3` or `*4` for RGBA). However, storing *all* your images this way is a waste of space - just do it for the ones you need for a figure.

Also note that the DeLTA images (1-4) are uint8, so `0` to `2**8-1 = 255`. Image 1 only takes up values in the range 4 to 22 out of a possible 0 to 255, meaning it was probably way too dark and not rescaled prior to conversion to an 8-bit image. In my experience, images are typically 14-bit (that depends on your camera) and therefore saved as 16-bit lossless formats like PNG or TIF (Omnipose can detect and segment JPEGs, but you would never use those for anything scientific, even for figures due to compression artifacts). Using only `22-4 = 18` levels of gray to depict the cells causes the distinct 'posterized' effect that you can see if you zoom up on the image.

12.1.2 Initialize model

Here we use one of the built-in model names. You can print out the available model names, too:

```
import cellpose_omni
from cellpose_omni import models
from cellpose_omni.models import MODEL_NAMES

MODEL_NAMES
```

```
['bact_phase_omni',
 'bact_fluor_omni',
 'worm_omni',
 'worm_bact_omni',
 'worm_high_res_omni',
 'cyto2_omni',
 'plant_omni',
 'bact_phase_cp',
 'bact_fluor_cp',
 'plant_cp',
 'worm_cp',
 'cyto',
 'nuclei',
 'cyto2']
```

We will choose the `bact_phase_omni` model.

```
model_name = 'bact_phase_omni'
model = models.CellposeModel(gpu=use_GPU, model_type=model_name)
```

```
2024-05-10 00:54:43,023 [INFO] models __init__....() line␣
↪427 >>bact_phase_omni<< model set to be used
2024-05-10 00:54:43,050 [INFO] core _use...torch() line␣
↪74 ** TORCH GPU version installed and working. **
2024-05-10 00:54:43,051 [INFO] assi...evice() line␣
↪85 >>>> using GPU
```

12.1.3 Run segmentation

```
import time
chans = [0,0] #this means segment based on first channel, no second channel

n = [-1] # make a list of integers to select which images you want to segment
n = range(nimg) # or just segment them all

# define parameters
params = {'channels':chans, # always define this with the model
          'rescale': None, # upscale or downscale your images, None = no rescaling
          'mask_threshold': -1, # erode or dilate masks with higher or lower values
          'flow_threshold': 0, # default is .4, but only needed if there are spurious␣
↪masks to clean up; slows down output
          'transparency': True, # transparency in flow output
          'omni': True, # we can turn off Omnipose mask reconstruction, not advised
          'cluster': True, # use DBSCAN clustering
          'resample': True, # whether or not to run dynamics on rescaled grid or␣
↪original grid
          'verbose': False, # turn on if you want to see more output
          'tile': False, # average the outputs from flipped (augmented) images; slower,␣
↪usually not needed
          'niter': 7, # None lets Omnipose calculate # of Euler iterations (usually <20)␣
↪but you can tune it for over/under segmentation
          'augment': False, # Can optionally rotate the image and average outputs,␣
↪usually not needed
          'affinity_seg': False, # new feature, stay tuned...
         }

tic = time.time()
masks, flows, styles = model.eval([imgs[i] for i in n],**params)

net_time = time.time() - tic

print('total segmentation time: {}s'.format(net_time))
```

```
  0%|          | 0/7 [00:00<?, ?it/s]
```

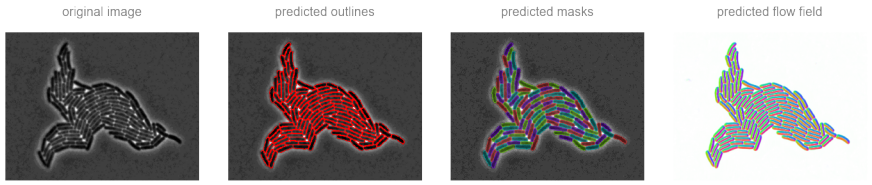
```
total segmentation time: 1.0175395011901855s
```

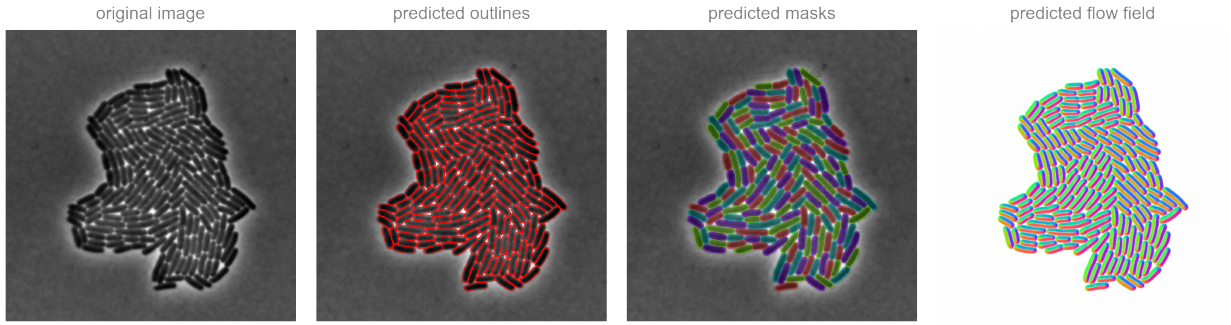
Note that since some functions require just-in-time numba compilation, the first round of segmentation will be slightly

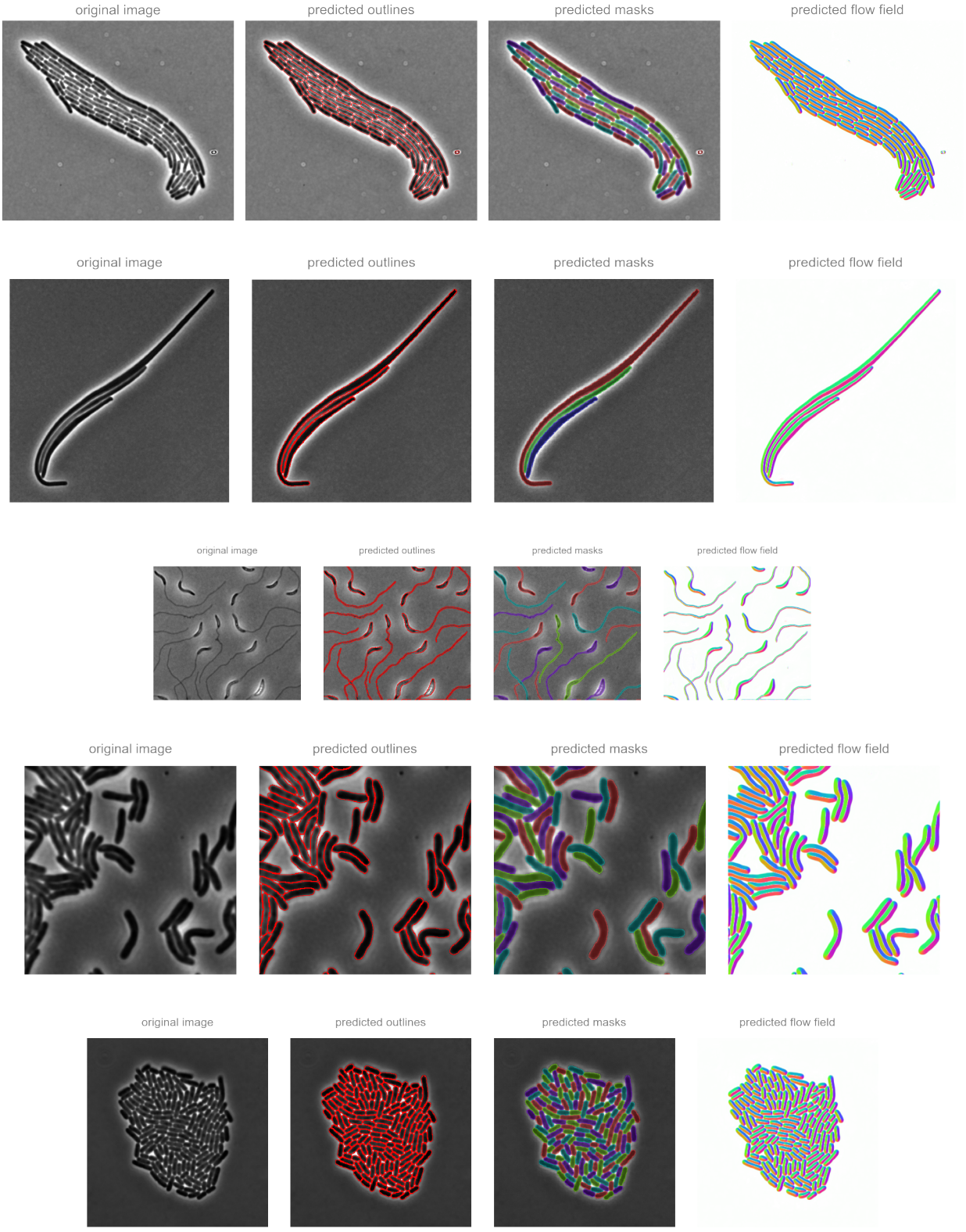
slower than subsequent runs.

12.1.4 Plot the results

```
from cellpose_omni import plot
import omnipose

for idx,i in enumerate(n):

    maski = masks[idx] # get masks
    bdi = flows[idx][-1] # get boundaries
    flowi = flows[idx][0] # get RGB flows

    # set up the output figure to better match the resolution of the images
    f = 5
    szX = maski.shape[-1]/mpl.rcParams['figure.dpi']*f
    szY = maski.shape[-2]/mpl.rcParams['figure.dpi']*f
    fig = plt.figure(figsize=(szY,szX*4))
    fig.patch.set_facecolor([0]*4)

    plot.show_segmentation(fig, omnipose.utils.normalize99(imgs[i]),
                           maski, flowi, bdi, channels=chans, omni=True,
                           interpolation=None)

    plt.tight_layout()
    plt.show()
```






original image
predicted outlines
predicted masks
predicted flow field
original image
predicted outlines
predicted masks
predicted flow field
original image
predicted outlines
predicted masks
predicted flow field
original image
predicted outlines
predicted masks
predicted flow field
original image
predicted outlines
predicted masks
predicted flow field

12.1.5 Save the results

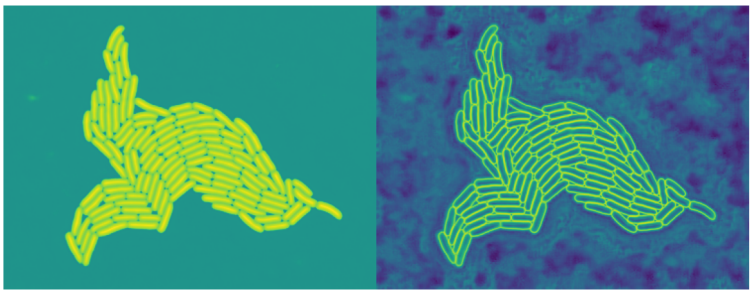
Often you will want to save your masks before moving on to analysis (that way to can just load them in instead of re-running segmentation). I improved the `cellpose.io` function quite a bit to be more flexible in where it can save. See the documentation page for the full list of options.

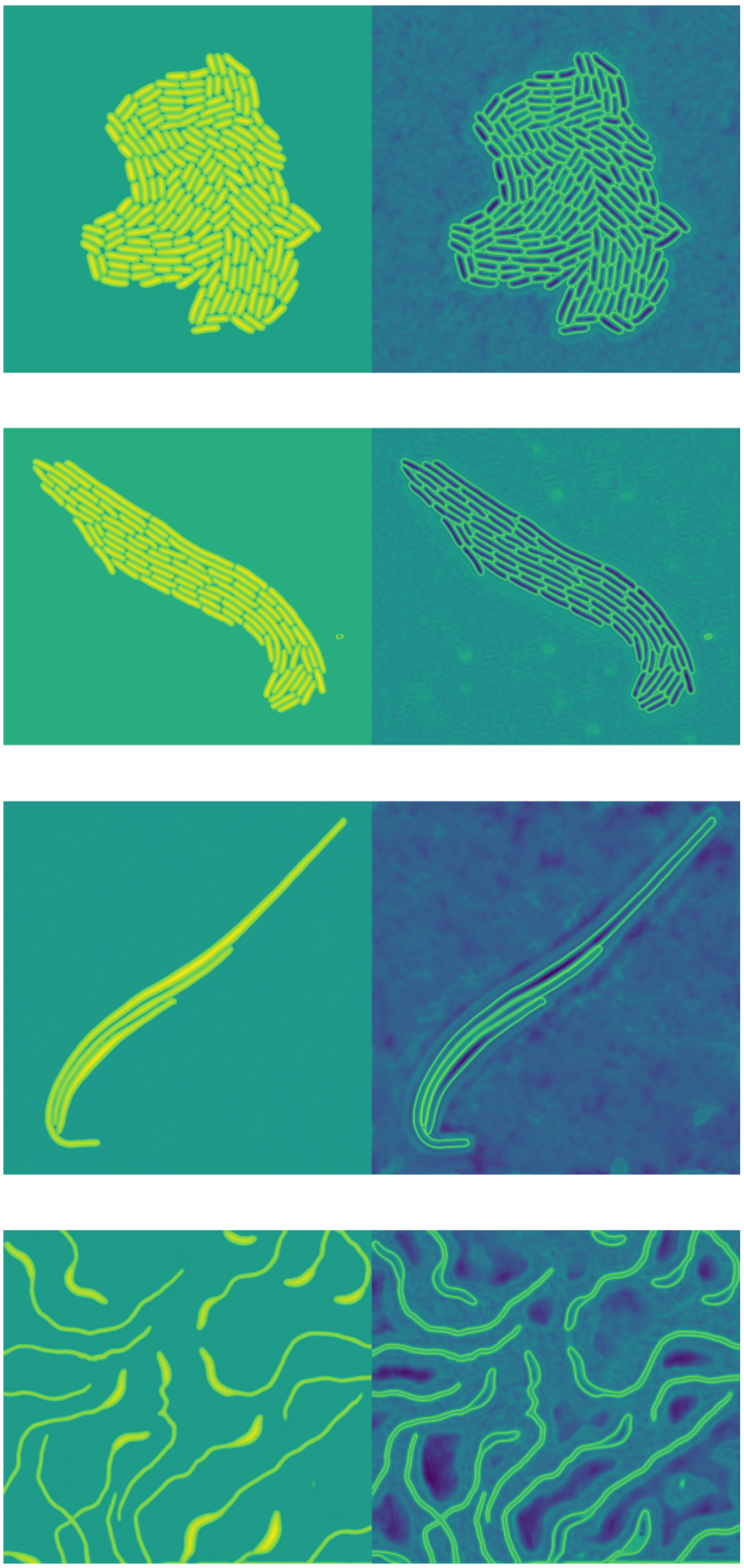
```
io.save_masks(imgs, masks, flows, files,
              png=False,
              tif=True, # whether to use PNG or TIF format
              suffix='', # suffix to add to files if needed
              save_flows=False, # saves both RGB depiction as *_flows.png and the raw␣
↪components as *_dP.tif
              save_outlines=False, # save outline images
              dir_above=0, # save output in the image directory or in the directory␣
↪above (at the level of the image directory)
              in_folders=True, # save output in folders (recommended)
              save_txt=False, # txt file for outlines in imageJ
              save_ncolor=False) # save ncolor version of masks for visualization and␣
↪editing
```

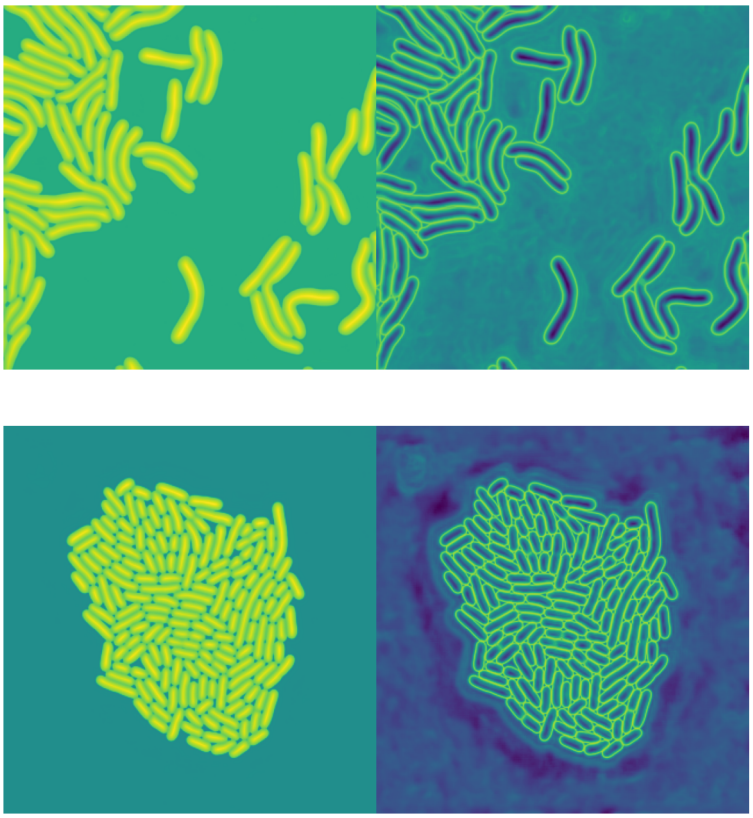
12.1.6 Debug results

The RGB flows shown above will give you some insight as to if there is an issue with the flow field outputs, but you can also check out the boundary and distance output:

```
for idx,i in enumerate(n):

    disti = flows[idx][2] # distance field prediction
    bdlti = flows[idx][4] # boundary logits prediction
    omnipose.plot.imshow(np.hstack([disti,bdlti]),5, cmap='viridis')
```







Notes on the above

The distance field is trained with background pixels set to `-5` in older models and `-<mean diameter>` in newer models. This helps to make the desired network output more balanced and give more distinction between edge pixels (which have values close to 0) and background (which ordinarily would have a value of 0). The flow field, being the gradient of the distance field, *by definition* has a magnitude of 1 everywhere - but we rescale it by `5` for training. This helps by bringing the desired flow component output more in the range of the boundary output, which is the input to the sigmoid function (so-called 'logits') and therefore ranges from about `-5` to `5`.

What you can see in the images above is that the features in the boundary, distance, and flow fields are all very consistent with each other. For example, the flow field has positive divergence where the boundary output is high and negative divergence where the distance field is high. This is by design, as I included the boundary field for the sole purpose of improving the prediction accuracy on the flow and distance fields.

12.2 Multiple channels

Omnipose inherits the capability of Cellpose to segment based on multi-channel images. We will use this as an opportunity to show how we can run several models at once on the same image(s), in this case comparing Omnipose to Cellpose trained on the cyto2 dataset.

```
# First, import dependencies.
import numpy as np
import time, os, sys
from cellpose_omni import models, core, utils
```

(continues on next page)

(continued from previous page)

```

# This checks to see if you have set up your GPU properly.
# CPU performance is a lot slower, but not a problem if
# you are only processing a few images.
use_GPU = core.use_gpu()
print('>>> GPU activated? %d'%use_GPU)

# for plotting
import matplotlib.pyplot as plt
plt.style.use('dark_background')
import matplotlib as mpl
%matplotlib inline
mpl.rcParams['figure.dpi'] = 300
from omnipose.plot import imshow, colorize
```

```
2024-01-18 00:27:35,995 [INFO] ** TORCH GPU version installed and working. **
>>> GPU activated? 1
```

12.2.1 Load file

This is one of the images from the cyto2 test dataset. Note that it is a good idea to always work with lists, even when the list of images is 1 long. It allows you to reuse your code easily when you do have a larger set of images to process.

```
from urllib.parse import urlparse
import skimage.io


urls = ['http://www.cellpose.org/static/images/img02.png']
files = []
for url in urls:
    parts = urlparse(url)
    filename = os.path.basename(parts.path)
    if not os.path.exists(filename):
        sys.stderr.write('Downloading: "{}" to {}\n'.format(url, filename))
        utils.download_url_to_file(url, filename)
    files.append(filename)

imgs = [skimage.io.imread(f) for f in files]
# print(imgs[0].shape)
imgs = [np.stack((im[...,-1],im[...,1])) for im in imgs] # put cytosol in 1st channel,␣
↪nucleus in 2nd
nimg = len(imgs)
```

```
(349, 467, 3)
```

Read in the images from the file list. It's a good idea to display the images before proceeding.

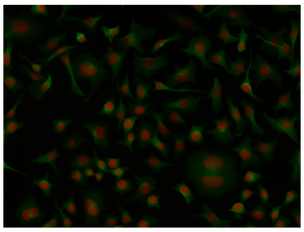
```
from cellpose_omni import io, transforms

```

(continues on next page)

(continued from previous page)

```
# print some infor about the images
for i in imgs:
    print('img shape:',i.shape)
nimg = len(imgs)
# print(nimg)

colors = np.array([[1,0,0],[0,1,0]])*1.0
for k in range(len(imgs)):
    imgs[k] = transforms.normalize99(imgs[k],omni=True)
    rgb = colorize(imgs[k],colors=colors)
    imshow(rgb)
```

```
img shape: (2, 349, 467)
```



12.2.2 Initialize models

We will compare two models here: cyto2 (Cellpose) and cyto2_omni. The latter was trained via the following command:

```
python -m cellpose --train --use_gpu --dir /home/kcutler/DataDrive/cyto2/train --mask_
↪filter _masks --n_epochs 4000 --pretrained_model None  --learning_rate 0.1 --diameter␣
↪36 --save_every 50 --save_each --omni --verbose --chan 1 --chan2 2 --RAdam --batch_
↪size 16 --img_filter _img
```

```
model_name = ['cyto2','cyto2_omni']
L = len(model_name)
model = [models.CellposeModel(gpu=use_GPU, model_type=m) for m in model_name]
```

```
2024-01-18 00:31:08,507 [INFO] >>cyto2<< model set to be used
2024-01-18 00:31:08,540 [INFO] ** TORCH GPU version installed and working. **
2024-01-18 00:31:08,540 [INFO] >>>> using GPU
2024-01-18 00:31:08,651 [INFO] >>cyto2_omni<< model set to be used
2024-01-18 00:31:08,652 [INFO] ** TORCH GPU version installed and working. **
2024-01-18 00:31:08,652 [INFO] >>>> using GPU
```

12.2.3 Run segmentation

The channels input can be very confusing. In the Cellpose documentation, it is stated that the list `[chan,chan2]` should represent the main channel to segment (`chan`) and the optional nuclear channel (`chan2`). But to train via CLI, `chan` is the "channel to segment" and `chan2` is the nuclear channel, and the Cellpose team states the CLI command used to train their model used `--chan 2 --chan2 1`. Because 0 is grayscale and 1,2,3 are R,G,B (note the 1-based indexing here) this means that the given training command actually trains with G cytosol and R nuclei. This might imply that the cyto2_omni model actually is trained 'incorrectly', if I understood this right.

On top of this, the downloaded image has nuclei in channel 2 and cytosol in channel 1 (blue and green, respectively), whereas the cyto2 dataset shows cytosol as channel 0 and nuclei as channel 1 (this may be a result of using OpenCV, which usies BGR by default instead of RGB). So in fact, I should have trained the cyto2_omni model with `--chan 1 --chan2 2`. (Have not yet done this with most recent models...) Keep this in mind as you train your own models.

Note: You can train Omnipose on arbitrary numbers of channels using the `--all_channels` parameter. The reason for the cyto* models being trained with two channel was to focus on eukaryotes with cytosol and nuclei tags while also including many other images that might be single-channel. The chan, chan2 arguments and corresponding code in Cellpose is thus highly specific to this use case. Omnipose should probably always be trained with `--all_channels` on a dataset with the same number of channels across all images. After training, initialize the model with `nchan=nchan` and evaluate using `chans=None`.

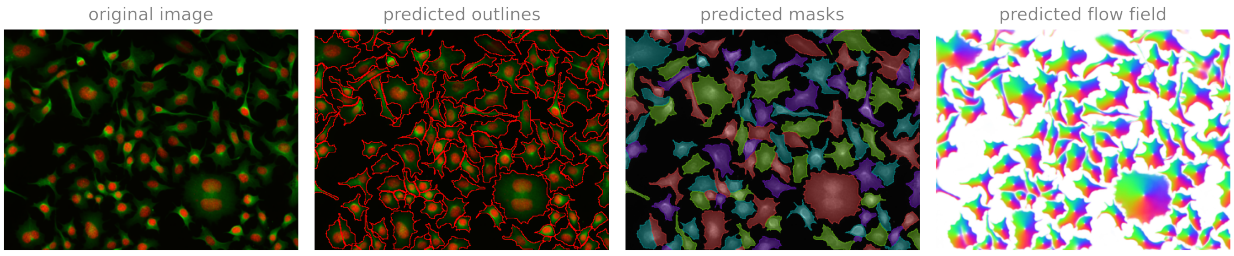
For now, the following shows what channel arguments you need for the provided cyto2 models:

```
chans = [[2,1],[1,2]] # green cytoplasm [2] and red nucleus [1], see above
n = range(nimg)

# define parameters
mask_threshold = [-1,-1,-1] #new model might need a bit lower
verbose =0 # turn on if you want to see more output
use_gpu = use_GPU #defined above
transparency = True # transparency in flow output
rescale= None # give this a number if you need to upscale or downscale your images
flow_threshold = 0 # default is .4, but only needed if there are spurious masks to clean up; slows down output
resample = False #whether or not to run dynamics on rescaled grid or original grid

N = L+1 # three options: pure cellpose, mixed, omnipose, new omnipose
omni = [0,1,1]
ind = [0,0,1]
masks, flows, styles = [[]]*N, [[]]*N, [[]]*N

diameter = 30
for i in range(N):
    masks[i], flows[i], styles[i] = model[ind[i]].eval([imgs[i] for i in n], channels=chans[ind[i]],
                                                       diameter=diameter,
                                                       mask_threshold=mask_threshold[i],
                                                       transparency=transparency,
                                                       flow_threshold=flow_threshold,
                                                       omni=omni[i], #toggle omni
                                                       resample=resample,verbose=verbose,
                                                       cluster=omni[i],
                                                       interp=True, tile=False)
```

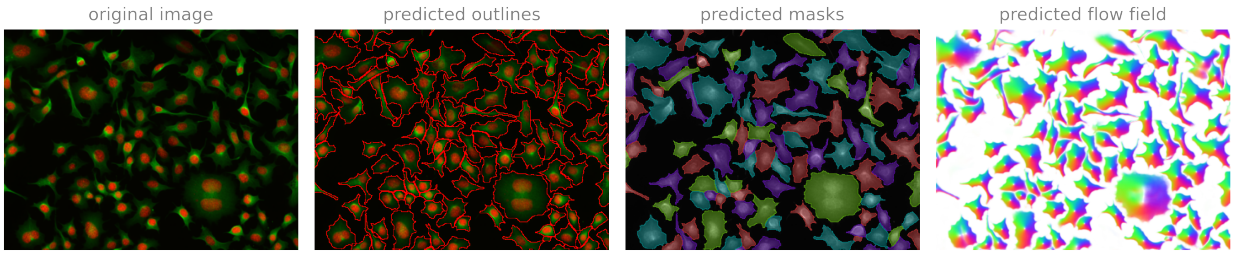
12.2.4 Plot the results

```
from cellpose_omni import plot
import omnipose

for idx,i in enumerate(n):

    for k,ki in enumerate(ind):

        print('model: {}, omni: {}'.format(model_name[ki],omni[ki]))
        maski = masks[k][idx] # get masks
        flowi = flows[k][idx][0] # get RGB flows
        imgi = omnipose.utils.normalize99(imgs[i])

        # set up the output figure to better match the resolution of the images
        f = 10
        szX = maski.shape[-1]/mpl.rcParams['figure.dpi']*f
        szY = maski.shape[-2]/mpl.rcParams['figure.dpi']*f
        fig = plt.figure(figsize=(szY,szX*4))
        fig.patch.set_facecolor([0]*4)

        plot.show_segmentation(fig,
                               imgi,
                               maski, flowi,
                               channels=chans[i],
                               channel_axis=0,
                               omni=True,
                               img_colors=colors,
                          interpolation=None)

        plt.tight_layout()
        plt.show()
```

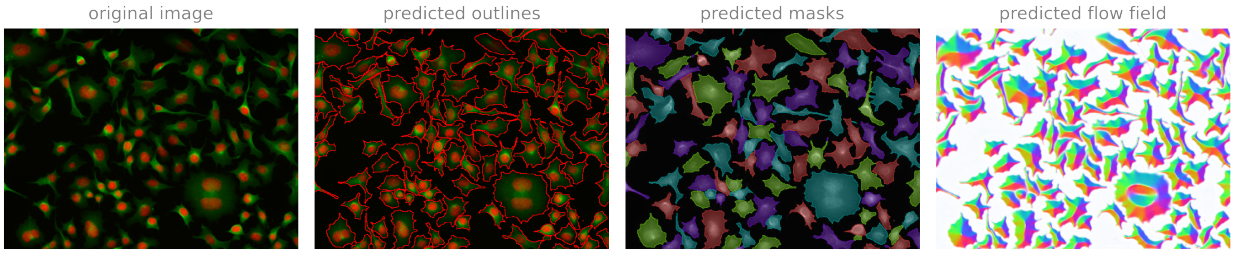
```
model: cyto2, omni: 0
```




```
model: cyto2, omni: 0
```




```
model: cyto2_omni, omni: 1
```



Some comments on the above: Omnipose pre-processes the images slightly differently (see `normalize99`) and therefore the flow is a bit different even with the same model and input image compared to stock Cellpose. The `cluster` option helps a lot to get accurate masks with Omnipose in thin regions, but can result in under-segmentation between cells with poorly-defined flow fields. This can be a weakness of Omnipose relative to Cellpose, but as seen in the paper, Omnipose does slightly better than Cellpose on the cyto2 dataset on average. On roundish and low-accuracy datasets like cyto2, Omnipose simply does better in some areas and worse in others.

12.3 Omnipose in 3D

You can use the `dim` (dimension) argument to tell Omnipose to segment your images using a 3D model. This means that an image stack or 3D array is treated as a 3D volume given to a network trained on 3D volumes. This is very different from `do_3D` in Cellpose, which cleverly leveraged 2D predictions on all 2D slices of a 3D volume to construct a 3D flow field for segmentation. It turns out that the pseudo-ND Cellpose flows are an approximation to the true 3D flows of Omnipose, because the flows in each slice point to a local center of the cell, a.k.a. the cell skeleton to which the Omnipose field points. Thus, is not recommended to use Omnipose 2D slice predictions with `do_3D`. Instead, this notebook assumes you have trained a 3D model such as the `plant_omni` model.

```
# Import dependencies
import numpy as np
from cellpose_omni import models, core

# This checks to see if you have set up your GPU properly.
# CPU performance is a lot slower, but not a problem if you
# are only processing a few images.
use_GPU = core.use_gpu()
print('>>> GPU activated? %d'%use_GPU)

# for plotting
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rcParams['figure.dpi'] = 300
```

(continues on next page)

(continued from previous page)

```
plt.style.use('dark_background')
%matplotlib inline
```

```
2023-08-08 00:57:34,560 [INFO] ** TORCH GPU version installed and working. **
>>> GPU activated? 1
```

12.3.1 Read in data

Here I am choosing one of the scaled-down volumes of the plant *Arabidopsis thaliana* dataset we used in the Omnipose paper.

```
from pathlib import Path
import os
from cellpose_omni import io

basedir = os.path.join(Path.cwd().parent,'test_files_3D')
files = io.get_image_files(basedir)
files # this displays the variable if it the last thing in the code block
```

```
['/home/kcutler/DataDrive/omnipose/docs/test_files_3D/Movie1_t00004_crop_gt.tif']
```

```
from cellpose_omni import io, transforms
from omnipose.utils import normalize99

imgs = [io.imread(f) for f in files]

# print some info about the images.
for i in imgs:
    print('Original image shape:',i.shape)
    print('data type:',i.dtype)
    print('data range:', i.min(),i.max())
nimg = len(imgs)
print('number of images:',nimg)
```

```
Original image shape: (162, 207, 443)
data type: uint8
data range: 0 247
number of images: 1
```

12.3.2 Initialize model

`plant_omni` is the model trained on these plant cell images. (The image we loaded is from the test set, of course.)

```
from cellpose_omni import models
model_name = 'plant_omni'

dim = 3
nclasses = 3 # flow + dist + boundary
```

(continues on next page)

(continued from previous page)

```
nchan = 1
omni = 1
rescale = False
diam_mean = 0
use_GPU = 0 # Most people do not have enough VRAM to run on GPU... 24GB not enough for this image, need nearly 48GB
model = models.CellposeModel(gpu=use_GPU, model_type=model_name, net_avg=False,
                             diam_mean=diam_mean, nclasses=nclasses, dim=dim, nchan=nchan)
```

```
2023-08-08 00:57:44,364 [INFO] >>plant_omni<< model set to be used
sdggsfgs
2023-08-08 00:57:44,364 [INFO] >>>> using CPU
```

12.3.3 Run segmentation

```
import torch
torch.cuda.empty_cache()
mask_threshold = -5 #usually this is -1
flow_threshold = 0.
diam_threshold = 12
net_avg = False
cluster = False
verbose = 1
tile = True
chans = None
compute_masks = 1
resample=False
rescale=None
omni=True
flow_factor = 10 # multiple to increase flow magnitude, useful in 3D
transparency = True

nimg = len(imgs)
masks_om, flows_om = [[]]*nimg,[[]]*nimg

# splitting the images into batches helps manage VRAM use so that memory can get properly released
# here we have just one image, but most people will have several to process
for k in range(nimg):
    masks_om[k], flows_om[k], _ = model.eval(imgs[k],
                                             channels=chans,
                                             rescale=rescale,
                                             mask_threshold=mask_threshold,
                                             net_avg=net_avg,
                                             transparency=transparency,
                                             flow_threshold=flow_threshold,
                                             omni=omni,
                                             resample=resample,
                                             verbose=verbose,
```

(continues on next page)

(continued from previous page)

```
                                                diam_threshold=diam_threshold,
                                                cluster=cluster,
                                                tile=tile,
                                                compute_masks=compute_masks,
                                                flow_factor=flow_factor)
```

```
2023-08-08 00:57:45,752 [INFO] Evaluating with flow_threshold 0.00, mask_threshold -5.00
2023-08-08 00:57:45,753 [INFO] using omni model, cluster False
2023-08-08 00:57:45,753 [INFO] not using dataparallel
2023-08-08 00:57:45,878 [INFO] multi-stack tiff read in as having 162 planes 1 channels
2023-08-08 00:58:36,584 [INFO] mask_threshold is -5.000000
2023-08-08 00:58:36,585 [INFO] Using hysteresis threshold.
dP_ times 10 for >2d, still experimenting
2023-08-08 00:58:37,615 [INFO] niter is None
2023-08-08 00:59:54,186 [INFO] Mean diameter is 25.683111
2023-08-08 00:59:54,307 [INFO] cluster: False, SKLEARN_ENABLED: True
2023-08-08 00:59:54,571 [INFO] nclasses: 5, mask.ndim: 3
2023-08-08 00:59:54,581 [INFO] Using boundary output to split edge defects.
2023-08-08 00:59:54,776 [INFO] Done finding masks.
2023-08-08 00:59:55,705 [INFO] compute_masks() execution time: 79.1 sec
2023-08-08 00:59:55,705 [INFO]         execution time per pixel: 5.18728e-06 sec/px
2023-08-08 00:59:55,709 [INFO]         execution time per cell pixel: 1.45631e-05 sec/px
```

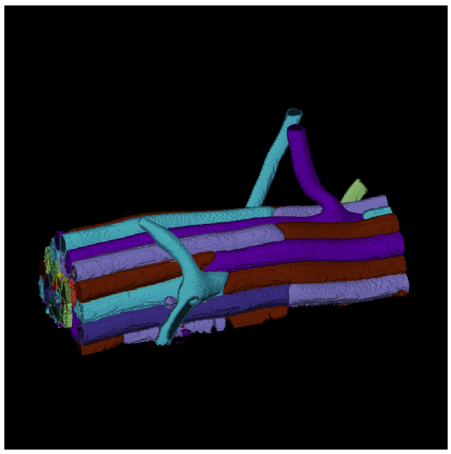
12.3.4 Plot results

3D segmentation is a lot harder to show than 2D. If anyone figures out a good way to use one of the many tools out there (`ipyvolume`, `K3D-Jupyter`, `itkwidgets`, `ipygany`) for *label* visualization (not image volumes), please let me know. Few of these are in active development, and my own 3D work requires robust label editing tools anyway, which I do not think any available tools offer. Hence I shall just load in Napari and show you an auto-captured screenshot.

```
%%capture
import ncolor
mask = masks_om[0]
mask_nc = ncolor.label(mask,max_depth=20)

import napari
viewer = napari.view_labels(mask_nc);
viewer.dims.ndisplay = 3
viewer.camera.center = [s//2 for s in mask.shape]
viewer.camera.zoom=1
viewer.camera.angles=(10.90517458968619, -20.777067798396835, 58.04311170773853)
viewer.camera.perspective=0.0
viewer.camera.interactive=True

img = viewer.screenshot(size=(1000,1000),scale=1,canvas_only=True,flash=False)
```

```
2023-08-08 00:59:58,954 [WARNING] Could not connect "org.freedesktop.IBus" to␣
↪globalEngineChanged(QString)
```

```
plt.figure(figsize=(3,3),frameon=False)
plt.imshow(img)
plt.axis('off')
plt.show()
```



12.3.5 Plot orthogonal slices

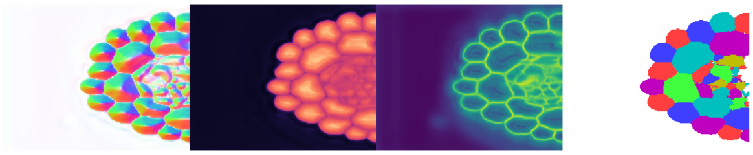
```
from cellpose_omni import plot
from omnipose.plot import apply_ncolor

mu = flows_om[0][1]
T = flows_om[0][2]
bd = flows_om[0][4]
# mu.shape,T.shape,bd.shape

d = mu.shape[0]

from omnipose.utils import rescale
c = np.array([1]*2+[0]*(d-2))
# c = np.arange(d)
def cyclic_perm(a):
    n = len(a)
    b = [[a[i - j] for i in range(n)] for j in range(n)]
    return b
slices = []
idx = np.arange(d)
cmap = mpl.colormaps['magma']
cmap2 = mpl.colormaps['viridis']

for inds in cyclic_perm(c):
    slc = tuple([slice(-1) if i else mu.shape[k+1]//2 for i,k in zip(inds,idx)])
    flow = plot.dx_to_circ(mu[np.where(inds)+slc],transparency=1)/255
    dist = cmap(rescale(T)[slc])
    bnds = cmap2(rescale(bd)[slc])
    msks = apply_ncolor(masks_om[0][slc])
```

(continues on next page)

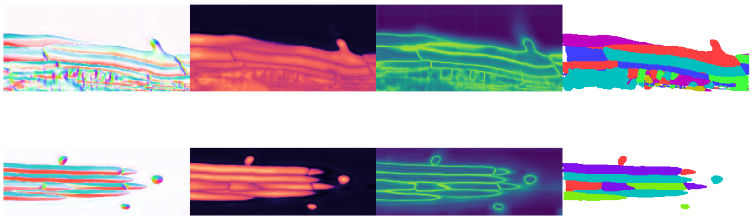
(continued from previous page)

```
    fig = plt.figure(figsize=[5]*2,frameon=False)
    plt.imshow(np.hstack((flow,dist,bnds,msks)),interpolation='none')
    plt.axis('off')
    plt.show()

```

```
4 -color algorthm failed,trying again with 5 colors. Depth 0
5 -color algorthm failed,trying again with 6 colors. Depth 1
```



```
4 -color algorthm failed,trying again with 5 colors. Depth 0
5 -color algorthm failed,trying again with 6 colors. Depth 1
```



Notes on the above

Slices do not always look crisp because we are cutting though boundaries. At these locations, the flow and distance fields darken and the boundary field brightens. This can result in flat and muddled regions that are hard to interpret. Again, interactive 3D visualization tools are needed to properly evaluate the results of the segmentation. In this case, we have cut through the middle of enough cells to confirm that the output looks reasonable.

This small dataset with problematic annotations was sufficient for demonstrating that Omnipose *can* be used on 3D data, but I again emphasize that any algorithm will only work *well* after training on well-annotated, representative examples. In this case, small cell clusters were neither well-annotated nor well-represented in the training set, and you can see the negative impact of that in this example.

These 3D models are incredibly VRAM-hungry, so all results in the paper were actually run on an AWS instance. Here I ran them on CPU, which is much slower but necessary to do even with a 24GB Titan RTX.

Runing Omnipose with do_3D

do_3D is not something you want to use with *any* Omnipose model, but you might want to use it with a 2D Cellpose model for 3D cells with extended shapes. This is because do_3D computes 2D flow fields from every yx, yz, and xz slice of the image and composites these components into a 3D field. It turns out that the center-seeking flow slices of Cellpose end up pointing roughly toward the local 3D skeleton, i.e. the do_3D Cellpose composite field approximates the true 3D flows of Omnipose. The 2D Omnipose field, on the other hand, cannot be composited into a useful 3D field.

Althought the do_3D Cellpose field directs pixels toward the skeleton, the stock Cellpose mask reconstruction algorithm tends to oversegment pixels into clusters along the skeleton. To avoid this, you can use a Cellpose model but with Omnipose mask reconstruciton by usin **omni**=True. Here is how to do this.

```
from cellpose_omni import models, core

# define cellpose model
model_name = 'plant_cp'

# this model was trained on 2D slices
dim = 2
nclasses = 2 # cellpose models have no boundary field, just flow and distance

# Cellpose defaults to 2 channels;
# this is the setup for grayscale in that case
nchan = 2
chans = [0,0]

# no rescaling for this model
diam_mean = 0


use_GPU = core.use_gpu()
model = models.CellposeModel(gpu=use_GPU, model_type=model_name, net_avg=False,
                             diam_mean=diam_mean, nclasses=nclasses, dim=dim,␣
↪nchan=nchan)


# segmentation parameters
omni = 1
rescale = False
mask_threshold = 0
net_avg = 0
verbose = 0
tile = 0
compute_masks = 1
rescale = None
flow_threshold=0.
do_3D=True
flow_factor=10

masks_cp, flows_cp, _ = model.eval(imgs,
                                   channels=chans,
                                   rescale=rescale,
                                   mask_threshold=mask_threshold,
```

(continues on next page)

(continued from previous page)

```
                                    net_avg=net_avg,
                                    transparency=True,
                                    flow_threshold=flow_threshold,
                                    verbose=verbose,
                                    tile=tile,
                                    compute_masks=compute_masks,
                                    do_3D=True,
                                    omni=omni,
                                    flow_factor=flow_factor)
```

```
2023-08-08 01:01:25,558 [INFO] ** TORCH GPU version installed and working. **
2023-08-08 01:01:25,558 [INFO] >>plant_cp<< model set to be used
2023-08-08 01:01:25,559 [INFO] ** TORCH GPU version installed and working. **
2023-08-08 01:01:25,560 [INFO] >>>> using GPU
2023-08-08 01:01:25,643 [INFO] using dataparallel
2023-08-08 01:01:25,682 [INFO] multi-stack tiff read in as having 162 planes 1 channels
2023-08-08 01:01:26,362 [INFO] running YX: 162 planes of size (207, 443)
2023-08-08 01:01:26,390 [INFO] 0%|          | 0/15 [00:00<?, ?it/s]
2023-08-08 01:01:26,550 [INFO] 7%|6         | 1/15 [00:00<00:02,  6.28it/s]
2023-08-08 01:01:26,700 [INFO] 13%|#3        | 2/15 [00:00<00:01,  6.50it/s]
2023-08-08 01:01:26,850 [INFO] 20%|##        | 3/15 [00:00<00:01,  6.58it/s]
2023-08-08 01:01:27,000 [INFO] 27%|##6       | 4/15 [00:00<00:01,  6.62it/s]
2023-08-08 01:01:27,149 [INFO] 33%|###3      | 5/15 [00:00<00:01,  6.64it/s]
2023-08-08 01:01:27,298 [INFO] 40%|####      | 6/15 [00:00<00:01,  6.66it/s]
2023-08-08 01:01:27,448 [INFO] 47%|####6     | 7/15 [00:01<00:01,  6.67it/s]
2023-08-08 01:01:27,597 [INFO] 53%|#####3    | 8/15 [00:01<00:01,  6.68it/s]
2023-08-08 01:01:27,747 [INFO] 60%|######    | 9/15 [00:01<00:00,  6.68it/s]
2023-08-08 01:01:27,896 [INFO] 67%|######6   | 10/15 [00:01<00:00,  6.69it/s]
2023-08-08 01:01:28,046 [INFO] 73%|#######3  | 11/15 [00:01<00:00,  6.69it/s]
2023-08-08 01:01:28,197 [INFO] 80%|########  | 12/15 [00:01<00:00,  6.66it/s]
2023-08-08 01:01:28,347 [INFO] 87%|########6 | 13/15 [00:01<00:00,  6.66it/s]
2023-08-08 01:01:28,497 [INFO] 93%|#########3| 14/15 [00:02<00:00,  6.67it/s]
2023-08-08 01:01:28,641 [INFO] 100%|##########| 15/15 [00:02<00:00,  6.74it/s]
2023-08-08 01:01:28,642 [INFO] 100%|##########| 15/15 [00:02<00:00,  6.66it/s]
2023-08-08 01:01:28,868 [INFO] running ZY: 207 planes of size (162, 443)
2023-08-08 01:01:28,900 [INFO] 0%|          | 0/19 [00:00<?, ?it/s]
2023-08-08 01:01:29,030 [INFO] 5%|5         | 1/19 [00:00<00:02,  7.77it/s]
2023-08-08 01:01:29,159 [INFO] 11%|#         | 2/19 [00:00<00:02,  7.76it/s]
2023-08-08 01:01:29,288 [INFO] 16%|#5        | 3/19 [00:00<00:02,  7.75it/s]
2023-08-08 01:01:29,417 [INFO] 21%|##1       | 4/19 [00:00<00:01,  7.75it/s]
2023-08-08 01:01:29,546 [INFO] 26%|##6       | 5/19 [00:00<00:01,  7.75it/s]
2023-08-08 01:01:29,675 [INFO] 32%|###1      | 6/19 [00:00<00:01,  7.75it/s]
2023-08-08 01:01:29,806 [INFO] 37%|###6      | 7/19 [00:00<00:01,  7.71it/s]
2023-08-08 01:01:29,935 [INFO] 42%|####2     | 8/19 [00:01<00:01,  7.71it/s]
2023-08-08 01:01:30,065 [INFO] 47%|####7     | 9/19 [00:01<00:01,  7.72it/s]
2023-08-08 01:01:30,194 [INFO] 53%|#####2    | 10/19 [00:01<00:01,  7.73it/s]
2023-08-08 01:01:30,323 [INFO] 58%|#####7    | 11/19 [00:01<00:01,  7.74it/s]
2023-08-08 01:01:30,452 [INFO] 63%|######3   | 12/19 [00:01<00:00,  7.74it/s]
2023-08-08 01:01:30,582 [INFO] 68%|######8   | 13/19 [00:01<00:00,  7.73it/s]
2023-08-08 01:01:30,712 [INFO] 74%|#######3  | 14/19 [00:01<00:00,  7.72it/s]
2023-08-08 01:01:30,841 [INFO] 79%|#######8  | 15/19 [00:01<00:00,  7.73it/s]
2023-08-08 01:01:30,970 [INFO] 84%|########4 | 16/19 [00:02<00:00,  7.73it/s]
```

(continues on next page)

(continued from previous page)

```
2023-08-08 01:01:31,099 [INFO] 89%|########9 | 17/19 [00:02<00:00,  7.74it/s]
2023-08-08 01:01:31,228 [INFO] 95%|#########4| 18/19 [00:02<00:00,  7.74it/s]
2023-08-08 01:01:31,354 [INFO] 100%|##########| 19/19 [00:02<00:00,  7.79it/s]
2023-08-08 01:01:31,355 [INFO] 100%|##########| 19/19 [00:02<00:00,  7.74it/s]
2023-08-08 01:01:31,694 [INFO] running ZX: 443 planes of size (162, 207)
2023-08-08 01:01:31,732 [INFO] 0%|          | 0/14 [00:00<?, ?it/s]
2023-08-08 01:01:31,853 [INFO] 7%|7         | 1/14 [00:00<00:01,  8.28it/s]
2023-08-08 01:01:31,974 [INFO] 14%|#4        | 2/14 [00:00<00:01,  8.25it/s]
2023-08-08 01:01:32,096 [INFO] 21%|##1       | 3/14 [00:00<00:01,  8.24it/s]
2023-08-08 01:01:32,219 [INFO] 29%|##8       | 4/14 [00:00<00:01,  8.21it/s]
2023-08-08 01:01:32,341 [INFO] 36%|###5      | 5/14 [00:00<00:01,  8.21it/s]
2023-08-08 01:01:32,462 [INFO] 43%|####2     | 6/14 [00:00<00:00,  8.22it/s]
2023-08-08 01:01:32,584 [INFO] 50%|#####     | 7/14 [00:00<00:00,  8.22it/s]
2023-08-08 01:01:32,705 [INFO] 57%|#####7    | 8/14 [00:00<00:00,  8.22it/s]
2023-08-08 01:01:32,827 [INFO] 64%|######4   | 9/14 [00:01<00:00,  8.22it/s]
2023-08-08 01:01:32,948 [INFO] 71%|#######1  | 10/14 [00:01<00:00,  8.23it/s]
2023-08-08 01:01:33,070 [INFO] 79%|#######8  | 11/14 [00:01<00:00,  8.22it/s]
2023-08-08 01:01:33,191 [INFO] 86%|########5 | 12/14 [00:01<00:00,  8.22it/s]
2023-08-08 01:01:33,313 [INFO] 93%|#########2| 13/14 [00:01<00:00,  8.22it/s]
2023-08-08 01:01:33,432 [INFO] 100%|##########| 14/14 [00:01<00:00,  8.28it/s]
2023-08-08 01:01:33,432 [INFO] 100%|##########| 14/14 [00:01<00:00,  8.23it/s]
2023-08-08 01:01:34,940 [INFO] network run in 9.22s
dP_ times 10 for >2d, still experimenting
2023-08-08 01:01:38,310 [INFO] masks created in 3.37s
```

12.3.6 Compare masks to ground truth

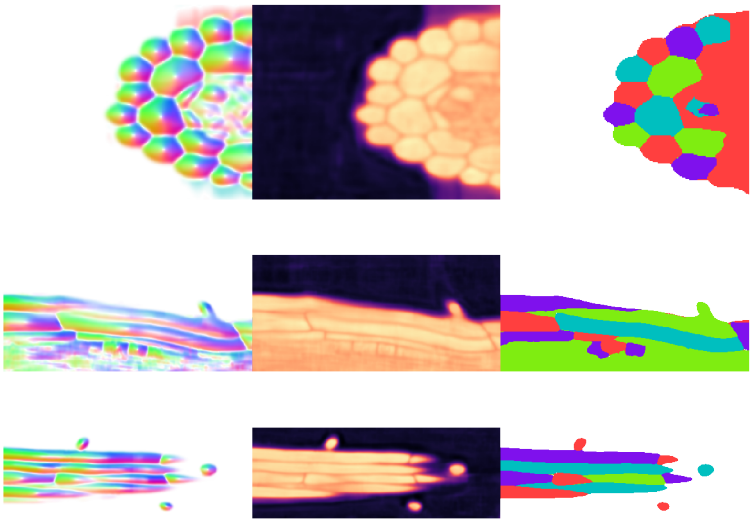
```
from fastremap import unique
mgt = io.imread(files[0][:-4]+'_masks.tif')
print('Cellpose do_3D + omni=True: {} masks. \nOmnipose 3D: {} masks. \nGround truth: {} masks'.format(len(unique(masks_cp[0])),
            len(unique(masks_om[0])),
            len(unique(mgt))))
```

```
Cellpose do_3D + omni=True: 55 masks.
Omnipose 3D: 204 masks.
Ground truth: 67 masks
```

For what it's worth, pure Cellpose gives ~550 masks in this volume, pure Omnipose gives ~200, and Cellpose model + Omnipose mask reconstruction gives ~50. I'm sorry to say that the ground truth for this dataset is quite bad, containing some undersegmented cells, but more importantly, an entire "ignore" region where there are many, many cells that are unlabeled. So the count of 67 cells in the ground truth refers only to the long cells on the outside of the root. Thus, 55 cells is a severe under-segmentation of the volume. Let's see why.

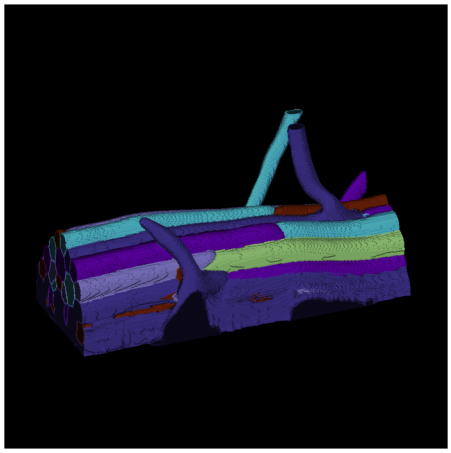
12.3.7 Plot results

```
from cellpose_omni import plot
from omnipose.plot import apply_ncolor

mu = flows_cp[0][1]
T = flows_cp[0][2]
bd = flows_cp[0][4]
# mu.shape,T.shape,bd.shape

d = mu.shape[0]

from omnipose.utils import rescale
c = np.array([1]*2+[0]*(d-2))
# c = np.arange(d)
def cyclic_perm(a):
    n = len(a)
    b = [[a[i - j] for i in range(n)] for j in range(n)]
    return b
slices = []
idx = np.arange(d)
cmap = mpl.colormaps['magma']
cmap2 = mpl.colormaps['viridis']

for inds in cyclic_perm(c):
    slc = tuple([slice(-1) if i else mu.shape[k+1]//2 for i,k in zip(inds,idx)])
    flow = plot.dx_to_circ(mu[np.where(inds)+slc],transparency=1)/255
    dist = cmap(rescale(T)[slc])
    msks = apply_ncolor(masks_cp[0][slc])

    fig = plt.figure(figsize=[5]*2,frameon=False)
    plt.imshow(np.hstack((flow,dist,msks)),interpolation='none')

    plt.axis('off')
    plt.show()
```



```
%%capture
import ncolor
mask = masks_cp[0]
mask_nc = ncolor.label(mask,max_depth=20)

import napari
viewer = napari.view_labels(mask_nc);
viewer.dims.ndisplay = 3
viewer.camera.center = [s//2 for s in mask.shape]
viewer.camera.zoom=1
viewer.camera.angles=(10.90517458968619, -20.777067798396835, 58.04311170773853)
viewer.camera.perspective=0.0
viewer.camera.interactive=True

img = viewer.screenshot(size=(1000,1000),scale=1,canvas_only=True,flash=False)
```

```
plt.figure(figsize=(3,3),frameon=False)
plt.imshow(img)
plt.axis('off')
plt.show()
```



It appears that **omni**=True does allow 2D Cellpose models to work in 3D, but the prediction quality - worsened by artifacts introduced by the compoisiting into 3D - is a limiting factor.

See settings for more information on algorithm parameters.

CHAPTER

THIRTEEN

COMMAND LINE

Running just `omnipose` in the command line interface will launch the *GUI*. I have left *training* new models - done exclusively via CLI - to its own page. The rest of this page refers to evaluation on the command line.

The command line allows batch processing and easy integration into downstream analysis pipelines like SuperSegger, Morphometrics, MicrobeJ, CellTool, and many others (any program that takes images and labels in directories). See *Settings* for an introduction to the settings. The command line interface accepts parameters from `cellpose_omni.models` for evaluation and from `cellpose_omni.io` for finding files and saving output.

13.1 How to segment images using CLI

Note: `omnipose` or `python -m omnipose` is equivalent to `python -m cellpose --omni`, as our fork of Cellpose still provides the main framework for running Omnipose.

Run `omnipose [arguments]` and specify the arguments as follows. For instance, to run on a folder with images where cytoplasm is green and nucleus is blue and save the output as a png (using default diameter 30):

```
omnipose --dir <img_dir> --pretrained_model cyto --chan 2 --chan2 3 --save_png
```

To do the same segmentation as in *mono_channel_bact.ipynb*, and save TIF masks (this turns off cp_output PNGs) to folders along with flows and outlines, run:

```
omnipose --dir <img_dir> --use_gpu --pretrained_model bact_phase_omni \
        --save_flows  --save_outlines --save_tif --in_folders
```

Rescaling for the `*bact*` models is disabled by default, but setting a diameter with the `--diameter` flag will rescale relative to 30px (*e.g.* `--diameter 15` would double the image in x and y before running the network).

Warning: The path given to `--dir` must be an absolute path.

13.2 Recommendations

There are some optional settings you should consider:

```
--dir_above --in_folders --save_tifs --save_flows --save_outlines --save_ncolor --no_npy
```

The `--no_npy` command just gets rid of the `.npy` output that many users do not need. `--save_tifs`, as an alternative to `--save_pngs`, does not save the four-panel plot output (that can take up a lot of space). Personally, I prefer to use `--save_outlines` when I want a whole folder of easy-to-visualize segmentation results and `--save_flows` when I want to debug them. These are also nice to have for making GIFs of cell growth, for example. `--save_ncolor` is handy for exporting *N-color* masks that are easier to edit by hand - but it is the 1-channel version, no RGB colormap applied (which is what you want for editing in Photoshop).

Most of all, `--in_folders` is something I always use so that these various outputs do not clutter up the image directory (`/image01.png`, `/image01_masks.tif`, `/image01_flows.tif`...) and instead dumps all the masks into a `/masks` folder, flows into `flows`, N-color masks into `/ncolor`, outlines into `/outlines`, and so on. Without the `--dir_above` command, these are inside the image directory. `--dir_above` will put those folders one directory above, parallel to the image directory, which is what I like and what SuperSegger expects.

`flow_threshold 0` is a very good idea if you have a lot of large images and do not need that cleanup step. Settings like `--mask_threshold 0.3` (0 is the default) can also be relevant. The *GUI* will automatically generate the parameters you need to recapitulate your results in CLI (just in notebook formatting for now - you will need to format those parameters according to these examples).

13.3 All options

You can print out the full list of features with `omnipose -h`. There are a lot of them, but with Omnipose we organized them into categories. See *CLI* to browse a bit easier. As demonstrated above, `input image arguments` and `output arguments` are the most relevant. See SuperSegger-Omnipose for an example of how to use these options to integrate Omnipose as a segmentation backend.

CHAPTER

FOURTEEN

API

This page exists to help users navigate the labyrinth of functions and classes that make up Omnipose.

14.1 Project structure

Omnipose is built on Cellpose, and functionally that means Cellpose actually imports Omnipose to replace many of its operations with the Omnipose versions with `omni=True`. Omnipose was first packaged into the Cellpose repo before I began making too many ND-generalizations (full rewrites) for the authors to maintain. Thus was birthed my `cellpose_omni` fork, which I published to PyPi separately from Omnipose for some time. I later decided that maintaining two packages for one project was overcomplicated for me and users (especially for installations from the repo), so the latest version of `cellpose_omni` now lives here. `cellpose_omni` still gets installed as its own subpackage when you install Omnipose. If you have issues migrating to the new version, make sure to `pip uninstall omnipose cellpose_omni` before re-installing Omnipose. The `install.py` script simply runs `pip install -e .{extras}` in the `omnipose` and `cellpose` directories.

If you encounter bugs with Omnipose, you can check the main Cellpose repo for related issues and also post them here. I do my best to keep up with with bug fixes and features from the main branch, but it helps me out a lot if users bring them to my attention. If there are any features or pull requests in Cellpose that you want to see in Omnipose ASAP, please let me know.

14.2 Modules

14.2.1 omnipose.core

affinity_to_boundary(masks, affinity_graph, ...)	Convert affinity graph to boundary map.
affinity_to_edges(affinity_graph, ...)	Convert symmetric affinity graph to list of edge tuples for connected components labeling.
affinity_to_masks(affinity_graph, ...[, ...])	Convert affinity graph to label matrix using connected components.
batch_labels(masks, bd, T, mu, tyx, dim, ...)	
boundary_to_affinity(masks, boundaries)	This function converts boundary+interior labels to an affinity graph.
boundary_to_masks(boundaries[, binary_mask, ...])	
compute_masks(dP, dist[, affinity_graph, ...])	Compute masks using dynamics from dP, dist, and boundary outputs.

continues on next page

Table 1 – continued from previous page

concatenate_labels(masks, links, nsample)	
diameters(masks[, dt, dist_threshold])	Calculate the mean cell diameter from a label matrix.
dist_to_diam(dt_pos, n)	Convert positive distance field values to a mean diameter.
div_rescale(dP, mask[, p])	Normalize the flow magnitude to rescaled 0-1 divergence.
divergence(f[, sp])	Computes divergence of vector field
divergence_torch(y)	
do_warp(A, M_inv, tyx[, offset, order, mode])	Wrapper function for affine transformations during augmentation.
fill_holes_and_remove_small_masks(masks[, ...])	fill holes in masks (2D/3D) and discard masks smaller than min_size (2D)
flow_error(maski, dP_net[, coords, ...])	error in flows from predicted masks vs flows predicted by network run on image
follow_flows(dP, dist, inds[, niter, ...])	define pixels and run dynamics to recover masks in 2D
get_boundary(mu, mask[, bd, affinity_graph, ...])	One way to get boundaries by considering flow dot products.
get_contour(labels, affinity_graph[, ...])	Sort 2D boundaries into cyclic paths.
get_link_matrix(links, piece_masks, inds, ...)	
get_links(masks, labels, bd[, connectivity])	
get_masks(p, bd, dist, mask, inds[, ...])	Omnipose mask recontruction algorithm.
get_masks_cp(p[, iscell, rpad, flows, ...])	create masks using pixel convergence after running dynamics
get_neigh_inds(coords, shape, steps)	For L pixels and S steps, find the neighboring pixel indexes 0,1,...,L for each step.
get_niter(dists)	Get number of iterations.
labels_to_flows(labels[, links, files, ...])	Convert labels (list of masks or flows) to flows for training model.
linker_label_to_links(maski, linker_label_list)	
links_to_boundary(masks, links)	Deprecated.
links_to_mask(masks, links)	Convert linked masks to stitched masks.
loss(self, lbl, y)	Loss function for Omnipose. :param lbl: transformed labels in array [nimg x nchan x xy[0] x xy[1]] lbl[:,0] cell masks lbl[:,1] thresholded mask layer lbl[:,2] boundary field lbl[:,3] smooth distance field lbl[:,4] boundary-emphasizing weights lbl[:,5:] flow components :type lbl: ND-array, float :param y: network predictions, with dimension D, these are: y[:,:D] flow field components at 0,1,...,D-1 y[:,D] distance fields at D y[:,D+1] boundary fields at D+1 :type y: ND-tensor, float.
masks_to_affinity(masks, coords, steps, ...)	Convert label matrix to affinity graph.
masks_to_flows(masks[, affinity_graph, ...])	Convert masks to flows.
masks_to_flows_batch(batch[, links, device, ...])	Batch process flows.
masks_to_flows_torch(masks, affinity_graph)	Convert ND masks to flows.
mode_filter(masks)	super fast mode filter (compared to scipy, idk about PIL) to clean up interpolated labels

continues on next page

Table 1 – continued from previous page

most_frequent(neighbor_masks)	
parametrize(steps, labs, unique_L, inds, ...)	Parametrize 2D boundaries.
parametrize_contours(steps, labs, unique_L, ...)	Helper function to sort 2D contours into cyclic paths.
random_crop_warp(img, Y, tyx, v1, v2, nchan, ...)	This sub-fuction of *random_rotate_and_resize()* recursively performs random cropping until a minimum number of cell pixels are found, then proceeds with augementations.
random_rotate_and_resize(X[, Y, ...])	augmentation by random rotation and resizing
remove_bad_flow_masks(masks, flows[, ...])	remove masks which have inconsistent flows
sigmoid(x)	The sigmoid function.
split_spacetime(augmented_affinity, mask[, ...])	Split lineage labels into frame-by-frame labels and Cell ID / spacetime labeling.
step_factor(t)	Euler integration suppression factor.
steps_batch(p, dP, niter[, omni, suppress, ...])	Euler integration of pixel locations p subject to flow dP for niter steps in N dimensions.

affinity_to_boundary

omnipose.core.**affinity_to_boundary**(*masks*, *affinity_graph*, *coords*)

Convert affinity graph to boundary map.

Internal hypervoxels are those that are fully connected to all their 3^D-1 neighbors, where D is the dimension. Boundary hypervoxels are those that are connected to fewer than this number and at least 1 other hypervoxel. Correct boundaries should have >=D connections, but the lower bound here is set to 1.

masks: ND array, int or binary
label matrix or binary foreground mask

affinity_graph: ND array, bool
hypervoxel affinity array, <3^D> by <number of foreground hypervoxels>

coords: tuple or ND array
coordinates of foreground hypervoxels, <dim>x<npix>

boundary

affinity_to_edges

omnipose.core.**affinity_to_edges**(*affinity_graph*, *neigh_inds*, *step_inds*, *px_inds*)

Convert symmetric affinity graph to list of edge tuples for connected components labeling.

affinity_to_masks

omnipose.core.**affinity_to_masks**(*affinity_graph*, *neigh_inds*, *iscell*, *coords*, *cardinal=True*, *exclude_interior=False*, *return_edges=False*, *verbose=False*)

Convert affinity graph to label matrix using connected components.

batch_labels

omnipose.core.**batch_labels**(*masks*, *bd*, *T*, *mu*, *tyx*, *dim*, *nclasses*, *device*, *dist_bg=5*)

boundary_to_affinity

omnipose.core.**boundary_to_affinity**(*masks*, *boundaries*)

This function converts boundary+interior labels to an affinity graph. Boundaries are taken to have label 1,2,...,N and interior pixels have some value M>N. This format is the best way I have found to annotate self-contact cells.

boundary_to_masks

omnipose.core.**boundary_to_masks**(*boundaries*, *binary_mask=None*, *min_size=9*, *dist=1.4142135623730951*, *connectivity=1*)

compute_masks

omnipose.core.**compute_masks**(*dP*, *dist*, *affinity_graph=None*, *bd=None*, *p=None*, *coords=None*, *iscell=None*, *niter=None*, *rescale=1.0*, *resize=None*, *mask_threshold=0.0*, *diam_threshold=12.0*, *flow_threshold=0.4*, *interp=True*, *cluster=False*, *boundary_seg=False*, *affinity_seg=False*, *do_3D=False*, *min_size=None*, *max_size=None*, *hole_size=None*, *omni=True*, *calc_trace=False*, *verbose=False*, *use_gpu=False*, *device=None*, *nclasses=2*, *dim=2*, *eps=None*, *hdbscan=False*, *flow_factor=6*, *debug=False*, *override=False*, *suppress=None*, *despur=True*)

Compute masks using dynamics from dP, dist, and boundary outputs. Called in cellpose.models().

Parameters

- **dP** (*float, ND array*) -- flow field components (2D: 2 x Ly x Lx, 3D: 3 x Lz x Ly x Lx)
- **dist** (*float, ND array*) -- distance field (Ly x Lx)
- **bd** (*float, ND array*) -- boundary field
- **p** (*float32, ND array*) -- initial locations of each pixel before dynamics, size [axis x Ly x Lx] or [axis x Lz x Ly x Lx].
- **coords** (*int32, 2D array*) -- non-zero pixels to run dynamics on [npixels x D]
- **niter** (*int32*) -- number of iterations of dynamics to run
- **rescale** (*float (optional, default None)*) -- resize factor for each image, if None, set to 1.0
- **resize** (*int, tuple*) -- shape of array (alternative to rescaling)
- **mask_threshold** (*float*) -- all pixels with value above threshold kept for masks, decrease to find more and larger masks
- **flow_threshold** (*float*) -- flow error threshold (all cells with errors below threshold are kept) (not used for Cellpose3D)
- **interp** (*bool*) -- interpolate during dynamics
- **cluster** (*bool*) -- use sub-pixel DBSCAN clustering of pixel coordinates to find masks

- **do_3D** (*bool (optional, default False)*) -- set to True to run 3D segmentation on 4D image input
- **min_size** (*int (optional, default 15)*) -- minimum number of pixels per mask, can turn off with -1
- **omni** (*bool*) -- use omnipose mask recontruction features
- **calc_trace** (*bool*) -- calculate pixel traces and return as part of the flow
- **verbose** (*bool*) -- turn on additional output to logs for debugging
- **use_gpu** (*bool*) -- use GPU of flow_threshold>0 (computes flows from predicted masks on GPU)
- **device** (*torch device*) -- what compute hardware to use to run the code (GPU VS CPU)
- **nclasses** -- number of output classes of the network (Omnipose=3,Cellpose=2)
- **dim** (*int*) -- dimensionality of data / model output
- **eps** (*float*) -- internal epsilon parameter for (H)DBSCAN
- **hdbscan** -- use better, but much SLOWER, hdbscan clustering algorithm (experimental)
- **flow_factor** -- multiple to increase flow magnitude (used in 3D only, experimental)
- **debug** -- option to return list of unique mask labels as a fourth output (for debugging only)

Returns

- **mask** (*int, ND array*) -- label matrix
- **p** (*float32, ND array*) -- final locations of each pixel after dynamics, size [axis x Ly x Lx] or [axis x Lz x Ly x Lx].
- **tr** (*float32, ND array*) -- intermediate locations of each pixel during dynamics, size [axis x niter x Ly x Lx] or [axis x niter x Lz x Ly x Lx]. For debugging/paper figures, very slow.
- **bd** (*float32, ND array*) -- boundary map
- **augmented_affinity** (*float32, ND array*) -- concatenated coordinates and affinity graph, hence (d+1,3**d,npix)

concatenate_labels

omnipose.core.**concatenate_labels**(*masks: ndarray*, *links: list*, *nsample: int*)

diameters

omnipose.core.**diameters**(*masks*, *dt=None*, *dist_threshold=0*)

Calculate the mean cell diameter from a label matrix.

Parameters

- **masks** (*ND array, float*) -- label matrix 0,...,N
- **dt** (*ND array, float*) -- distance field
- **dist_threshold** (*float*) -- cutoff below which all values in dt are set to 0. Must be >=0.

Returns

diam -- a single number that corresponds to the average diameter of labeled regions in the image, see dist_to_diam()

Return type
float

dist_to_diam

omnipose.core.**dist_to_diam**(*dt_pos*, *n*)

Convert positive distance field values to a mean diameter.

Parameters

- **dt_pos** (*1D array, float*) -- array of positive distance field values
- **n** (*int*) -- dimension of volume. dt_pos is always 1D because only the positive values int he distance field are passed in.

Returns
mean diameter -- a single number that corresponds to the diameter of the N-sphere when dt_pos for a sphere is given to the function, holds constant for extending rods of uniform width, much better than the diameter of a circle of equivalent area for estimating the short-axis dimensions of objects

Return type
float

div_rescale

omnipose.core.**div_rescale**(*dP*, *mask*, *p=1*)

Normalize the flow magnitude to rescaled 0-1 divergence.

Parameters

- **dP** (*float, ND array*) -- flow field
- **mask** (*int, ND array*) -- label matrix

Returns
dP -- rescaled flow field

Return type
float, ND array

divergence

omnipose.core.**divergence**(*f*, *sp=None*)

Computes divergence of vector field

Parameters

- **f** (*ND array, float*) -- vector field components [Fx,Fy,Fz,...]
- **sp** (*ND array, float*) -- spacing between points in respecitve directions [spx, spy, spz,...]

divergence_torch

omnipose.core.**divergence_torch**(*y*)

do_warp

omnipose.core.**do_warp**(*A*, *M_inv*, *tyx*, *offset=0*, *order=1*, *mode='constant'*, ***kwargs*)

Wrapper function for affine transformations during augmentation. Uses scipy.ndimage.affine_transform().

Parameters

- **A** (*NDarray, int or float*) -- input image to be transformed
- **M_inv** (*NDarray, float*) -- inverse tranformation matrix
- **order** (*int*) -- interpolation order, 1 is equivalent to 'nearest',

fill_holes_and_remove_small_masks

omnipose.core.**fill_holes_and_remove_small_masks**(*masks*, *min_size=None*, *max_size=None*, *hole_size=3*, *dim=2*)

fill holes in masks (2D/3D) and discard masks smaller than min_size (2D)

fill holes in each mask using scipy.ndimage.morphology.binary_fill_holes

Parameters

- **masks** (*int, 2D or 3D array*) -- labelled masks, 0=NO masks; 1,2,...=mask labels, size [Ly x Lx] or [Lz x Ly x Lx]
- **min_size** (*int (optional, default 3**dim)*) -- minimum number of pixels per mask (exclusive), can turn off with -1
- **max_size** (*int (optional, default None)*) -- maximum number of pixels per mask (exclusive)
- **hole_size** (*int (optional, default 3)*) -- holes bigger than this are NOT filled
- **dim** (*int (optional, default 2)*) -- dimension of the masks

Returns

masks -- masks with holes filled and masks smaller than min_size removed, 0=NO masks; 1,2,...=mask labels, size [Ly x Lx] or [Lz x Ly x Lx]

Return type

int, 2D or 3D array

flow_error

omnipose.core.**flow_error**(*maski*, *dP_net*, *coords=None*, *affinity_graph=None*, *use_gpu=False*, *device=None*, *omni=True*)

error in flows from predicted masks vs flows predicted by network run on image

This function serves to benchmark the quality of masks, it works as follows 1. The predicted masks are used to create a flow diagram 2. The mask-flows are compared to the flows that the network predicted

If there is a discrepancy between the flows, it suggests that the mask is incorrect. Masks with flow_errors greater than 0.4 are discarded by default. Setting can be changed in Cellpose.eval or CellposeModel.eval.

Parameters

- **maski** (*ND-array (int)*) -- masks produced from running dynamics on dP_net, where 0=NO masks; 1,2... are mask labels
- **dP_net** (*ND-array (float)*) -- ND flows where dP_net.shape[1:] = maski.shape

Returns

- **flow_errors** (*float array with length maski.max()*) -- mean squared error between predicted flows and flows from masks
- **dP_masks** (*ND-array (float)*) -- ND flows produced from the predicted masks

follow_flows

omnipose.core.**follow_flows**(*dP*, *dist*, *inds*, *niter=None*, *interp=True*, *use_gpu=True*, *device=None*, *omni=True*, *suppress=False*, *calc_trace=False*, *verbose=False*)

define pixels and run dynamics to recover masks in 2D

Pixels are meshgrid. Only pixels with non-zero cell-probability are used (as defined by inds)

Parameters

- **dP** (*float32, 3D or 4D array*) -- flows [axis x Ly x Lx] or [axis x Lz x Ly x Lx]
- **inds** (*int, ND array*) -- initial indices of pixels for the Euler integration
- **niter** (*int*) -- number of iterations of dynamics to run
- **interp** (*bool*) -- interpolate during dynamics
- **use_gpu** (*bool*) -- use GPU to run interpolated dynamics (faster than CPU)
- **omni** (*bool*) -- flag to enable Omnipose suppressed Euler integration etc.
- **calc_trace** (*bool*) -- flag to store and retrun all pixel coordinates during Euler integration (slow)

Returns

- **p** (*float32, ND array*) -- final locations of each pixel after dynamics
- **inds** (*int, ND array*) -- initial indices of pixels for the Euler integration [npixels x ndim]
- **tr** (*float32, ND array*) -- list of intermediate pixel coordinates for each step of the Euler integration

get_boundary

omnipose.core.**get_boundary**(*mu*, *mask*, *bd=None*, *affinity_graph=None*, *contour=False*, *use_gpu=False*, *device=None*, *desprue=False*)

One way to get boundaries by considering flow dot products. Will be deprecated.

get_contour

omnipose.core.**get_contour**(*labels*, *affinity_graph*, *coords=None*, *neighbors=None*, *cardinal_only=True*)

Sort 2D boundaries into cyclic paths.

labels: 2D array, int
label matrix

affinity_graph: 2D array, bool
pixel affinity array, 9 by number of foreground pixels

get_link_matrix

omnipose.core.**get_link_matrix**(*links*, *piece_masks*, *inds*, *idx*, *is_link*)

get_links

omnipose.core.**get_links**(*masks*, *labels*, *bd*, *connectivity=1*)

get_masks

omnipose.core.**get_masks**(*p*, *bd*, *dist*, *mask*, *inds*, *nclasses=2*, *cluster=False*, *diam_threshold=12.0*, *eps=None*, *hdbscan=False*, *verbose=False*)

Omnipose mask recontruction algorithm.

This function is called after dynamics are run. The final pixel coordinates are provided, and cell labels are assigned to clusters found by labeling the pixel clusters after rounding the coordinates (snapping each pixel to the grid and labeling the resulting binary mask) or by using DBSCAN or HDBSCAN for sub-pixel clustering.

Parameters

- **p** (*float32, ND array*) -- final locations of each pixel after dynamics
- **bd** (*float, ND array*) -- boundary field
- **dist** (*float, ND array*) -- distance field
- **mask** (*bool, ND array*) -- binary cell mask
- **inds** (*int, ND array*) -- initial indices of pixels for the Euler integration [npixels x ndim]
- **nclasses** (*int*) -- number of prediciton classes
- **cluster** (*bool*) -- use DBSCAN clustering instead of coordinate thresholding
- **diam_threshold** (*float*) -- mean diameter under which clustering will be turned on automatically
- **eps** (*float*) -- internal espilon parameter for (H)DBSCAN
- **hdbscan** (*bool*) -- use better, but much SLOWER, hdbscan clustering algorithm
- **verbose** (*bool*) -- option to print more info to log file

Returns

- **mask** (*int, ND array*) -- label matrix
- **labels** (*int, list*) -- all unique labels

get_masks_cp

omnipose.core.**get_masks_cp**(*p*, *iscell=None*, *rpad=20*, *flows=None*, *use_gpu=False*, *device=None*)

create masks using pixel convergence after running dynamics

Makes a histogram of final pixel locations p, initializes masks at peaks of histogram and extends the masks from the peaks so that they include all pixels with more than 2 final pixels p. Discards masks with flow errors greater than the threshold.

Parameters

- **p** (*float32, 3D or 4D array*) -- final locations of each pixel after dynamics, size [axis x Ly x Lx] or [axis x Lz x Ly x Lx].
- **iscell** (*bool, 2D or 3D array*) -- if iscell is not None, set pixels that are iscell False to stay in their original location.
- **rpad** (*int (optional, default 20)*) -- histogram edge padding
- **flows** (*float, 3D or 4D array (optional, default None)*) -- flows [axis x Ly x Lx] or [axis x Lz x Ly x Lx]. If flows is not None, then masks with inconsistent flows are removed using *remove_bad_flow_masks*.

Returns

M0 -- masks with inconsistent flow masks removed, 0=NO masks; 1,2,...=mask labels, size [Ly x Lx] or [Lz x Ly x Lx]

Return type

int, 2D or 3D array

get_neigh_inds

omnipose.core.**get_neigh_inds**(*coords*, *shape*, *steps*)

For L pixels and S steps, find the neighboring pixel indexes 0,1,...,L for each step. Background index is -1. Returns:

Parameters

- **coords** (*tuple or ND array*) -- coordinates of nonzero pixels, <dim>x<npix>
- **shape** (*tuple or list, int*) -- shape of the image array
- **steps** (*ND array, int*) -- list or array of ND steps to neighbors

Returns

- **indexes** (*1D array*) -- list of pixel indexes 0,1,...L-1
- **neigh_inds** (*2D array*) -- SxL array corresponding to affinity graph
- **ind_matrix** (*ND array*) -- indexes inserted into the ND image volume

get_niter

omnipose.core.**get_niter**(*dists*)

Get number of iterations.

Parameters
dists (*ND array, float*) -- array of (nonnegative) distance field values

Returns
niter -- number of iterations empirically found to be the lower bound for convergence of the distance field relaxation method

Return type
int

labels_to_flows

omnipose.core.**labels_to_flows**(*labels*, *links=None*, *files=None*, *use_gpu=False*, *device=None*, *omni=True*, *redo_flows=False*, *dim=2*)

Convert labels (list of masks or flows) to flows for training model.

if files is not None, flows are saved to files to be reused

Parameters

- **labels** (*list of ND-arrays*) -- labels[k] can be 2D or 3D, if [3 x Ly x Lx] then it is assumed that flows were precomputed. Otherwise labels[k][0] or labels[k] (if 2D) is used to create flows.
- **links** (*list of label links*) -- These lists of label pairs define which labels are "linked", i.e. should be treated as part of the same object. This is how Omnipose handles internal/self-contact boundaries during training.
- **files** (*list of strings*) -- list of file names for the base images that are appended with '_flows.tif' for saving.
- **use_gpu** (*bool*) -- flag to use GPU for speedup. Note that Omnipose fixes some bugs that caused the Cellpose GPU implementation to have different behavior compared to the Cellpose CPU implementation.
- **device** (*torch device*) -- what compute hardware to use to run the code (GPU VS CPU)
- **omni** (*bool*) -- flag to generate Omnipose flows instead of Cellpose flows
- **redo_flows** (*bool*) -- flag to overwrite existing flows. This is necessary when changing over from Cellpose to Omnipose, as the flows are very different.
- **dim** (*int*) -- integer representing the intrinsic dimensionality of the data. This allows users to generate 3D flows for volumes. Some dependencies will need to be to be extended to allow for 4D, but the image and label loading is generalized to ND.

Returns
flows -- flows[k][0] is labels[k], flows[k][1] is cell distance transform, flows[k][2:2+dim] are the (T)YX flow components, and flows[k][-1] is heat distribution / smooth distance

Return type
list of [4 x Ly x Lx] arrays

linker_label_to_links

omnipose.core.**linker_label_to_links**(*maski*, *linker_label_list*)

links_to_boundary

omnipose.core.**links_to_boundary**(*masks*, *links*)

Deprecated. Use masks_to_affinity instead.

links_to_mask

omnipose.core.**links_to_mask**(*masks*, *links*)

Convert linked masks to stitched masks.

loss

omnipose.core.**loss**(*self*, *lbl*, *y*)

Loss function for Omnipose. :param lbl: transformed labels in array [nimg x nchan x xy[0] x xy[1]]

lbl[:,0] cell masks lbl[:,1] thresholded mask layer lbl[:,2] boundary field lbl[:,3] smooth distance field lbl[:,4] boundary-emphasizing weights lbl[:,5:] flow components

Parameters

y (*ND-tensor, float*) -- network predictions, with dimension D, these are: y[:,:D] flow field components at 0,1,...,D-1 y[:,D] distance fields at D y[:,D+1] boundary fields at D+1

masks_to_affinity

omnipose.core.**masks_to_affinity**(*masks*, *coords*, *steps*, *inds*, *idx*, *fact*, *sign*, *dim*, *links=None*, *edges=None*, *dists=None*, *cutoff=1.4142135623730951*)

Convert label matrix to affinity graph. Here the affinity graph is an NxM matrix, where N is the number of possible hypercube connections (3**dimension) and M is the number of foreground hypervoxels. Self-connections are set to 0.

idx is the central index of the kernel, inds[0]. edges is a list of tuples (y1,y2,y3,...),(x1,x2,x3,...) etc. to which all adjacent pixels should be connected concatenated masks should be paddedby 1 to make sure that doesn't cause unextpected label merging dist can be used instead for edge connectivity

masks_to_flows

omnipose.core.**masks_to_flows**(*masks*, *affinity_graph=None*, *dists=None*, *coords=None*, *links=None*, *use_gpu=True*, *device=None*, *omni=True*, *dim=2*, *smooth=False*, *normalize=False*, *n_iter=None*, *verbose=False*)

Convert masks to flows.

First, we find the scalar field. In Omnipose, this is the distance field. In Cellpose, this is diffusion from center pixel. Center of masks where diffusion starts is defined to be the closest pixel to the median of all pixels that is inside the mask.

The flow components are then found as hthe gradient of the scalar field.

Parameters

- **masks** (*int, ND array*) -- labeled masks, 0 = background, 1,2,...,N = mask labels
- **dists** (*ND array, float*) -- array of (nonnegative) distance field values
- **affinity_graph** (*ND array, bool*) -- hypervoxel affinity array, alternative to providing overseg labels and links the most general way to compute flows, and can represent internal boundaries
- **links** (*list of label links*) -- list of tuples used for treating label pairs as the same
- **use_gpu** (*bool*) -- flag to use GPU for speedup. Note that Omnipose fixes some bugs that caused the Cellpose GPU implementation to have different behavior compared to the Cellpose CPU implementation.
- **device** (*torch device*) -- what compute hardware to use to run the code (GPU VS CPU)
- **omni** (*bool*) -- flag to generate Omnipose flows instead of Cellpose flows
- **dim** (*int*) -- dimensionality of image data

Returns

- **mu** (*float, 3D or 4D array*) -- flows in Y = mu[-2], flows in X = mu[-1]. if masks are 3D, flows in Z = mu[0].
- **mu_c** (*float, 2D or 3D array*) -- for each pixel, the distance to the center of the mask in which it resides

masks_to_flows_batch

omnipose.core.**masks_to_flows_batch**(*batch*, *links=[None]*, *device=device(type='cpu')*, *omni=True*, *dim=2*, *smooth=False*, *normalize=False*, *affinity_field=False*, *initialize=False*, *n_iter=None*, *verbose=False*)

Batch process flows. This includes padding with relection to not have weird cutoff flows.

Parameters
mask_batch (*list, NDarray*) -- list of masks all of shape tyx

Return type
concatenated labels, links, etc. and slices to extract them

masks_to_flows_torch

omnipose.core.**masks_to_flows_torch**(*masks*, *affinity_graph*, *coords=None*, *dists=None*, *device=device(type='cpu')*, *omni=True*, *affinity_field=False*, *smooth=False*, *normalize=False*, *n_iter=None*, *weight=1*, *return_flows=True*, *edges=None*, *initialize=False*, *verbose=False*)

Convert ND masks to flows.

Omnipose find distance field, Cellpose uses diffusion from center of mass.

Parameters

- **masks** (*int, ND array*) -- labelled masks, 0 = background, 1,2,...,N = mask labels
- **dists** (*ND array, float*) -- array of (nonnegative) distance field values
- **device** (*torch device*) -- what compute hardware to use to run the code (GPU VS CPU)

- **omni** (*bool*) -- flag to generate Omnipose flows instead of Cellpose flows
- **smooth** (*bool*) -- use relaxation to smooth out distance and therby flow field
- **n_iter** (*int*) -- override number of iterations

Returns

- **mu** (*float, 3D or 4D array*) -- flows in Y = mu[-2], flows in X = mu[-1]. if masks are 3D, flows in Z or T = mu[0].
- **dist** (*float, 2D or 3D array*) -- scalar field representing temperature distribution (Cellpose) or the smooth distance field (Omnipose)

mode_filter

omnipose.core.**mode_filter**(*masks*)

super fast mode filter (compared to scipy, idk about PIL) to clean up interpolated labels

most_frequent

omnipose.core.**most_frequent**(*neighbor_masks*)

parametrize

omnipose.core.**parametrize**(*steps*, *labs*, *unique_L*, *inds*, *ind_shift*, *values*, *step_ok*)

Parametrize 2D boundaries.

parametrize_contours

omnipose.core.**parametrize_contours**(*steps*, *labs*, *unique_L*, *neigh_inds*, *step_ok*, *csum*)

Helper function to sort 2D contours into cyclic paths. See get_contour().

random_crop_warp

omnipose.core.**random_crop_warp**(*img*, *Y*, *tyx*, *v1*, *v2*, *nchan*, *rescale*, *scale_range*, *gamma_range*, *do_flip*, *ind*, *do_labels=True*, *depth=0*)

This sub-fuction of *random_rotate_and_resize()* recursively performs random cropping until a minimum number of cell pixels are found, then proceeds with augemntations.

Parameters

- **X** (*float, list of ND arrays*) -- image array of size [nchan x Lt x Ly x Lx] or [Lt x Ly x Lx]
- **Y** (*float, ND array*) -- image label array of size [nlabels x Lt x Ly x Lx] or [Lt x Ly x Lx].. The 1st channel of Y is always nearest-neighbor interpolated (assumed to be masks or 0-1 representation). If Y.shape[0]==3, then the labels are assumed to be [cell probability, T flow, Y flow, X flow].
- **tyx** (*int, tuple*) -- size of transformed images to return, e.g. (Ly,Lx) or (Lt,Ly,Lx)
- **nchan** (*int*) -- number of channels the images have

- **rescale** (*float, array or list*) -- how much to resize images by before performing augmentations
- **scale_range** (*float*) -- Range of resizing of images for augmentation. Images are resized by (1-scale_range/2) + scale_range * np.random.rand()
- **gamma_range** (*float, list*) -- images are gamma-adjusted im**gamma for gamma in [low,high]
- **do_flip** (*bool (optional, default True)*) -- whether or not to flip images horizontally
- **ind** (*int*) -- image index (for debugging)
- **dist_bg** (*float*) -- nonegative value X for assigning -X to where distance=0 (deprecated, now adapts to field values)
- **depth** (*int*) -- how many time this function has been called on an image

Returns

- **imgi** (*float, ND array*) -- transformed images in array [nchan x xy[0] x xy[1]]
- **lbl** (*float, ND array*) -- transformed labels in array [nchan x xy[0] x xy[1]]
- **scale** (*float, 1D array*) -- scalar by which the image was resized

random_rotate_and_resize

omnipose.core.**random_rotate_and_resize**(*X*, *Y=None*, *scale_range=1.0*, *gamma_range=[0.75, 2.5]*, *tyx=(224, 224)*, *do_flip=True*, *rescale=None*, *inds=None*, *nchan=1*)

augmentation by random rotation and resizing

X and Y are lists or arrays of length nimg, with channels x Lt x Ly x Lx (channels optional, Lt only in 3D)

Parameters

- **X** (*float, list of ND arrays*) -- list of image arrays of size [nchan x Lt x Ly x Lx] or [Lt x Ly x Lx]
- **Y** (*float, list of ND arrays*) -- list of image labels of size [nlabels x Lt x Ly x Lx] or [Lt x Ly x Lx]. The 1st channel of Y is always nearest-neighbor interpolated (assumed to be masks or 0-1 representation). If Y.shape[0]==3, then the labels are assumed to be [distance, T flow, Y flow, X flow].
- **links** (*list of label links*) -- lists of label pairs linking parts of multi-label object together this is how omnipose gets around boudary artifacts druing image warps
- **scale_range** (*float (optional, default 1.0)*) -- Range of resizing of images for augmentation. Images are resized by (1-scale_range/2) + scale_range * np.random.rand()
- **gamma_range** (*float, list*) -- images are gamma-adjusted im**gamma for gamma in [low,high]
- **tyx** (*int, tuple*) -- size of transformed images to return, e.g. (Ly,Lx) or (Lt,Ly,Lx)
- **do_flip** (*bool (optional, default True)*) -- whether or not to flip images horizontally
- **rescale** (*float, array or list*) -- how much to resize images by before performing augmentations

- **inds** (*int, list*) -- image indices (for debugging)
- **nchan** (*int*) -- number of channels the images have

Returns

- **imgi** (*float, ND array*) -- transformed images in array [nimg x nchan x xy[0] x xy[1]]
- **lbl** (*float, ND array*) -- transformed labels in array [nimg x nchan x xy[0] x xy[1]]
- **scale** (*float, 1D array*) -- scalar(s) by which each image was resized

remove_bad_flow_masks

omnipose.core.**remove_bad_flow_masks**(*masks*, *flows*, *coords=None*, *affinity_graph=None*, *threshold=0.4*, *use_gpu=False*, *device=None*, *omni=True*)

remove masks which have inconsistent flows

Uses metrics.flow_error to compute flows from predicted masks and compare flows to predicted flows from network. Discards masks with flow errors greater than the threshold.

Parameters

- **masks** (*int, 2D or 3D array*) -- labelled masks, 0=NO masks; 1,2,...=mask labels, size [Ly x Lx] or [Lz x Ly x Lx]
- **flows** (*float, 3D or 4D array*) -- flows [axis x Ly x Lx] or [axis x Lz x Ly x Lx]
- **threshold** (*float*) -- masks with flow error greater than threshold are discarded

Returns

masks -- masks with inconsistent flow masks removed, 0=NO masks; 1,2,...=mask labels, size [Ly x Lx] or [Lz x Ly x Lx]

Return type

int, 2D or 3D array

sigmoid

omnipose.core.**sigmoid**(*x*)

The sigmoid function.

split_spacetime

omnipose.core.**split_spacetime**(*augmented_affinity*, *mask*, *verbose=False*)

Split lineage labels into frame-by-frame labels and Cell ID / spacetime labeling.

step_factor

omnipose.core.**step_factor**(*t*)

Euler integration suppression factor.

Conveneient wrapper function allowed me to test out several supression factors.

Parameters
t (*int*) -- time step

steps_batch

omnipose.core.**steps_batch**(*p*, *dP*, *niter*, *omni=True*, *suppress=True*, *interp=True*, *calc_trace=False*, *calc_bd=False*, *verbose=False*)

Euler integration of pixel locations p subject to flow dP for niter steps in N dimensions.

Parameters

- **p** (*float32, tensor*) -- pixel locations [axis x Lz x Ly x Lx] (start at initial meshgrid)
- **dP** (*float32, ND array*) -- flows [axis x Lz x Ly x Lx]
- **niter** (*int32*) -- number of iterations of dynamics to run

Returns
p -- final locations of each pixel after dynamics

Return type
float32, ND array

14.2.2 omnipose.utils

add_gaussian_noise(image[, mean, var])	
add_poisson_noise(image)	
apply_gaussian_blur(image, kernel_size, sigma)	Applies a Gaussian blur to the image.
apply_shifts(moving_images, shifts)	
auto_chunked_quantile(tensor, q)	
average_tiles_ND(y, subs, shape)	average results of network over tiles
bbox_to_slice(bbox, shape[, pad, im_pad])	return the tuple of slices for cropping an image based on the skimage.measure bounding box optional padding allows for the bounding box to be expanded, but not outside the original image dimensions
border_indices(tyx)	Return flat indices of border values in ND.
clean_boundary(labels[, boundary_thickness, ...])	Delete boundary masks below a given size threshold within a certain distance from the boundary.
correct_illumination(img[, sigma])	
crop_bbox(mask[, pad, iterations, im_pad, ...])	Take a label matrix and return a list of bounding boxes identifying clusters of labels.

continues on next page

Table 2 – continued from previous page

cross_reg(imstack[, upsample_factor, order, ...])	Find the transformation matrices for all images in a time series to align to the beginning frame.
cubestats(n)	Gets the number of m-dimensional hypercubes connected to the n-cube, including itself.
curve_filter(im[, filterWidth])	curveFilter : calculates the curvatures of an image.
extract_skeleton(distance_field)	
find_files(directory, suffix[, exclude_suffixes])	
find_nonzero_runs(a)	
findbetween(s[, string1, string2])	Find text between string1 and string2.
gaussian_kernel(size, sigma[, device])	Creates a 2D Gaussian kernel with mean 0.
generate_slices(image_shape, crop_size)	Generate slices for cropping an image into crops of size crop_size.
get_boundary(mask)	ND binary mask boundary using mahotas.
get_edge_masks(labels, dists)	Finds and returns masks that are largely cut off by the edge of the image.
get_flip(idx)	
get_module(x)	
get_neigh_inds(neighbors, coords, shape[, ...])	For L pixels and S steps, find the neighboring pixel indexes 0,1,...,L for each step.
get_neighbors(coords, steps, dim, shape[, ...])	Get the coordinates of all neighbor pixels.
get_neighbors_torch(input, steps)	This version not yet used/tested.
get_spruepoints(bw)	
get_steps(dim)	Get a symmetrical list of all 3**N points in a hypercube represented by a list of all possible sequences of -1, 0, and 1 in ND.
getname(path[, prefix, suffix, padding])	Extract the file name.
hysteresis_threshold(image, low, high)	Pytorch implementation of skimage.filters.apply_hysteresis_threshold().
is_integer(var)	
kernel_setup(dim)	Get relevant kernel information for the hypercube of interest.
load_nested_list(file_path)	Helper function to load affinity graphs.
localnormalize(im[, sigma1, sigma2])	
localnormalize_GPU(im[, sigma1, sigma2])	
make_tiles_ND(imgi[, bsize, augment, ...])	make tiles of image to run at test-time
make_unique(masks)	Relabel stack of label matrices such that there is no repeated label across slices.
mask_outline_overlay(img, masks, outlines[, ...])	Apply a color overlay to a grayscale image based on a label matrix.
mono_mask_bd(masks, outlines[, color, a])	

continues on next page

Table 2 – continued from previous page

moving_average(x, w)	
normalize99(Y[, lower, upper, ...])	normalize array/tensor so 0.0 is 0.01st percentile and 1.0 is 99.99th percentile Upper and lower percentile ranges configurable.
normalize_field(mu[, use_torch, cutoff])	normalize all nonzero field vectors to magnitude 1
normalize_image(im, mask[, target, ...])	Normalize image by rescaling from 0 to 1 and then adjusting gamma to bring average background to specified value (0.5 by default).
normalize_stack(vol, mask[, bg, ...])	Adjust image stacks so that background is (1) consistent in brightness and (2) brought to an even average via semantic gamma normalization.
phase_cross_correlation_GPU(image_stack[, ...])	
phase_cross_correlation_GPU_old(image_stack)	
ravel_index(b, shp)	
remap_pairs(pairs, replacements)	
rescale(T[, floor, ceiling, exclude_dims])	Rescale data between 0 and 1.
rotate(V, theta[, order, output_shape, center])	
safe_divide(num, den[, cutoff])	Division ignoring zeros and NaNs in the denominator.
save_nested_list(file_path, nested_list)	Helper function to save affinity graphs.
shift_stack(imstack, shift_vectors[, order, ...])	Shift each time slice of imstack according to list of nD shifts.
shifts_to_slice(shifts, shape)	Find the minimal crop box from time lapse registration shifts.
steps_to_indices(steps)	Get indices of the hupercubes sharing m-faces on the central n-cube.
subsample_affinity(augmented_affinity, slc, mask)	Helper function to subsample an affinity graph according to an image crop slice and a foreground selection mask.
thin_skeleton(image)	
to_16_bit(im)	Rescale image [0,2^16-1] and then cast to uint16.
to_8_bit(im)	Rescale image [0,2^8-1] and then cast to uint8.
torch_norm(a[, dim, keepdim])	Wrapper for torch.linalg.norm to handle ARM architecture.
unaugment_tiles_ND(y, inds[, unet])	reverse test-time augmentations for averaging
unravel_index(index, shape)	

add_gaussian_noise

omnipose.utils.**add_gaussian_noise**(*image*, *mean=0*, *var=0.01*)

add_poisson_noise

omnipose.utils.**add_poisson_noise**(*image*)

apply_gaussian_blur

omnipose.utils.**apply_gaussian_blur**(*image*, *kernel_size*, *sigma*, *device=device(type='cuda')*)

Applies a Gaussian blur to the image.

Parameters

- **image** (*torch.Tensor*) -- The image to blur.
- **kernel_size** (*int*) -- The size of the Gaussian kernel.
- **sigma** (*float*) -- The standard deviation of the Gaussian distribution.

Returns
The blurred image.

Return type
torch.Tensor

apply_shifts

omnipose.utils.**apply_shifts**(*moving_images*, *shifts*)

auto_chunked_quantile

omnipose.utils.**auto_chunked_quantile**(*tensor*, *q*)

average_tiles_ND

omnipose.utils.**average_tiles_ND**(*y*, *subs*, *shape*)

average results of network over tiles

Parameters

- **y** (*float, [ntiles x nclasses x bsize x bsize]*) -- output of cellpose network for each tile
- **subs** (*list*) -- list of slices for each subtile
- **shape** (*int, list or tuple*) -- shape of pre-tiled image (may be larger than original image if image size is less than bsize)

Returns
yf -- network output averaged over tiles

Return type
float32, [nclasses x Ly x Lx]

bbox_to_slice

omnipose.utils.**bbox_to_slice**(*bbox*, *shape*, *pad=0*, *im_pad=0*)

return the tuple of slices for cropping an image based on the skimage.measure bounding box optional padding allows for the bounding box to be expanded, but not outside the original image dimensions

Parameters

- **bbox** (*ndarray, float*) -- input bounding box, e.g. [y0,x0,y1,x1]
- **shape** (*array, tuple, or list, int*) -- shape of corresponding array to be sliced
- **pad** (*array, tuple, or list, int*) -- padding to be applied to each axis of the bounding box can be a common padding (5 means 5 on every side) or a list of each axis padding ([3,4] means 3 on y and 4 on x). N-volume requires an N-tuple.
- **im_pad** (*int*) -- region around the edges to avoid (pull back coordinate limits)

Return type
tuple of slices

border_indices

omnipose.utils.**border_indices**(*tyx*)

Return flat indices of border values in ND. Use via A.flat[border_indices].

clean_boundary

omnipose.utils.**clean_boundary**(*labels*, *boundary_thickness=3*, *area_thresh=30*, *cutoff=0.5*)

Delete boundary masks below a given size threshold within a certain distance from the boundary.

Parameters

- **boundary_thickness** (*int*) -- labels within a stripe of this thickness along the boundary will be candidates for removal.
- **area_thresh** (*int*) -- labels with area below this value will be removed.
- **cutoff** (*float*) -- Fraction from 0 to 1 of the overlap with the boundary before the mask is removed. Default 0.5. Set to 0 if you want any mask touching the boundary to be removed.

Return type
label matrix with small edge labels removed.

correct_illumination

omnipose.utils.**correct_illumination**(*img*, *sigma=5*)

crop_bbox

omnipose.utils.**crop_bbox**(*mask*, *pad=10*, *iterations=3*, *im_pad=0*, *area_cutoff=0*, *max_dim=inf*, *get_biggest=False*, *binary=False*)

Take a label matrix and return a list of bounding boxes identifying clusters of labels.

Parameters

- **mask** (*matrix of integer labels*) --
- **pad** (*amount of space in pixels to add around the label (does not extend beyond image edges, will shrink for consistency)*) --
- **iterations** (*number of dilation iterations to merge labels separated by this number of pixel or less*) --
- **im_pad** (*amount of space to subtract off the label matrix edges*) --
- **area_cutoff** (*label clusters below this area in square pixels will be ignored*) --
- **max_dim** (*if a cluster is above this cutoff, quit and return the original image bounding box*) --

Returns
slices

Return type
list of bounding box slices with padding

cross_reg

omnipose.utils.**cross_reg**(*imstack*, *upsample_factor=100*, *order=1*, *normalization=None*, *reverse=True*, *localnorm=True*)

Find the transformation matrices for all images in a time series to align to the beginning frame.

cubestats

omnipose.utils.**cubestats**(*n*)

Gets the number of m-dimensional hypercubes connected to the n-cube, including itself.

Parameters
n (*int*) -- dimension of hypercube

Returns

- *List whose length tells us how many hypercube types there are (point/edge/pixel/voxel...)*
- *connected to the central hypercube and whose entries denote many there in each group.*
- *E.g., a square would be n=2, so cubestats returns [4, 4, 1] for four points (m=0),*
- *four edges (m=1), and one face (the original square,m=n=2).*

curve_filter

omnipose.utils.**curve_filter**(*im*, *filterWidth=1.5*)

curveFilter : calculates the curvatures of an image.

INPUT :

im : image to be filtered filterWidth : filter width

OUTPUT :

M_ : Mean curvature of the image without negative values G_ : Gaussian curvature of the image without negative values C1_ : Principal curvature 1 of the image without negative values C2_ : Principal curvature 2 of the image without negative values M : Mean curvature of the ima ge G : Gaussian curvature of the image C1 : Principal curvature 1 of the image C2 : Principal curvature 2 of the image im_xx : del^2 x / del x^2 im_yy : del^2 x / del y^2 im_xy : del^2 x / del x del y

extract_skeleton

omnipose.utils.**extract_skeleton**(*distance_field*)

find_files

omnipose.utils.**find_files**(*directory*, *suffix*, *exclude_suffixes=[]*)

find_nonzero_runs

omnipose.utils.**find_nonzero_runs**(*a*)

findbetween

omnipose.utils.**findbetween**(*s*, *string1='['*, *string2=']'*)

Find text between string1 and string2.

gaussian_kernel

omnipose.utils.**gaussian_kernel**(*size: int*, *sigma: float*, *device=device(type='cuda')*)

Creates a 2D Gaussian kernel with mean 0.

Parameters

- **size** (*int*) -- The size of the kernel. Should be an odd number.
- **sigma** (*float*) -- The standard deviation of the Gaussian distribution.

Returns

The Gaussian kernel.

Return type

torch.Tensor

generate_slices

omnipose.utils.**generate_slices**(*image_shape*, *crop_size*)

Generate slices for cropping an image into crops of size crop_size.

get_boundary

omnipose.utils.**get_boundary**(*mask*)

ND binary mask boundary using mahotas.

Parameters
mask (*ND array, bool*) -- binary mask

Return type
Binary boundary map

get_edge_masks

omnipose.utils.**get_edge_masks**(*labels*, *dists*)

Finds and returns masks that are largely cut off by the edge of the image.

This function loops over all masks touching the image boundary and compares the maximum value of the distance field along the boundary to the top quartile of distance within the mask. Regions whose edges just skim the image edge will not be classified as an "edge mask" by this criteria, whereas masks cut off in their center (where distance is high) will be returned as part of this output.

Parameters

- **labels** (*ND array, int*) -- label matrix
- **dists** (*ND array, float*) -- distance field (calculated with reflection padding of labels)

Returns
clean_labels -- label matrix of all cells qualifying as 'edge masks'

Return type
ND array, int

get_flip

omnipose.utils.**get_flip**(*idx*)

get_module

omnipose.utils.**get_module**(*x*)

get_neigh_inds

omnipose.utils.**get_neigh_inds**(*neighbors*, *coords*, *shape*, *background_reflect=False*)

For L pixels and S steps, find the neighboring pixel indexes 0,1,...,L for each step. Background index is -1. Returns:

Parameters

- **coords** (*tuple, int*) -- coordinates of nonzero pixels, <dim>x<npix>
- **shape** (*tuple, int*) -- shape of the image array

Returns

- **indexes** (*1D array*) -- list of pixel indexes 0,1,...L-1
- **neigh_inds** (*2D array*) -- SxL array corresponding to affinity graph
- **ind_matrix** (*ND array*) -- indexes inserted into the ND image volume

get_neighbors

omnipose.utils.**get_neighbors**(*coords*, *steps*, *dim*, *shape*, *edges=None*, *pad=0*)

Get the coordinates of all neighbor pixels. Coordinates of pixels that are out-of-bounds get clipped.

get_neighbors_torch

omnipose.utils.**get_neighbors_torch**(*input*, *steps*)

This version not yet used/tested.

get_spruepoints

omnipose.utils.**get_spruepoints**(*bw*)

get_steps

omnipose.utils.**get_steps**(*dim*)

Get a symmetrical list of all 3**N points in a hypercube represented by a list of all possible sequences of -1, 0, and 1 in ND.

1D: [[-1],[0],[1]] 2D: [[-1, -1],

[-1, 0], [-1, 1], [0, -1], [0, 0], [0, 1], [1, -1], [1, 0], [1, 1]]

The opposite pixel at index i is always found at index -(i+1). The number of possible face, edge, vertex, etc. connections grows exponentially with dimension: 3 steps in 1D, 9 steps in 3D, 3**N in ND.

getname

omnipose.utils.**getname**(*path*, *prefix=''*, *suffix=''*, *padding=0*)

Extract the file name.

hysteresis_threshold

omnipose.utils.**hysteresis_threshold**(*image*, *low*, *high*)

Pytorch implementation of skimage.filters.apply_hysteresis_threshold(). Discprepencies occur for very high thresholds/thin objects.

is_integer

omnipose.utils.**is_integer**(*var*)

kernel_setup

omnipose.utils.**kernel_setup**(*dim*)

Get relevant kernel information for the hypercube of interest. Calls get_steps(), steps_to_indices().

Parameters

dim (*int*) -- dimension (usually 2 or 3, but can be any positive integer)

Returns

- **steps** (*ndarray, int*) -- list of steps to each kernal point see get_steps()
- **idx** (*int*) -- index of the central point within the step list this is always (3**dim)//2
- **inds** (*ndarray, int*) -- list of kernel points sorted by type see steps_to_indices()
- **fact** (*float*) -- list of face/edge/vertex/... distances see steps_to_indices()
- **sign** (*1D array, int*) -- signature distinguishing each kind of m-face via the number of steps see steps_to_indices()

load_nested_list

omnipose.utils.**load_nested_list**(*file_path*)

Helper function to load affinity graphs.

localnormalize

omnipose.utils.**localnormalize**(*im*, *sigma1=2*, *sigma2=20*)

localnormalize_GPU

omnipose.utils.**localnormalize_GPU**(*im*, *sigma1=2*, *sigma2=20*)

make_tiles_ND

omnipose.utils.**make_tiles_ND**(*imgi*, *bsize=224*, *augment=False*, *tile_overlap=0.1*, *normalize=True*, *return_tiles=True*)

make tiles of image to run at test-time

if augmented, tiles are flipped and tile_overlap=2.

- original
- flipped vertically
- flipped horizontally
- flipped vertically and horizontally

Parameters

- **imgi** (*float32*) -- array that's nchan x Ly x Lx
- **bsize** (*float (optional, default 224)*) -- size of tiles
- **augment** (*bool (optional, default False)*) -- flip tiles and set tile_overlap=2.
- **tile_overlap** (*float (optional, default 0.1)*) -- fraction of overlap of tiles

Returns

- **IMG** (*float32*) -- tensor of shape ntiles,nchan,bsize,bsize
- **subs** (*list*) -- list of slices for each subtile
- **shape** (*tuple*) -- shape of original image

make_unique

omnipose.utils.**make_unique**(*masks*)

Relabel stack of label matrices such that there is no repeated label across slices.

mask_outline_overlay

omnipose.utils.**mask_outline_overlay**(*img*, *masks*, *outlines*, *mono=None*)

Apply a color overlay to a grayscale image based on a label matrix. mono is a single color to use. Otherwise, N sinebow colors are used.

mono_mask_bd

omnipose.utils.**mono_mask_bd**(*masks*, *outlines*, *color=[1, 0, 0]*, *a=0.25*)

moving_average

omnipose.utils.**moving_average**(*x*, *w*)

normalize99

omnipose.utils.**normalize99**(*Y*, *lower=0.01*, *upper=99.99*, *contrast_limits=None*, *dim=None*)

normalize array/tensor so 0.0 is 0.01st percentile and 1.0 is 99.99th percentile Upper and lower percentile ranges configurable.

Parameters

- **Y** (*ndarray/tensor, float*) -- Input array/tensor.
- **upper** (*float*) -- upper percentile above which pixels are sent to 1.0
- **lower** (*float*) -- lower percentile below which pixels are sent to 0.0
- **contrast_limits** (*list, float (optional, override computation)*) -- list of two floats, lower and upper contrast limits

Return type

normalized array/tensor with a minimum of 0 and maximum of 1

normalize_field

omnipose.utils.**normalize_field**(*mu*, *use_torch=False*, *cutoff=0*)

normalize all nonzero field vectors to magnitude 1

Parameters

mu (*ndarray, float*) -- Component array of lenth N by L1 by L2 by ... by LN.

Return type

normalized component array of identical size.

normalize_image

omnipose.utils.**normalize_image**(*im*, *mask*, *target=0.5*, *foreground=False*, *iterations=1*, *scale=1*, *channel_axis=0*, *per_channel=True*)

Normalize image by rescaling from 0 to 1 and then adjusting gamma to bring average background to specified value (0.5 by default).

Parameters

- **im** (*ndarray, float*) -- input image or volume
- **mask** (*ndarray, int or bool*) -- input labels or foreground mask
- **target** (*float*) -- target background/foreground value in the range 0-1
- **channel_axis** (*int*) -- the axis that contains the channels

Return type
gamma-normalized array with a minimum of 0 and maximum of 1

normalize_stack

omnipose.utils.**normalize_stack**(*vol*, *mask*, *bg=0.5*, *bright_foreground=None*, *subtractive=False*, *iterations=1*, *equalize_foreground=1*, *quantiles=[0.01, 0.99]*)

Adjust image stacks so that background is (1) consistent in brightness and (2) brought to an even average via semantic gamma normalization.

phase_cross_correlation_GPU

omnipose.utils.**phase_cross_correlation_GPU**(*image_stack*, *upsample_factor=10*, *normalization=None*)

phase_cross_correlation_GPU_old

omnipose.utils.**phase_cross_correlation_GPU_old**(*image_stack*, *target_index=None*, *upsample_factor=10*, *reverse=False*, *normalize=False*)

ravel_index

omnipose.utils.**ravel_index**(*b*, *shp*)

remap_pairs

omnipose.utils.**remap_pairs**(*pairs*, *replacements*)

rescale

omnipose.utils.**rescale**(*T*, *floor=None*, *ceiling=None*, *exclude_dims=None*)

Rescale data between 0 and 1. exclude_dims is the axis or axes that will remain.

rotate

omnipose.utils.**rotate**(*V*, *theta*, *order=1*, *output_shape=None*, *center=None*)

safe_divide

omnipose.utils.**safe_divide**(*num*, *den*, *cutoff=0*)

Division ignoring zeros and NaNs in the denominator.

save_nested_list

omnipose.utils.**save_nested_list**(*file_path*, *nested_list*)

Helper function to save affinity graphs.

shift_stack

omnipose.utils.**shift_stack**(*imstack*, *shift_vectors*, *order=1*, *cval=None*, *prefilter=True*)

Shift each time slice of imstack according to list of nD shifts.

shifts_to_slice

omnipose.utils.**shifts_to_slice**(*shifts*, *shape*)

Find the minimal crop box from time lapse registration shifts.

steps_to_indices

omnipose.utils.**steps_to_indices**(*steps*)

Get indices of the hupercubes sharing m-faces on the central n-cube. These are sorted by the connectivity (by center, face, edge, vertex, ...). I.e., the central point index is first, followed by cardinal directions, ordinals, and so on.

subsample_affinity

omnipose.utils.**subsample_affinity**(*augmented_affinity*, *slc*, *mask*)

Helper function to subsample an affinity graph according to an image crop slice and a foreground selection mask.

Parameters

- **augmented_affinity** (*NDarray, int64*) -- Stacked neighbor coordinate array and affinity graph. For dimension d, augmented_affinity[:d] are the neighbor coordinates of shape (d,3**d,npix) and augmented_affinity[d] is the affinity graph of shape (3**d,npix).
- **slc** (*tuple, slice*) -- tuple of slices along each dimension defining the crop window
- **mask** (*NDarray, bool*) -- foreground selection mask, in the image space of the original graph (i.e., not already sliced)

Return type

Augmented affinity graph corresponding to the cropped/masked region.

thin_skeleton

omnipose.utils.**thin_skeleton**(*image*)

to_16_bit

omnipose.utils.**to_16_bit**(*im*)

Rescale image [0,2^16-1] and then cast to uint16.

to_8_bit

omnipose.utils.**to_8_bit**(*im*)

Rescale image [0,2^8-1] and then cast to uint8.

torch_norm

omnipose.utils.**torch_norm**(*a*, *dim=0*, *keepdim=False*)

Wrapper for torch.linalg.norm to handle ARM architecture.

unaugment_tiles_ND

omnipose.utils.**unaugment_tiles_ND**(*y*, *inds*, *unet=False*)

reverse test-time augmentations for averaging

Parameters

- **y** (*float32*) -- array of shape (ntiles, nchan, *DIMS) where nchan = (*DP,distance) (and boundary if nlasses=3)
- **unet** (*bool (optional, False)*) -- whether or not unet output or cellpose output

Returns

y

Return type

float32

unravel_index

omnipose.utils.**unravel_index**(*index*, *shape*)

14.2.3 omnipose.plot

apply_ncolor(masks[, offset, cmap, ...])	
color_from_RGB(im, rgb, m[, bd, mode, ...])	
colored_line(x, y, ax[, z, line_width, MAP])	
colored_line_segments(xs, ys[, zs, color, ...])	
colorize(im[, colors, color_weights, ...])	
colorize_GPU(im[, colors, color_weights, ...])	
create_colormap(image, labels)	Create a colormap based on the average color of each label in the image.
custom_new_gc(self)	
faded_segment_resample(xs, ys[, zs, color, ...])	
image_grid(images[, column_titles, ...])	Display a grid of images with uniform spacing.
imshow(imgs[, figsize, ax, hold, titles, ...])	
plot_color_swatches(colors[, figsize, dpi])	
plot_edges(shape, affinity_graph, neighbors, ...)	
rgb_flow(dP[, transparency, mask, norm, device])	Meant for stacks of dP, unsqueeze if using on a single plane.
segmented_resample(xs, ys[, zs, color, ...])	
sinebow(N[, bg_color, offset])	Generate a color dictionary for use in visualizing N-colored labels.
truncate_colormap(cmap[, minval, maxval, n])	

apply_ncolor

omnipose.plot.**apply_ncolor**(*masks*, *offset=0*, *cmap=None*, *max_depth=20*, *expand=True*)

color_from_RGB

omnipose.plot.**color_from_RGB**(*im*, *rgb*, *m*, *bd=None*, *mode='inner'*, *connectivity=2*)

colored_line

omnipose.plot.**colored_line**(*x*, *y*, *ax*, *z=None*, *line_width=1*, *MAP='jet'*)

colored_line_segments

omnipose.plot.**colored_line_segments**(*xs*, *ys*, *zs=None*, *color='k'*, *mid_colors=False*)

colorize

omnipose.plot.**colorize**(*im*, *colors=None*, *color_weights=None*, *offset=0*, *channel_axis=- 1*)

colorize_GPU

omnipose.plot.**colorize_GPU**(*im*, *colors=None*, *color_weights=None*, *offset=0*, *channel_axis=- 1*)

create_colormap

omnipose.plot.**create_colormap**(*image*, *labels*)

Create a colormap based on the average color of each label in the image.

Parameters

- **image** (*ndarray*) -- An RGB image.
- **labels** (*ndarray*) -- A 2D array of labels corresponding to the image.

Returns

colormap -- A colormap where each row is the RGB color for the corresponding label.

Return type

ndarray

custom_new_gc

omnipose.plot.**custom_new_gc**(*self*)

faded_segment_resample

omnipose.plot.**faded_segment_resample**(*xs*, *ys*, *zs=None*, *color='k'*, *fade_len=20*, *n_resample=100*, *direction='Head'*)

image_grid

omnipose.plot.**image_grid**(*images*, *column_titles=None*, *row_titles=None*, *plot_labels=None*, *xticks=[]*, *yticks=[]*, *outline=False*, *outline_color=[0.5, 0.5, 0.5]*, *padding=0.05*, *fontsize=10*, *fontcolor=[0.5, 0.5, 0.5]*, *fig_scale=6*, *dpi=300*, *order='ij'*, *lpad=0.05*, *lpos='top_middle'*, ***kwargs*)

Display a grid of images with uniform spacing. Accepts a neested list of images, with each sublist having cosnsitent YXC diemnsions.

imshow

omnipose.plot.**imshow**(*imgs*, *figsize=2*, *ax=None*, *hold=False*, *titles=None*, *title_size=None*, *spacing=0.05*, *textcolor=[0.5, 0.5, 0.5]*, *dpi=300*, ***kwargs*)

plot_color_swatches

omnipose.plot.**plot_color_swatches**(*colors*, *figsize=0.5*, *dpi=100*)

plot_edges

omnipose.plot.**plot_edges**(*shape*, *affinity_graph*, *neighbors*, *coords*, *figsize=1*, *fig=None*, *ax=None*, *extent=None*, *slc=None*, *pic=None*, *edgecol=[0.75, 0.75, 0.75, 0.5]*, *linewidth=0.15*, *step_inds=None*, *cmap='inferno'*, *origin='lower'*, *bounds=None*)

rgb_flow

omnipose.plot.**rgb_flow**(*dP*, *transparency=True*, *mask=None*, *norm=True*, *device=device(type='cpu')*)

Meant for stacks of dP, unsqueeze if using on a single plane.

segmented_resample

omnipose.plot.**segmented_resample**(*xs*, *ys*, *zs=None*, *color='k'*, *n_resample=100*, *mid_colors=False*)

sinebow

omnipose.plot.**sinebow**(*N*, *bg_color=[0, 0, 0, 0]*, *offset=0*)

Generate a color dictionary for use in visualizing N-colored labels. Background color defaults to transparent black.

Parameters

- **N** (*int*) -- number of distinct colors to generate (excluding background)
- **bg_color** (*ndarray, list, or tuple of length 4*) -- RGBA values specifying the background color at the front of the dictionary.

Returns

Dictionary with entries {int

Return type

RGBA array} to map integer labels to RGBA colors.

truncate_colormap

omnipose.plot.**truncate_colormap**(*cmap*, *minval=0.0*, *maxval=1.0*, *n=100*)

14.2.4 cellpose_omni.models

ARM	bool(x) -> bool
BD_MODEL_NAMES	Built-in mutable sequence.
C1_BD_MODELS	Built-in mutable sequence.
C1_MODELS	Built-in mutable sequence.
C2_BD_MODELS	Built-in mutable sequence.
C2_MODELS	Built-in mutable sequence.
C2_MODEL_NAMES	Built-in mutable sequence.
CP_MODELS	Built-in mutable sequence.
Cellpose([gpu, model_type, net_avg, device, ...])	main model which combines SizeModel and CellposeModel
CellposeModel([gpu, pretrained_model, ...])	**param gpu** whether or not to save model to GPU, will check if GPU available
MODEL_DIR	Path subclass for non-Windows systems.
MODEL_NAMES	Built-in mutable sequence.
MXNET_ENABLED	bool(x) -> bool
OMNI_INSTALLED	bool(x) -> bool
SizeModel(cp_model[, device, pretrained_size])	linear regression model for determining the size of objects in image used to rescale before input to cp_model uses styles from cp_model
cache_model_path(basename)	
deprecation_warning_cellprob_dist_threshold(...)	
model_path(model_type, model_index, use_torch)	
models_logger	Instances of the Logger class represent a single logging channel.
size_model_path(model_type, use_torch)	

ARM

cellpose_omni.models.**ARM = False**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

BD_MODEL_NAMES

cellpose_omni.models.**BD_MODEL_NAMES = ['bact_phase_omni', 'bact_fluor_omni', 'worm_omni', 'worm_bact_omni', 'worm_high_res_omni', 'cyto2_omni', 'plant_omni']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

C1_BD_MODELS

cellpose_omni.models.**C1_BD_MODELS = ['plant_omni']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

C1_MODELS

cellpose_omni.models.**C1_MODELS = []**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

C2_BD_MODELS

cellpose_omni.models.**C2_BD_MODELS = ['bact_phase_omni', 'bact_fluor_omni', 'worm_omni', 'worm_bact_omni', 'worm_high_res_omni', 'cyto2_omni']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

C2_MODELS

cellpose_omni.models.**C2_MODELS = ['bact_phase_cp', 'bact_fluor_cp', 'plant_cp', 'worm_cp']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

C2_MODEL_NAMES

cellpose_omni.models.**C2_MODEL_NAMES = ['bact_phase_omni', 'bact_fluor_omni', 'worm_omni', 'worm_bact_omni', 'worm_high_res_omni', 'cyto2_omni', 'bact_phase_cp', 'bact_fluor_cp', 'plant_cp', 'worm_cp', 'cyto', 'nuclei', 'cyto2']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

CP_MODELS

cellpose_omni.models.**CP_MODELS = ['cyto', 'nuclei', 'cyto2']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

Cellpose

class cellpose_omni.models.**Cellpose**(*gpu=False*, *model_type='cyto'*, *net_avg=True*, *device=None*, *use_torch=True*, *model_dir=None*, *dim=2*, *omni=None*)

Bases: `object`

main model which combines SizeModel and CellposeModel

Parameters

- **gpu** (*bool (optional, default False)*) -- whether or not to use GPU, will check if GPU available
- **model_type** (*str (optional, default 'cyto')*) -- 'cyto'=cytoplasm model; 'nuclei'=nucleus model
- **net_avg** (*bool (optional, default True)*) -- loads the 4 built-in networks and averages them if True, loads one network if False
- **device** (*gpu device (optional, default None)*) -- where model is saved (e.g. mx.gpu() or mx.cpu()), overrides gpu input, recommended if you want to use a specific GPU (e.g. mx.gpu(4) or torch.cuda.device(4))
- **torch** (*bool (optional, default True)*) -- run model using torch if available

Methods Summary

eval(x[, batch_size, channels, ...])	run cellpose and get masks

Methods Documentation

eval(*x*, *batch_size=8*, *channels=None*, *channel_axis=None*, *z_axis=None*, *invert=False*, *normalize=True*, *diameter=30.0*, *do_3D=False*, *anisotropy=None*, *net_avg=True*, *augment=False*, *tile=True*, *tile_overlap=0.1*, *resample=True*, *interp=True*, *cluster=False*, *boundary_seg=False*, *affinity_seg=False*, *despur=True*, *flow_threshold=0.4*, *mask_threshold=0.0*, *cellprob_threshold=None*, *dist_threshold=None*, *diam_threshold=12.0*, *min_size=15*, *max_size=None*, *stitch_threshold=0.0*, *rescale=None*, *progress=None*, *omni=False*, *verbose=False*, *transparency=False*, *model_loaded=False*)

run cellpose and get masks

Parameters

- **x** (*list or array of images*) -- can be list of 2D/3D images, or array of 2D/3D images, or 4D image array
- **batch_size** (*int (optional, default 8)*) -- number of 224x224 patches to run simultaneously on the GPU (can make smaller or bigger depending on GPU memory usage)

- **channels** (*list (optional, default None)*) -- list of channels, either of length 2 or of length number of images by 2. First element of list is the channel to segment (0=grayscale, 1=red, 2=green, 3=blue). Second element of list is the optional nuclear channel (0=none, 1=red, 2=green, 3=blue). For instance, to segment grayscale images, input [0,0]. To segment images with cells in green and nuclei in blue, input [2,3]. To segment one grayscale image and one image with cells in green and nuclei in blue, input [[0,0], [2,3]].
- **channel_axis** (*int (optional, default None)*) -- if None, channels dimension is attempted to be automatically determined
- **z_axis** (*int (optional, default None)*) -- if None, z dimension is attempted to be automatically determined
- **invert** (*bool (optional, default False)*) -- invert image pixel intensity before running network (if True, image is also normalized)
- **normalize** (*bool (optional, default True)*) -- normalize data so 0.0=1st percentile and 1.0=99th percentile of image intensities in each channel
- **diameter** (*float (optional, default 30.)*) -- if set to None, then diameter is automatically estimated if size model is loaded
- **do_3D** (*bool (optional, default False)*) -- set to True to run 3D segmentation on 4D image input
- **anisotropy** (*float (optional, default None)*) -- for 3D segmentation, optional rescaling factor (e.g. set to 2.0 if Z is sampled half as dense as X or Y)
- **net_avg** (*bool (optional, default True)*) -- runs the 4 built-in networks and averages them if True, runs one network if False
- **augment** (*bool (optional, default False)*) -- tiles image with overlapping tiles and flips overlapped regions to augment
- **tile** (*bool (optional, default True)*) -- tiles image to ensure GPU/CPU memory usage limited (recommended)
- **tile_overlap** (*float (optional, default 0.1)*) -- fraction of overlap of tiles when computing flows
- **resample** (*bool (optional, default True)*) -- run dynamics at original image size (will be slower but create more accurate boundaries)
- **interp** (*bool (optional, default True)*) -- interpolate during 2D dynamics (not available in 3D) (in previous versions it was False)
- **flow_threshold** (*float (optional, default 0.4)*) -- flow error threshold (all cells with errors below threshold are kept) (not used for 3D)
- **mask_threshold** (*float (optional, default 0.0)*) -- all pixels with value above threshold kept for masks, decrease to find more and larger masks
- **dist_threshold** (*float (optional, default None) DEPRECATED*) -- use mask_threshold instead
- **cellprob_threshold** (*float (optional, default None) DEPRECATED*) -- use mask_threshold instead
- **min_size** (*int (optional, default 15)*) -- minimum number of pixels per mask, can turn off with -1

- **stitch_threshold** (*float (optional, default 0.0)*) -- if stitch_threshold>0.0 and not do_3D and equal image sizes, masks are stitched in 3D to return volume segmentation
- **rescale** (*float (optional, default None)*) -- if diameter is set to None, and rescale is not None, then rescale is used instead of diameter for resizing image
- **progress** (*pyqt progress bar (optional, default None)*) -- to return progress bar status to GUI
- **omni** (*bool (optional, default False)*) -- use omnipose mask recontruction features
- **calc_trace** (*bool (optional, default False)*) -- calculate pixel traces and return as part of the flow
- **verbose** (*bool (optional, default False)*) -- turn on additional output to logs for debugging
- **verbose** -- turn on additional output to logs for debugging
- **transparency** (*bool (optional, default False)*) -- modulate flow opacity by magnitude instead of brightness (can use flows on any color background)
- **model_loaded** (*bool (optional, default False)*) -- internal variable for determining if model has been loaded, used in __main__.py

Returns

- **masks** (*list of 2D arrays, or single 3D array (if do_3D=True)*) -- labelled image, where 0=no masks; 1,2,...=mask labels
- **flows** (*list of lists 2D arrays, or list of 3D arrays (if do_3D=True)*) -- flows[k][0] = XY flow in HSV 0-255 flows[k][1] = flows at each pixel flows[k][2] = scalar cell probability (Cellpose) or distance transform (Omnipose) flows[k][3] = final pixel locations after Euler integration flows[k][4] = boundary output (nonempty for Omnipose) flows[k][5] = pixel traces (nonempty for calc_trace=True)
- **styles** (*list of 1D arrays of length 256, or single 1D array (if do_3D=True)*) -- style vector summarizing each image, also used to estimate size of objects in image
- **diams** (*list of diameters, or float (if do_3D=True)*)

CellposeModel

class cellpose_omni.models.**CellposeModel**(*gpu=False*, *pretrained_model=False*, *model_type=None*, *net_avg=True*, *use_torch=True*, *diam_mean=30.0*, *device=None*, *residual_on=True*, *style_on=True*, *concatenation=False*, *nchan=1*, *nclasses=2*, *dim=2*, *omni=True*, *checkpoint=False*, *dropout=False*, *kernel_size=2*)

Bases: UnetModel

Parameters

- **gpu** (*bool (optional, default False)*) -- whether or not to save model to GPU, will check if GPU available
- **pretrained_model** (*str or list of strings (optional, default False)*) -- path to pretrained cellpose model(s), if None or False, no model loaded

- **model_type** (*str (optional, default None)*) -- 'cyto'=cytoplasm model; 'nuclei'=nucleus model; if None, pretrained_model used
- **net_avg** (*bool (optional, default True)*) -- loads the 4 built-in networks and averages them if True, loads one network if False
- **torch** (*bool (optional, default True)*) -- use torch nn rather than mxnet
- **diam_mean** (*float (optional, default 27.)*) -- mean 'diameter', 27. is built in value for 'cyto' model
- **device** (*mxnet device (optional, default None)*) -- where model is saved (mx.gpu() or mx.cpu()), overrides gpu input, recommended if you want to use a specific GPU (e.g. mx.gpu(4))
- **model_dir** (*str (optional, default None)*) -- overwrite the built in model directory where cellpose looks for models
- **omni** (*use omnipose model (optional, default False)*) --

Methods Summary

eval(x[, batch_size, indices, channels, ...])	Evaluation for CellposeModel.
loss_fn(lbl, y)	loss function between true labels lbl and prediction y This is the one used to train the instance segmentation network.
train(train_data, train_labels[, ...])	train network with images train_data

Methods Documentation

eval(*x*, *batch_size=8*, *indices=None*, *channels=None*, *channel_axis=None*, *z_axis=None*, *normalize=True*, *invert=False*, *rescale=None*, *diameter=None*, *do_3D=False*, *anisotropy=None*, *net_avg=True*, *augment=False*, *tile=True*, *tile_overlap=0.1*, *bsize=224*, *num_workers=8*, *resample=True*, *interp=True*, *cluster=False*, *suppress=None*, *boundary_seg=False*, *affinity_seg=False*, *despur=True*, *flow_threshold=0.4*, *mask_threshold=0.0*, *diam_threshold=12.0*, *niter=None*, *cellprob_threshold=None*, *dist_threshold=None*, *flow_factor=5.0*, *compute_masks=True*, *min_size=15*, *max_size=None*, *stitch_threshold=0.0*, *progress=None*, *show_progress=True*, *omni=False*, *calc_trace=False*, *verbose=False*, *transparency=False*, *loop_run=False*, *model_loaded=False*, *hysteresis=True*)

Evaluation for CellposeModel. Segment list of images x, or 4D array - Z x nchan x Y x X

Parameters

- **x** (*list or array of images*) -- can be list of 2D/3D/4D images, or array of 2D/3D/4D images
- **batch_size** (*int (optional, default 8)*) -- number of 224x224 patches to run simultaneously on the GPU (can make smaller or bigger depending on GPU memory usage)
- **channels** (*list (optional, default None)*) -- list of channels, either of length 2 or of length number of images by 2. First element of list is the channel to segment (0=grayscale, 1=red, 2=green, 3=blue). Second element of list is the optional nuclear channel (0=none, 1=red, 2=green, 3=blue). For instance, to segment grayscale images, input [0,0]. To segment images with cells in green and nuclei in blue, input [2,3]. To segment one grayscale image and one image with cells in green and nuclei in blue, input [[0,0], [2,3]].

- **channel_axis** (*int (optional, default None)*) -- if None, channels dimension is attempted to be automatically determined
- **z_axis** (*int (optional, default None)*) -- if None, z dimension is attempted to be automatically determined
- **normalize** (*bool (default, True)*) -- normalize data so 0.0=1st percentile and 1.0=99th percentile of image intensities in each channel
- **invert** (*bool (optional, default False)*) -- invert image pixel intensity before running network
- **rescale** (*float (optional, default None)*) -- resize factor for each image, if None, set to 1.0
- **diameter** (*float (optional, default None)*) -- diameter for each image (only used if rescale is None), if diameter is None, set to diam_mean
- **do_3D** (*bool (optional, default False)*) -- set to True to run 3D segmentation on 4D image input
- **anisotropy** (*float (optional, default None)*) -- for 3D segmentation, optional rescaling factor (e.g. set to 2.0 if Z is sampled half as dense as X or Y)
- **net_avg** (*bool (optional, default True)*) -- runs the 4 built-in networks and averages them if True, runs one network if False
- **augment** (*bool (optional, default False)*) -- tiles image with overlapping tiles and flips overlapped regions to augment
- **tile** (*bool (optional, default True)*) -- tiles image to ensure GPU/CPU memory usage limited (recommended)
- **tile_overlap** (*float (optional, default 0.1)*) -- fraction of overlap of tiles when computing flows
- **resample** (*bool (optional, default True)*) -- run dynamics at original image size (will be slower but create more accurate boundaries)
- **interp** (*bool (optional, default True)*) -- interpolate during 2D dynamics (not available in 3D) (in previous versions it was False)
- **flow_threshold** (*float (optional, default 0.4)*) -- flow error threshold (all cells with errors below threshold are kept) (not used for 3D)
- **mask_threshold** (*float (optional, default 0.0)*) -- all pixels with value above threshold kept for masks, decrease to find more and larger masks
- **dist_threshold** (*float (optional, default None) DEPRECATED*) -- use mask_threshold instead
- **cellprob_threshold** (*float (optional, default None) DEPRECATED*) -- use mask_threshold instead
- **compute_masks** (*bool (optional, default True)*) -- Whether or not to compute dynamics and return masks. This is set to False when retrieving the styles for the size model.
- **min_size** (*int (optional, default 15)*) -- minimum number of pixels per mask, can turn off with -1
- **stitch_threshold** (*float (optional, default 0.0)*) -- if stitch_threshold>0.0 and not do_3D, masks are stitched in 3D to return volume segmentation

- **progress** (*pyqt progress bar (optional, default None)*) -- to return progress bar status to GUI
- **omni** (*bool (optional, default False)*) -- use omnipose mask reconstruction features
- **calc_trace** (*bool (optional, default False)*) -- calculate pixel traces and return as part of the flow
- **verbose** (*bool (optional, default False)*) -- turn on additional output to logs for debugging
- **transparency** (*bool (optional, default False)*) -- modulate flow opacity by magnitude instead of brightness (can use flows on any color background)
- **loop_run** (*bool (optional, default False)*) -- internal variable for determining if model has been loaded, stops model loading in loop over images
- **model_loaded** (*bool (optional, default False)*) -- internal variable for determining if model has been loaded, used in __main__.py

Returns

- **masks** (*list of 2D arrays, or single 3D array (if do_3D=True)*) -- labelled image, where 0=no masks; 1,2,...=mask labels
- **flows** (*list of lists 2D arrays, or list of 3D arrays (if do_3D=True)*) -- flows[k][0] = 8-bit RGb phase plot of flow field flows[k][1] = flows at each pixel flows[k][2] = scalar cell probability (Cellpose) or distance transform (Omnipose) flows[k][3] = boundary output (nonempty for Omnipose) flows[k][4] = final pixel locations after Euler integration flows[k][5] = pixel traces (nonempty for calc_trace=True)
- **styles** (*list of 1D arrays of length 64, or single 1D array (if do_3D=True)*) -- style vector summarizing each image, also used to estimate size of objects in image

loss_fn(*lbl*, *y*)

loss function between true labels lbl and prediction y This is the one used to train the instance segmentation network.

train(*train_data*, *train_labels*, *train_links=None*, *train_files=None*, *test_data=None*, *test_labels=None*, *test_links=None*, *test_files=None*, *channels=None*, *channel_axis=0*, *normalize=True*, *save_path=None*, *save_every=100*, *save_each=False*, *learning_rate=0.2*, *n_epochs=500*, *momentum=0.9*, *SGD=True*, *weight_decay=1e-05*, *batch_size=8*, *dataloader=False*, *num_workers=0*, *nimg_per_epoch=None*, *rescale=True*, *min_train_masks=5*, *netstr=None*, *tyx=None*, *timing=False*, *do_autocast=False*, *affinity_field=False*)

train network with images train_data

Parameters

- **train_data** (*list of arrays (2D or 3D)*) -- images for training
- **train_labels** (*list of arrays (2D or 3D)*) -- labels for train_data, where 0=no masks; 1,2,...=mask labels can include flows as additional images
- **train_links** (*list of label links*) -- These lists of label pairs define which labels are "linked", i.e. should be treated as part of the same object. This is how Omnipose handles internal/self-contact boundaries during training.
- **train_files** (*list of strings*) -- file names for images in train_data (to save flows for future runs)
- **test_data** (*list of arrays (2D or 3D)*) -- images for testing

- **test_labels** (*list of arrays (2D or 3D)*) -- See train_labels.
- **test_links** (*list of label links*) -- See train_links.
- **test_files** (*list of strings*) -- file names for images in test_data (to save flows for future runs)
- **channels** (*list of ints (default, None)*) -- channels to use for training
- **normalize** (*bool (default, True)*) -- normalize data so 0.0=1st percentile and 1.0=99th percentile of image intensities in each channel
- **save_path** (*string (default, None)*) -- where to save trained model, if None it is not saved
- **save_every** (*int (default, 100)*) -- save network every [save_every] epochs
- **learning_rate** (*float or list/np.ndarray (default, 0.2)*) -- learning rate for training, if list, must be same length as n_epochs
- **n_epochs** (*int (default, 500)*) -- how many times to go through whole training set during training
- **weight_decay** (*float (default, 0.00001)*) --
- **SGD** (*bool (default, True)*) -- use SGD as optimization instead of RAdam
- **batch_size** (*int (optional, default 8)*) -- number of tyx-sized patches to run simultaneously on the GPU (can make smaller or bigger depending on GPU memory usage)
- **nimg_per_epoch** (*int (optional, default None)*) -- minimum number of images to train on per epoch, with a small training set (< 8 images) it may help to set to 8
- **rescale** (*bool (default, True)*) -- whether or not to rescale images to diam_mean during training, if True it assumes you will fit a size model after training or resize your images accordingly, if False it will try to train the model to be scale-invariant (works worse)
- **min_train_masks** (*int (default, 5)*) -- minimum number of masks an image must have to use in training set
- **netstr** (*str (default, None)*) -- name of network, otherwise saved with name as params + training start time
- **tyx** (*int, tuple (default, 224x224 in 2D)*) -- size of image patches used for training

MODEL_DIR

cellpose_omni.models.**MODEL_DIR = PosixPath('/home/docs/.cellpose/models')**

Path subclass for non-Windows systems.

On a POSIX system, instantiating a Path should return this object.

MODEL_NAMES

cellpose_omni.models.**MODEL_NAMES = ['bact_phase_omni', 'bact_fluor_omni', 'worm_omni', 'worm_bact_omni', 'worm_high_res_omni', 'cyto2_omni', 'plant_omni', 'bact_phase_cp', 'bact_fluor_cp', 'plant_cp', 'worm_cp', 'cyto', 'nuclei', 'cyto2']**

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

MXNET_ENABLED

cellpose_omni.models.**MXNET_ENABLED = False**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

OMNI_INSTALLED

cellpose_omni.models.**OMNI_INSTALLED = True**

bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

SizeModel

class cellpose_omni.models.**SizeModel**(*cp_model*, *device=None*, *pretrained_size=None*, ***kwargs*)

Bases: object

linear regression model for determining the size of objects in image used to rescale before input to cp_model uses styles from cp_model

Parameters

- **cp_model** (*UnetModel or CellposeModel*) -- model from which to get styles
- **device** (*mxnet device (optional, default mx.cpu())*) -- where cellpose model is saved (mx.gpu() or mx.cpu())
- **pretrained_size** (*str*) -- path to pretrained size model
- **omni** (*bool*) -- whether or not to use distance-based size metrics corresponding to 'omni' model

Methods Summary

eval(x[, channels, channel_axis, normalize, ...])	Evaluation for SizeModel.
train(train_data, train_labels[, test_data, ...])	train size model with images train_data to estimate linear model from styles to diameters

Methods Documentation

eval(*x*, *channels=None*, *channel_axis=None*, *normalize=True*, *invert=False*, *augment=False*, *tile=True*, *batch_size=8*, *progress=None*, *interp=True*, *omni=False*)

Evaluation for SizeModel. Use images x to produce style or use style input to predict size of objects in image.

Object size estimation is done in two steps: 1. use a linear regression model to predict size from style in image 2. resize image to predicted size and run CellposeModel to get output masks.

Take the median object size of the predicted masks as the final predicted size.

Parameters

- **x** (*list or array of images*) -- can be list of 2D/3D images, or array of 2D/3D images
- **channels** (*list (optional, default None)*) -- list of channels, either of length 2 or of length number of images by 2. First element of list is the channel to segment (0=grayscale, 1=red, 2=green, 3=blue). Second element of list is the optional nuclear channel (0=none, 1=red, 2=green, 3=blue). For instance, to segment grayscale images, input [0,0]. To segment images with cells in green and nuclei in blue, input [2,3]. To segment one grayscale image and one image with cells in green and nuclei in blue, input [[0,0], [2,3]].
- **channel_axis** (*int (optional, default None)*) -- if None, channels dimension is attempted to be automatically determined
- **normalize** (*bool (default, True)*) -- normalize data so 0.0=1st percentile and 1.0=99th percentile of image intensities in each channel
- **invert** (*bool (optional, default False)*) -- invert image pixel intensity before running network
- **augment** (*bool (optional, default False)*) -- tiles image with overlapping tiles and flips overlapped regions to augment
- **tile** (*bool (optional, default True)*) -- tiles image to ensure GPU/CPU memory usage limited (recommended)
- **progress** (*pyqt progress bar (optional, default None)*) -- to return progress bar status to GUI

Returns

- **diam** (*array, float*) -- final estimated diameters from images x or styles style after running both steps
- **diam_style** (*array, float*) -- estimated diameters from style alone

train(*train_data*, *train_labels*, *test_data=None*, *test_labels=None*, *channels=None*, *normalize=True*, *learning_rate=0.2*, *n_epochs=10*, *l2_regularization=1.0*, *batch_size=8*)

train size model with images train_data to estimate linear model from styles to diameters

Parameters

- **train_data** (*list of arrays (2D or 3D)*) -- images for training
- **train_labels** (*list of arrays (2D or 3D)*) -- labels for train_data, where 0=no masks; 1,2,...=mask labels can include flows as additional images
- **channels** (*list of ints (default, None)*) -- channels to use for training

- **normalize** (*bool (default, True)*) -- normalize data so 0.0=1st percentile and 1.0=99th percentile of image intensities in each channel
- **n_epochs** (*int (default, 10)*) -- how many times to go through whole training set (taking random patches) for styles for diameter estimation
- **l2_regularization** (*float (default, 1.0)*) -- regularize linear model from styles to diameters
- **batch_size** (*int (optional, default 8)*) -- number of 224x224 patches to run simultaneously on the GPU (can make smaller or bigger depending on GPU memory usage)

cache_model_path

cellpose_omni.models.**cache_model_path**(*basename*)

deprecation_warning_cellprob_dist_threshold

cellpose_omni.models.**deprecation_warning_cellprob_dist_threshold**(*cellprob_threshold*, *dist_threshold*)

model_path

cellpose_omni.models.**model_path**(*model_type*, *model_index*, *use_torch*)

models_logger

cellpose_omni.models.**models_logger = <Logger cellpose_omni.models (INFO)>**

Instances of the Logger class represent a single logging channel. A "logging channel" indicates an area of an application. Exactly how an "area" is defined is up to the application developer. Since an application can have any number of areas, logging channels are identified by a unique string. Application areas can be nested (e.g. an area of "input processing" might include sub-areas "read CSV files", "read XLS files" and "read Gnumeric files"). To cater for this natural nesting, channel names are organized into a namespace hierarchy where levels are separated by periods, much like the Java or Python package namespace. So in the instance given above, channel names might be "input" for the upper level, and "input.csv", "input.xls" and "input.gnu" for the sub-levels. There is no arbitrary limit to the depth of nesting.

size_model_path

cellpose_omni.models.**size_model_path**(*model_type*, *use_torch*)

14.2.5 cellpose_omni.io

check_dir(path)	
get_image_files(folder[, mask_filter, ...])	find all images in a folder and if look_one_level_down all subfolders
get_label_files(img_names[, label_filter, ...])	Get the corresponding labels and flows for the given file images.
getname(path[, suffix])	
imread(filename)	
imsave(filename, arr)	
imwrite(filename, arr, **kwargs)	
load_links(filename)	Read a txt or csv file with label links. These should look like: 1,2 1,3 4,7 6,19 . . . Returns links as a set of tuples.
load_train_test_data(train_dir[, test_dir, ...])	Loads the training and optional test data for training runs.
logger_setup([verbose])	
masks_flows_to_seg(images, masks, flows, ...)	save output of model eval to be loaded in GUI
outlines_to_text(base, outlines)	
save_masks(images, masks, flows, file_names)	save masks + nicely plotted segmentation image to png and/or tiff
save_server([parent, filename])	Uploads a *_seg.npy file to the bucket.
save_to_png(images, masks, flows, file_names)	deprecated (runs io.save_masks with png=True)
write_links(savedir, basename, links)	Write label link file.

check_dir

cellpose_omni.io.**check_dir**(*path*)

get_image_files

cellpose_omni.io.**get_image_files**(*folder*, *mask_filter='_masks'*, *img_filter=None*, *look_one_level_down=False*, *extensions=['png', 'jpg', 'jpeg', 'tif', 'tiff']*, *pattern=None*)

find all images in a folder and if look_one_level_down all subfolders

get_label_files

cellpose_omni.io.**get_label_files**(*img_names*, *label_filter='_cp_masks'*, *img_filter=''*, *ext=None*, *dir_above=False*, *subfolder=''*, *parent=None*, *flows=False*, *links=False*)

Get the corresponding labels and flows for the given file images. If no extension is given, looks for TIF, TIFF, and PNG. If multiple are found, the first in the list is returned. If extension is given, no checks for file existence are made - useful for finding nonstandard output like txt or npy.

Parameters

- **img_names** (*list, str*) -- list of full image file paths
- **label_filter** (*str*) -- the label filter sufix, defaults to _cp_masks can be _flows, _ncolor, etc.
- **ext** (*str*) -- the label extension can be .tif, .png, .txt, etc.
- **img_filter** (*str*) -- the image filter suffix, e.g. _img
- **dir_above** (*bool*) -- whether or not masks are stored in the image parent folder
- **subfolder** (*str*) -- the name of the subfolder where the labels are stored
- **parent** (*str*) -- parent folder or list of folders where masks are stored, if different from images
- **flows** (*Bool*) -- whether or not to search for and return stored flows
- **links** (*bool*) -- whether or not to search for and return stored link files

Return type
list of all absolute label paths (str)

getname

cellpose_omni.io.**getname**(*path*, *suffix=''*)

imread

cellpose_omni.io.**imread**(*filename*)

imsave

cellpose_omni.io.**imsave**(*filename*, *arr*)

imwrite

cellpose_omni.io.**imwrite**(*filename*, *arr*, ***kwargs*)

load_links

cellpose_omni.io.**load_links**(*filename*)

Read a txt or csv file with label links. These should look like:

1,2 1,3 4,7 6,19 . . .

Returns links as a set of tuples.

load_train_test_data

cellpose_omni.io.**load_train_test_data**(*train_dir*, *test_dir=None*, *image_filter=''*, *mask_filter='_masks'*, *unet=False*, *look_one_level_down=True*, *omni=False*, *do_links=True*)

Loads the training and optional test data for training runs.

logger_setup

cellpose_omni.io.**logger_setup**(*verbose=False*)

masks_flows_to_seg

cellpose_omni.io.**masks_flows_to_seg**(*images*, *masks*, *flows*, *diams*, *file_names*, *channels=None*)

save output of model eval to be loaded in GUI

can be list output (run on multiple images) or single output (run on single image)

saved to file_names[k]+'_seg.npy'

Parameters

- **images** (*(list of) 2D or 3D arrays*) -- images input into cellpose
- **masks** (*(list of) 2D arrays, int*) -- masks output from cellpose_omni.eval, where 0=NO masks; 1,2,...=mask labels
- **flows** (*(list of) list of ND arrays*) -- flows output from cellpose_omni.eval
- **diams** (*float array*) -- diameters used to run Cellpose
- **file_names** (*(list of) str*) -- names of files of images
- **channels** (*list of int (optional, default None)*) -- channels used to run Cellpose

outlines_to_text

cellpose_omni.io.**outlines_to_text**(*base*, *outlines*)

save_masks

cellpose_omni.io.**save_masks**(*images*, *masks*, *flows*, *file_names*, *png=True*, *tif=False*, *suffix=''*, *save_flows=False*, *save_outlines=False*, *outline_col=[1, 0, 0]*, *save_ncolor=False*, *dir_above=False*, *in_folders=False*, *savedir=None*, *save_txt=True*, *save_plot=True*, *omni=True*, *channel_axis=None*, *channels=None*)

save masks + nicely plotted segmentation image to png and/or tiff

if png, masks[k] for images[k] are saved to file_names[k]+'_cp_masks.png'

if tif, masks[k] for images[k] are saved to file_names[k]+'_cp_masks.tif'

if png and matplotlib installed, full segmentation figure is saved to file_names[k]+'_cp.png'

only tif option works for 3D data

Parameters

- **images** (*(list of) 2D, 3D or 4D arrays*) -- images input into cellpose
- **masks** (*(list of) 2D arrays, int*) -- masks output from cellpose_omni.eval, where 0=NO masks; 1,2,...=mask labels
- **flows** (*(list of) list of ND arrays*) -- flows output from cellpose_omni.eval
- **file_names** (*(list of) str*) -- names of files of images
- **savedir** (*str*) -- absolute path where images will be saved. Default is none (saves to image directory)
- **save_flows** (*bool*) -- Can choose which outputs/views to save. ncolor is a 4 (or 5, if 4 takes too long) index version of the labels that is way easier to visualize than having hundreds of unique colors that may be similar and touch. Any color map can be applied to it (0,1,2,3,4,...).
- **save_outlines** (*bool*) -- Can choose which outputs/views to save. ncolor is a 4 (or 5, if 4 takes too long) index version of the labels that is way easier to visualize than having hundreds of unique colors that may be similar and touch. Any color map can be applied to it (0,1,2,3,4,...).
- **save_ncolor** (*bool*) -- Can choose which outputs/views to save. ncolor is a 4 (or 5, if 4 takes too long) index version of the labels that is way easier to visualize than having hundreds of unique colors that may be similar and touch. Any color map can be applied to it (0,1,2,3,4,...).
- **save_txt** (*bool*) -- Can choose which outputs/views to save. ncolor is a 4 (or 5, if 4 takes too long) index version of the labels that is way easier to visualize than having hundreds of unique colors that may be similar and touch. Any color map can be applied to it (0,1,2,3,4,...).

save_server

cellpose_omni.io.**save_server**(*parent=None*, *filename=None*)

Uploads a *_seg.npy file to the bucket.

Parameters

- **parent** (*PyQt.MainWindow (optional, default None)*) -- GUI window to grab file info from
- **filename** (*str (optional, default None)*) -- if no GUI, send this file to server

save_to_png

cellpose_omni.io.**save_to_png**(*images*, *masks*, *flows*, *file_names*)

deprecated (runs io.save_masks with png=True)

does not work for 3D images

write_links

cellpose_omni.io.**write_links**(*savedir*, *basename*, *links*)

Write label link file. See load_links() for its output format.

Parameters

- **savedir** (*string*) -- directory in which to save
- **basename** (*string*) -- file name base to which _links.txt is appended.
- **links** (*set*) -- set of label tuples {(x,y),(z,w),...}

14.2.6 cellpose_omni.plot

disk(med, r, Ly, Lx)	returns pixels of disk with radius r and center med
dx_to_circ(dP[, transparency, mask, ...])	dP is 2 x Y x X => 'optic' flow representation
image_to_rgb(img0[, channels, channel_axis, ...])	image is 2 x Ly x Lx or Ly x Lx x 2 - change to RGB Ly x Lx x 3
interesting_patch(mask[, bsize])	get patch of size bsize x bsize with most masks
mask_overlay(img, masks[, colors, omni])	overlay masks on image (set image to grayscale)
mask_rgb(masks[, colors])	masks in random rgb colors
outline_view(img0, maski[, boundaries, ...])	Generates a red outline overlay onto image.
show_segmentation(fig, img, maski, flowi[, ...])	plot segmentation results (like on website)

disk

cellpose_omni.plot.**disk**(*med*, *r*, *Ly*, *Lx*)

returns pixels of disk with radius r and center med

dx_to_circ

cellpose_omni.plot.**dx_to_circ**(*dP*, *transparency=False*, *mask=None*, *sinebow=True*, *norm=True*)

dP is 2 x Y x X => 'optic' flow representation

Parameters

- **dP** (*2xLyxLx array*) -- Flow field components [dy,dx]
- **transparency** (*bool, default False*) -- magnitude of flow controls opacity, not lightness (clear background)
- **mask** (*2D array*) -- Multiplies each RGB component to suppress noise

image_to_rgb

cellpose_omni.plot.**image_to_rgb**(*img0*, *channels=None*, *channel_axis=- 1*, *omni=False*)

image is 2 x Ly x Lx or Ly x Lx x 2 - change to RGB Ly x Lx x 3

interesting_patch

cellpose_omni.plot.**interesting_patch**(*mask*, *bsize=130*)

get patch of size bsize x bsize with most masks

mask_overlay

cellpose_omni.plot.**mask_overlay**(*img*, *masks*, *colors=None*, *omni=False*)

overlay masks on image (set image to grayscale)

Parameters

- **img** (*int or float, 2D or 3D array*) -- img is of size [Ly x Lx (x nchan)]
- **masks** (*int, 2D array*) -- masks where 0=NO masks; 1,2,...=mask labels
- **colors** (*int, 2D array (optional, default None)*) -- size [nmasks x 3], each entry is a color in 0-255 range

Returns

RGB -- array of masks overlaid on grayscale image

Return type

uint8, 3D array

mask_rgb

cellpose_omni.plot.**mask_rgb**(*masks*, *colors=None*)

masks in random rgb colors

Parameters

- **masks** (*int, 2D array*) -- masks where 0=NO masks; 1,2,...=mask labels
- **colors** (*int, 2D array (optional, default None)*) -- size [nmasks x 3], each entry is a color in 0-255 range

Returns

RGB -- array of masks overlaid on grayscale image

Return type

uint8, 3D array

outline_view

cellpose_omni.plot.**outline_view**(*img0*, *maski*, *boundaries=None*, *color=[1, 0, 0]*, *channels=None*, *channel_axis=- 1*, *mode='inner'*, *connectivity=2*, *skip_formatting=False*)

Generates a red outline overlay onto image.

show_segmentation

cellpose_omni.plot.**show_segmentation**(*fig*, *img*, *maski*, *flowi*, *bdi=None*, *channels=None*, *file_name=None*, *omni=False*, *seg_norm=False*, *bg_color=None*, *outline_color=[1, 0, 0]*, *img_colors=None*, *channel_axis=- 1*, *display=True*, *interpolation='bilinear'*)

plot segmentation results (like on website)

Can save each panel of figure with file_name option. Use channels option if img input is not an RGB image with 3 channels.

Parameters

- **fig** (*matplotlib.pyplot.figure*) -- figure in which to make plot
- **img** (*2D or 3D array*) -- image input into cellpose
- **maski** (*int, 2D array*) -- for image k, masks[k] output from cellpose_omni.eval, where 0=NO masks; 1,2,...=mask labels
- **flowi** (*int, 2D array*) -- for image k, flows[k][0] output from cellpose_omni.eval (RGB of flows)
- **channels** (*list of int (optional, default [0,0])*) -- channels used to run Cellpose, no need to use if image is RGB
- **file_name** (*str (optional, default None)*) -- file name of image, if file_name is not None, figure panels are saved
- **omni** (*bool (optional, default False)*) -- use omni version of normalize99, image_to_rgb
- **seg_norm** (*bool (optional, default False)*) -- improve cell visibility under labels
- **bg_color** (*float (Optional, default none)*) -- background color to draw behind flow (visible if flow transparency is on)

- **img_colors** (*NDarray, float (Optional, default none)*) -- colors to which each image channel will be mapped (multichannel defaults to sinebow)

14.2.7 cellpose_omni.metrics

aggregated_jaccard_index(masks_true, masks_pred)	AJI = intersection of all matched masks / union of all masks
average_precision(masks_true, masks_pred[, ...])	average precision estimation: AP = TP / (TP + FP + FN)
boundary_scores(masks_true, masks_pred, scales)	boundary precision / recall / Fscore
flow_error(maski, dP_net[, use_gpu, device])	error in flows from predicted masks vs flows predicted by network run on image
mask_ious(masks_true, masks_pred)	return best-matched masks

aggregated_jaccard_index

cellpose_omni.metrics.**aggregated_jaccard_index**(*masks_true*, *masks_pred*)

AJI = intersection of all matched masks / union of all masks

Parameters

- **masks_true** (*list of ND-arrays (int) or ND-array (int)*) -- where 0=NO masks; 1,2... are mask labels
- **masks_pred** (*list of ND-arrays (int) or ND-array (int)*) -- ND-array (int) where 0=NO masks; 1,2... are mask labels

Returns

aji

Return type

aggregated jaccard index for each set of masks

average_precision

cellpose_omni.metrics.**average_precision**(*masks_true*, *masks_pred*, *threshold=[0.5, 0.75, 0.9]*)

average precision estimation: AP = TP / (TP + FP + FN)

This function is based heavily on the *fast* stardist matching functions (https://github.com/mpicbg-csbd/stardist/blob/master/stardist/matching.py)

Parameters

- **masks_true** (*list of ND-arrays (int) or ND-array (int)*) -- where 0=NO masks; 1,2... are mask labels
- **masks_pred** (*list of ND-arrays (int) or ND-array (int)*) -- ND-array (int) where 0=NO masks; 1,2... are mask labels

Returns

- **ap** (*array [len(masks_true) x len(threshold)]*) -- average precision at thresholds
- **tp** (*array [len(masks_true) x len(threshold)]*) -- number of true positives at thresholds
- **fp** (*array [len(masks_true) x len(threshold)]*) -- number of false positives at thresholds
- **fn** (*array [len(masks_true) x len(threshold)]*) -- number of false negatives at thresholds

boundary_scores

cellpose_omni.metrics.**boundary_scores**(*masks_true*, *masks_pred*, *scales*)

boundary precision / recall / Fscore

flow_error

cellpose_omni.metrics.**flow_error**(*maski*, *dP_net*, *use_gpu=False*, *device=None*)

error in flows from predicted masks vs flows predicted by network run on image

This function serves to benchmark the quality of masks, it works as follows 1. The predicted masks are used to create a flow diagram 2. The mask-flows are compared to the flows that the network predicted

If there is a discrepancy between the flows, it suggests that the mask is incorrect. Masks with flow_errors greater than 0.4 are discarded by default. Setting can be changed in Cellpose.eval or CellposeModel.eval.

Parameters

- **maski** (*ND-array (int)*) -- masks produced from running dynamics on dP_net, where 0=NO masks; 1,2... are mask labels
- **dP_net** (*ND-array (float)*) -- ND flows where dP_net.shape[1:] = maski.shape

Returns

- **flow_errors** (*float array with length maski.max()*) -- mean squared error between predicted flows and flows from masks
- **dP_masks** (*ND-array (float)*) -- ND flows produced from the predicted masks

mask_ious

cellpose_omni.metrics.**mask_ious**(*masks_true*, *masks_pred*)

return best-matched masks

14.2.8 cellpose_omni.dynamics

compute_masks(dP, cellprob[, bd, p, inds, ...])	compute masks using dynamics from dP, cellprob, and boundary
follow_flows(dP[, mask, inds, niter, ...])	define pixels and run dynamics to recover masks in 2D
get_masks(p[, iscell, rpad, flows, ...])	create masks using pixel convergence after running dynamics
labels_to_flows(labels[, files, use_gpu, ...])	convert labels (list of masks or flows) to flows for training model
map_coordinates(I, yc, xc, Y)	bilinear interpolation of image 'I' in-place with ycoordinates yc and xcoordinates xc to Y
masks_to_flows(masks[, use_gpu, device])	convert masks to flows using diffusion from center pixel
masks_to_flows_cpu(masks[, device])	convert masks to flows using diffusion from center pixel Center of masks where diffusion starts is defined to be the closest pixel to the median of all pixels that is inside the mask.
masks_to_flows_gpu(masks[, device])	convert masks to flows using diffusion from center pixel Center of masks where diffusion starts is defined using COM :param masks: labelled masks 0=NO masks; 1,2,...=mask labels :type masks: int, 2D or 3D array
remove_bad_flow_masks(masks, flows[, ...])	remove masks which have inconsistent flows
steps2D(p, dP, inds, niter[, omni, calc_trace])	run dynamics of pixels to recover masks in 2D
steps2D_interp(p, dP, niter[, use_gpu, ...])	
steps3D(p, dP, inds, niter)	run dynamics of pixels to recover masks in 3D

compute_masks

cellpose_omni.dynamics.**compute_masks**(*dP*, *cellprob*, *bd=None*, *p=None*, *inds=None*, *niter=200*, *mask_threshold=0.0*, *diam_threshold=12.0*, *flow_threshold=0.4*, *interp=True*, *do_3D=False*, *min_size=15*, *resize=None*, *verbose=False*, *use_gpu=False*, *device=None*, *nclasses=3*, *calc_trace=False*)

compute masks using dynamics from dP, cellprob, and boundary

follow_flows

cellpose_omni.dynamics.**follow_flows**(*dP*, *mask=None*, *inds=None*, *niter=200*, *interp=True*, *use_gpu=True*, *device=None*, *omni=False*, *calc_trace=False*)

define pixels and run dynamics to recover masks in 2D

Pixels are meshgrid. Only pixels with non-zero cell-probability are used (as defined by inds)

Parameters

- **dP** (*float32, 3D or 4D array*) -- flows [axis x Ly x Lx] or [axis x Lz x Ly x Lx]
- **mask** (*(optional, default None)*) -- pixel mask to seed masks. Useful when flows have low magnitudes.
- **niter** (*int (optional, default 200)*) -- number of iterations of dynamics to run
- **interp** (*bool (optional, default True)*) -- interpolate during 2D dynamics (not available in 3D) (in previous versions + paper it was False)

- **use_gpu** (*bool (optional, default False)*) -- use GPU to run interpolated dynamics (faster than CPU)

Returns
p -- final locations of each pixel after dynamics

Return type
float32, 3D array

get_masks

cellpose_omni.dynamics.**get_masks**(*p*, *iscell=None*, *rpad=20*, *flows=None*, *threshold=0.4*, *use_gpu=False*, *device=None*)

create masks using pixel convergence after running dynamics

Makes a histogram of final pixel locations p, initializes masks at peaks of histogram and extends the masks from the peaks so that they include all pixels with more than 2 final pixels p. Discards masks with flow errors greater than the threshold.

Parameters

- **p** (*float32, 3D or 4D array*) -- final locations of each pixel after dynamics, size [axis x Ly x Lx] or [axis x Lz x Ly x Lx].
- **iscell** (*bool, 2D or 3D array*) -- if iscell is not None, set pixels that are iscell False to stay in their original location.
- **rpad** (*int (optional, default 20)*) -- histogram edge padding
- **threshold** (*float (optional, default 0.4)*) -- masks with flow error greater than threshold are discarded (if flows is not None)
- **flows** (*float, 3D or 4D array (optional, default None)*) -- flows [axis x Ly x Lx] or [axis x Lz x Ly x Lx]. If flows is not None, then masks with inconsistent flows are removed using *remove_bad_flow_masks*.

Returns
M0 -- masks with inconsistent flow masks removed, 0=NO masks; 1,2,...=mask labels, size [Ly x Lx] or [Lz x Ly x Lx]

Return type
int, 2D or 3D array

labels_to_flows

cellpose_omni.dynamics.**labels_to_flows**(*labels*, *files=None*, *use_gpu=False*, *device=None*, *redo_flows=False*, *links=None*, *dim=2*)

convert labels (list of masks or flows) to flows for training model

if files is not None, flows are saved to files to be reused

Parameters
labels (*list of ND-arrays*) -- labels[k] can be 2D or 3D, if [3 x Ly x Lx] then it is assumed that flows were precomputed. Otherwise labels[k][0] or labels[k] (if 2D) is used to create flows and cell probabilities.

Returns
flows -- flows[k][0] is labels[k], flows[k][1] is cell distance transform, flows[k][2] is Y flow, flows[k][3] is X flow, and flows[k][4] is heat distribution

Return type
list of [4 x Ly x Lx] arrays

map_coordinates

cellpose_omni.dynamics.**map_coordinates**(*I*, *yc*, *xc*, *Y*)

bilinear interpolation of image 'I' in-place with ycoordinates yc and xcoordinates xc to Y

Parameters

- **I** (*C x Ly x Lx*) --
- **yc** (*ni*) -- new y coordinates
- **xc** (*ni*) -- new x coordinates
- **Y** (*C x ni*) -- I sampled at (yc,xc)

masks_to_flows

cellpose_omni.dynamics.**masks_to_flows**(*masks*, *use_gpu=False*, *device=None*)

convert masks to flows using diffusion from center pixel

Center of masks where diffusion starts is defined to be the closest pixel to the median of all pixels that is inside the mask. Result of diffusion is converted into flows by computing the gradients of the diffusion density map.

Parameters
masks (*int, 2D or 3D array*) -- labelled masks 0=NO masks; 1,2,...=mask labels

Returns

- **mu** (*float, 3D or 4D array*) -- flows in Y = mu[-2], flows in X = mu[-1]. if masks are 3D, flows in Z = mu[0].
- **mu_c** (*float, 2D or 3D array*) -- for each pixel, the distance to the center of the mask in which it resides

masks_to_flows_cpu

cellpose_omni.dynamics.**masks_to_flows_cpu**(*masks*, *device=None*)

convert masks to flows using diffusion from center pixel Center of masks where diffusion starts is defined to be the closest pixel to the median of all pixels that is inside the mask. Result of diffusion is converted into flows by computing the gradients of the diffusion density map. :param masks: labelled masks 0=NO masks; 1,2,...=mask labels :type masks: int, 2D array

Returns

- **mu** (*float, 3D array*) -- flows in Y = mu[-2], flows in X = mu[-1]. if masks are 3D, flows in Z = mu[0].
- **mu_c** (*float, 2D array*) -- for each pixel, the distance to the center of the mask in which it resides

masks_to_flows_gpu

cellpose_omni.dynamics.**masks_to_flows_gpu**(*masks*, *device=None*)

convert masks to flows using diffusion from center pixel Center of masks where diffusion starts is defined using COM :param masks: labelled masks 0=NO masks; 1,2,...=mask labels :type masks: int, 2D or 3D array

Returns

- **mu** (*float, 3D or 4D array*) -- flows in Y = mu[-2], flows in X = mu[-1]. if masks are 3D, flows in Z = mu[0].
- **mu_c** (*float, 2D or 3D array*) -- for each pixel, the distance to the center of the mask in which it resides

remove_bad_flow_masks

cellpose_omni.dynamics.**remove_bad_flow_masks**(*masks*, *flows*, *threshold=0.4*, *use_gpu=False*, *device=None*)

remove masks which have inconsistent flows

Uses metrics.flow_error to compute flows from predicted masks and compare flows to predicted flows from network. Discards masks with flow errors greater than the threshold.

Parameters

- **masks** (*int, 2D or 3D array*) -- labelled masks, 0=NO masks; 1,2,...=mask labels, size [Ly x Lx] or [Lz x Ly x Lx]
- **flows** (*float, 3D or 4D array*) -- flows [axis x Ly x Lx] or [axis x Lz x Ly x Lx]
- **threshold** (*float (optional, default 0.4)*) -- masks with flow error greater than threshold are discarded.

Returns

masks -- masks with inconsistent flow masks removed, 0=NO masks; 1,2,...=mask labels, size [Ly x Lx] or [Lz x Ly x Lx]

Return type

int, 2D or 3D array

steps2D

cellpose_omni.dynamics.**steps2D**(*p*, *dP*, *inds*, *niter*, *omni=False*, *calc_trace=False*)

run dynamics of pixels to recover masks in 2D

Euler integration of dynamics dP for niter steps

Parameters

- **p** (*float32, 3D array*) -- pixel locations [axis x Ly x Lx] (start at initial meshgrid)
- **dP** (*float32, 3D array*) -- flows [axis x Ly x Lx]
- **inds** (*int32, 2D array*) -- non-zero pixels to run dynamics on [npixels x 2]
- **niter** (*int32*) -- number of iterations of dynamics to run

Returns

p -- final locations of each pixel after dynamics

Return type
float32, 3D array

steps2D_interp

cellpose_omni.dynamics.**steps2D_interp**(*p*, *dP*, *niter*, *use_gpu=False*, *device=None*, *omni=False*, *calc_trace=False*)

steps3D

cellpose_omni.dynamics.**steps3D**(*p*, *dP*, *inds*, *niter*)

run dynamics of pixels to recover masks in 3D

Euler integration of dynamics dP for niter steps

Parameters

- **p** (*float32, 4D array*) -- pixel locations [axis x Lz x Ly x Lx] (start at initial meshgrid)
- **dP** (*float32, 4D array*) -- flows [axis x Lz x Ly x Lx]
- **inds** (*int32, 2D array*) -- non-zero pixels to run dynamics on [npixels x 3]
- **niter** (*int32*) -- number of iterations of dynamics to run

Returns
p -- final locations of each pixel after dynamics

Return type
float32, 4D array

14.2.9 cellpose_omni.transforms

average_tiles(y, ysub, xsub, Ly, Lx)	average results of network over tiles
convert_image(x, channels[, channel_axis, ...])	return image with z first, channels last and normalized intensities
make_tiles(imgi[, bsize, augment, tile_overlap])	make tiles of image to run at test-time
move_axis(img[, m_axis, first])	move axis m_axis to first or last position
move_axis_new(a, axis, pos)	Move ndarray axis to new location, preserving order of other axes.
move_min_dim(img[, force])	move minimum dimension last as channels if < 10, or force==True
normalize99(Y[, lower, upper, omni])	normalize image so 0.0 is 0.01st percentile and 1.0 is 99.99th percentile
normalize_field(mu[, omni])	
normalize_img(img[, axis, invert, omni])	normalize each channel of the image so that so that 0.0=1st percentile and 1.0=99th percentile of image intensities
original_random_rotate_and_resize(X[, Y, ...])	augmentation by random rotation and resizing X and Y are lists or arrays of length nimg, with dims channels x Ly x Lx (channels optional)
pad_image_ND(img0[, div, extra, dim])	pad image for test-time so that its dimensions are a multiple of 16 (2D or 3D)
random_rotate_and_resize(X[, Y, ...])	augmentation by random rotation and resizing
reshape(data[, channels, chan_first, ...])	reshape data using channels
reshape_and_normalize_data(train_data[, ...])	inputs converted to correct shapes for *training* and rescaled so that 0.0=1st percentile and 1.0=99th percentile of image intensities in each channel
reshape_train_test(train_data, train_labels, ...)	check sizes and reshape train and test data for training
resize_image(img0[, Ly, Lx, rsz, ...])	resize image for computing flows / unresize for computing dynamics
unaugment_tiles(y[, unet])	reverse test-time augmentations for averaging
update_axis(m_axis, to_squeeze, ndim)	

average_tiles

cellpose_omni.transforms.**average_tiles**(*y*, *ysub*, *xsub*, *Ly*, *Lx*)

average results of network over tiles

Parameters

- **y** (*float, [ntiles x nclasses x bsize x bsize]*) -- output of cellpose network for each tile
- **ysub** (*list*) -- list of arrays with start and end of tiles in Y of length ntiles
- **xsub** (*list*) -- list of arrays with start and end of tiles in X of length ntiles
- **Ly** (*int*) -- size of pre-tiled image in Y (may be larger than original image if image size is less than bsize)
- **Lx** (*int*) -- size of pre-tiled image in X (may be larger than original image if image size is less than bsize)

Returns
yf -- network output averaged over tiles

Return type
float32, [nclasses x Ly x Lx]

convert_image

cellpose_omni.transforms.**convert_image**(*x*, *channels*, *channel_axis=None*, *z_axis=None*, *do_3D=False*, *normalize=True*, *invert=False*, *nchan=2*, *dim=2*, *omni=False*)

return image with z first, channels last and normalized intensities

make_tiles

cellpose_omni.transforms.**make_tiles**(*imgi*, *bsize=224*, *augment=False*, *tile_overlap=0.1*)

make tiles of image to run at test-time

if augmented, tiles are flipped and tile_overlap=2.

- original
- flipped vertically
- flipped horizontally
- flipped vertically and horizontally

Parameters

- **imgi** (*float32*) -- array that's nchan x Ly x Lx
- **bsize** (*float (optional, default 224)*) -- size of tiles
- **augment** (*bool (optional, default False)*) -- flip tiles and set tile_overlap=2.
- **tile_overlap** (*float (optional, default 0.1)*) -- fraction of overlap of tiles

Returns

- **IMG** (*float32*) -- array that's ntiles x nchan x bsize x bsize
- **ysub** (*list*) -- list of arrays with start and end of tiles in Y of length ntiles
- **xsub** (*list*) -- list of arrays with start and end of tiles in X of length ntiles

move_axis

cellpose_omni.transforms.**move_axis**(*img*, *m_axis=- 1*, *first=True*)

move axis m_axis to first or last position

move_axis_new

cellpose_omni.transforms.**move_axis_new**(*a*, *axis*, *pos*)

Move ndarray axis to new location, preserving order of other axes.

move_min_dim

cellpose_omni.transforms.**move_min_dim**(*img*, *force=False*)

move minimum dimension last as channels if < 10, or force==True

normalize99

cellpose_omni.transforms.**normalize99**(*Y*, *lower=0.01*, *upper=99.99*, *omni=False*)

normalize image so 0.0 is 0.01st percentile and 1.0 is 99.99th percentile

normalize_field

cellpose_omni.transforms.**normalize_field**(*mu*, *omni=False*)

normalize_img

cellpose_omni.transforms.**normalize_img**(*img*, *axis=- 1*, *invert=False*, *omni=False*)

normalize each channel of the image so that so that 0.0=1st percentile and 1.0=99th percentile of image intensities

optional inversion

Parameters

- **img** (*ND-array (at least 3 dimensions)*) --
- **axis** (*channel axis to loop over for normalization*) --

Returns

img -- normalized image of same size

Return type

ND-array, float32

original_random_rotate_and_resize

cellpose_omni.transforms.**original_random_rotate_and_resize**(*X*, *Y=None*, *scale_range=1.0*, *xy=(224, 224)*, *do_flip=True*, *rescale=None*, *unet=False*)

augmentation by random rotation and resizing X and Y are lists or arrays of length nimg, with dims channels x Ly x Lx (channels optional)

Parameters

- **X** (*LIST of ND-arrays, float*) -- list of image arrays of size [nchan x Ly x Lx] or [Ly x Lx]

- **Y** (*LIST of ND-arrays, float (optional, default None)*) -- list of image labels of size [nlabels x Ly x Lx] or [Ly x Lx]. The 1st channel of Y is always nearest-neighbor interpolated (assumed to be masks or 0-1 representation). If Y.shape[0]==3 and not unet, then the labels are assumed to be [cell probability, Y flow, X flow]. If unet, second channel is dist_to_bound.
- **scale_range** (*float (optional, default 1.0)*) -- Range of resizing of images for augmentation. Images are resized by (1-scale_range/2) + scale_range * np.random.rand()
- **xy** (*tuple, int (optional, default (224,224))*) -- size of transformed images to return
- **do_flip** (*bool (optional, default True)*) -- whether or not to flip images horizontally
- **rescale** (*array, float (optional, default None)*) -- how much to resize images by before performing augmentations
- **unet** (*bool (optional, default False)*) --

Returns

- **imgi** (*ND-array, float*) -- transformed images in array [nimg x nchan x xy[0] x xy[1]]
- **lbl** (*ND-array, float*) -- transformed labels in array [nimg x nchan x xy[0] x xy[1]]
- **scale** (*array, float*) -- amount by which each image was resized

pad_image_ND

cellpose_omni.transforms.**pad_image_ND**(*img0*, *div=16*, *extra=1*, *dim=2*)

pad image for test-time so that its dimensions are a multiple of 16 (2D or 3D)

Parameters

- **img0** (*ND-array*) -- image of size [nchan (x Lz) x Ly x Lx]
- **div** (*int (optional, default 16)*) --

Returns

- **I** (*ND-array*) -- padded image
- **ysub** (*array, int*) -- yrange of pixels in I corresponding to img0
- **xsub** (*array, int*) -- xrange of pixels in I corresponding to img0

random_rotate_and_resize

cellpose_omni.transforms.**random_rotate_and_resize**(*X*, *Y=None*, *scale_range=1.0*, *gamma_range=[0.5, 4]*, *tyx=None*, *do_flip=True*, *rescale=None*, *unet=False*, *inds=None*, *omni=False*, *dim=2*, *nchan=1*, *nclasses=3*, *device=None*)

augmentation by random rotation and resizing

X and Y are lists or arrays of length nimg, with dims channels x Ly x Lx (channels optional)

Parameters

- **X** (*LIST of ND-arrays, float*) -- list of image arrays of size [nchan x Ly x Lx] or [Ly x Lx]

- **Y** (*LIST of ND-arrays, float (optional, default None)*) -- list of image labels of size [nlabels x Ly x Lx] or [Ly x Lx]. The 1st channel of Y is always nearest-neighbor interpolated (assumed to be masks or 0-1 representation). If Y.shape[0]==3 and not unet, then the labels are assumed to be [cell probability, Y flow, X flow]. If unet, second channel is dist_to_bound.
- **scale_range** (*float (optional, default 1.0)*) -- Range of resizing of images for augmentation. Images are resized by (1-scale_range/2) + scale_range * np.random.rand()
- **gamma_range** (*float (optional, default 0.5)*) -- Images are gamma-adjusted im**gamma for gamma in (1-gamma_range,1+gamma_range)
- **xy** (*tuple, int (optional, default (224,224))*) -- size of transformed images to return
- **do_flip** (*bool (optional, default True)*) -- whether or not to flip images horizontally
- **rescale** (*array, float (optional, default None)*) -- how much to resize images by before performing augmentations
- **unet** (*bool (optional, default False)*) --

Returns

- **imgi** (*ND-array, float*) -- transformed images in array [nimg x nchan x xy[0] x xy[1]]
- **lbl** (*ND-array, float*) -- transformed labels in array [nimg x nchan x xy[0] x xy[1]]
- **scale** (*array, float*) -- amount each image was resized by

reshape

cellpose_omni.transforms.**reshape**(*data*, *channels=[0, 0]*, *chan_first=False*, *channel_axis=0*)

reshape data using channels

Parameters

- **data** (*numpy array that's (Z x) Ly x Lx x nchan*) -- if data.ndim==8 and data.shape[0]<8, assumed to be nchan x Ly x Lx
- **channels** (*list of int of length 2 (optional, default [0,0])*) -- First element of list is the channel to segment (0=grayscale, 1=red, 2=green, 3=blue). Second element of list is the optional nuclear channel (0=none, 1=red, 2=green, 3=blue). For instance, to train on grayscale images, input [0,0]. To train on images with cells in green and nuclei in blue, input [2,3].
- **channel_axis** (*int, default 0*) -- the axis that corresponds to channels (usually 0 or -1)

Returns
data

Return type
numpy array that's (Z x) Ly x Lx x nchan (if chan_first==False)

reshape_and_normalize_data

cellpose_omni.transforms.**reshape_and_normalize_data**(*train_data*, *test_data=None*, *channels=None*, *channel_axis=0*, *normalize=True*, *omni=False*, *dim=2*)

inputs converted to correct shapes for *training* and rescaled so that 0.0=1st percentile and 1.0=99th percentile of image intensities in each channel

Parameters

- **train_data** (*list of ND-arrays, float*) -- list of training images of size [Ly x Lx], [nchan x Ly x Lx], or [Ly x Lx x nchan]
- **test_data** (*list of ND-arrays, float (optional, default None)*) -- list of testing images of size [Ly x Lx], [nchan x Ly x Lx], or [Ly x Lx x nchan]
- **channels** (*list of int of length 2 (optional, default None)*) -- First element of list is the channel to segment (0=grayscale, 1=red, 2=green, 3=blue). Second element of list is the optional nuclear channel (0=none, 1=red, 2=green, 3=blue). For instance, to train on grayscale images, input [0,0]. To train on images with cells in green and nuclei in blue, input [2,3].
- **normalize** (*bool (optional, True)*) -- normalize data so 0.0=1st percentile and 1.0=99th percentile of image intensities in each channel

Returns

- **train_data** (*list of ND-arrays, float*) -- list of training images of size [2 x Ly x Lx]
- **test_data** (*list of ND-arrays, float (optional, default None)*) -- list of testing images of size [2 x Ly x Lx]
- **run_test** (*bool*) -- whether or not test_data was correct size and is useable during training

reshape_train_test

cellpose_omni.transforms.**reshape_train_test**(*train_data*, *train_labels*, *test_data*, *test_labels*, *channels*, *channel_axis=0*, *normalize=True*, *dim=2*, *omni=False*)

check sizes and reshape train and test data for training

resize_image

cellpose_omni.transforms.**resize_image**(*img0*, *Ly=None*, *Lx=None*, *rsz=None*, *interpolation=1*, *no_channels=False*)

resize image for computing flows / unresize for computing dynamics

Parameters

- **img0** (*ND-array*) -- image of size [Y x X x nchan] or [Lz x Y x X x nchan] or [Lz x Y x X]
- **Ly** (*int, optional*) --
- **Lx** (*int, optional*) --
- **rsz** (*float, optional*) -- resize coefficient(s) for image; if Ly is None then rsz is used
- **interpolation** (*cv2 interp method (optional, default cv2.INTER_LINEAR)*) --

Returns
imgs -- image of size [Ly x Lx x nchan] or [Lz x Ly x Lx x nchan]

Return type
ND-array

unaugment_tiles

cellpose_omni.transforms.**unaugment_tiles**(*y*, *unet=False*)

reverse test-time augmentations for averaging

Parameters

- **y** (*float32*) -- array that's ntiles_y x ntiles_x x chan x Ly x Lx where chan = (dY, dX, cell prob)
- **unet** (*bool (optional, False)*) -- whether or not unet output or cellpose output

Returns
y

Return type
float32

update_axis

cellpose_omni.transforms.**update_axis**(*m_axis*, *to_squeeze*, *ndim*)

CHAPTER

FIFTEEN

CLI

See *command line examples* for typical use cases.

```
usage: omnipose [image args] [model args] [...]
```

15.1 input image arguments

--dir folder containing data on which to run or train

--look_one_level_down run processing on all subdirectories of current folder

--mxnet use mxnet

--img_filter filter images by this suffix

--channel_axis axis of image which corresponds to image channels

--z_axis axis of image which corresponds to Z dimension

--chan channel to segment; 0: GRAY, 1: RED, 2: GREEN, 3: BLUE. Default: 0

--chan2 nuclear channel (if cyto, optional); 0: NONE, 1: RED, 2: GREEN, 3: BLUE. Default: 0

--invert invert grayscale channel

--all_channels use all channels in image if using own model and images with special channels

--dim number of spatiotemporal dimensions of images (not counting channels). Default: 2

15.2 model arguments

--pretrained_model model to use

--unet run standard unet instead of cellpose flow output

--nclasses number of prediction classes for model (3 for Cellpose, 4 for Omnipose boundary field)

--nchan number of channels on which model is trained

--kernel_size kernel size for maskpool. Starts at 2, higher means more aggressive downsampling.

15.3 algorithm arguments

--omni	Omnipose algorithm (disabled by default)
--affinity_seg	use new affinity segmentation algorithm (disabled by default)
--cluster	DBSCAN clustering. Reduces oversegmentation of thin features (disabled by default)
--no_suppress	Euler integration 1/t suppression reduces oversegmentation but can give undersegmentation in 3D; this flag disables it.
--fast_mode	make code run faster by turning off 4 network averaging and resampling
--no_resample	disable dynamics on full image (makes algorithm faster for images with large diameters)
--no_net_avg	make code run faster by only running 1 network
--no_interp	do not interpolate when running dynamics (was default)
--do_3D	process images as 3D stacks of images (nplanes x nchan x Ly x Lx
--diameter	cell diameter, 0 disables unless sizemodel is present. Default: 0.0
--rescale	image rescaling factor (r = diameter / model diameter)
--stitch_threshold	compute masks in 2D then stitch together masks with IoU>0.9 across planes
--flow_threshold	flow error threshold, 0 turns off this optional QC step. Default: 0.4
--mask_threshold	mask threshold, default is 0, decrease to find more and larger masks
--niter	Number of Euler iterations, enter value to override Omnipose diameter estimation (under/over-segment)
--anisotropy	anisotropy of volume in 3D
--diam_threshold	cell diameter threshold for upscaling before mask rescontruction, default 12
--exclude_on_edges	discard masks which touch edges of image
--min_size	minimum size for masks, helps if small debris is labeled
--max_size	maximum size for masks, helps if background patches are labeled

15.4 output arguments

--save_png	save masks as png
--save_tif	save masks as tif
--no_npy	suppress saving of npy
--savedir	folder to which segmentation results will be saved (defaults to input image directory)
--dir_above	save output folders adjacent to image folder instead of inside it (off by default)
--in_folders	flag to save output in folders (off by default)
--save_flows	whether or not to save RGB images of flows when masks are saved (disabled by default)

--save_outlines	whether or not to save RGB outline images when masks are saved (disabled by default)
--save_ncolor	whether or not to save minimal "n-color" masks (disabled by default
--save_txt	flag to enable txt outlines for ImageJ (disabled by default)
--transparency	store flows with background transparent (alpha=flow magnitude) (disabled by default)

15.5 training arguments

--train	train network using images in dir
--train_size	train size network at end of training
--mask_filter	end string for masks to run on. Default: "_masks"
--test_dir	folder containing test data (optional)
--learning_rate	learning rate. Default: 0.2
--n_epochs	number of epochs. Default: 500
--batch_size	batch size. Default: 8
--num_workers	number of dataloader workers. Default: 0
--dataloader	Use pytorch dataloader instead of older manual loading code.
--min_train_masks	minimum number of masks a training image must have to be used. Default: 1
--residual_on	use residual connections
--style_on	use style vector
--concatenation	concatenate downsampled layers with upsampled layers (off by default which means they are added)
--save_every	number of epochs to skip between saves. Default: 100
--save_each	save the model under a different filename per --save_every epoch for later comparsion
--RAdam	use RAdam instead of SGD
--checkpoint	turn on checkpoints to reduce memory usage
--dropout	Use dropout in training
--tyx	list of yx, zyx, or tyx dimensions for training
--links	Search and use link files for multi-label objects.
--amp	Use Automatic Mixed Precision.
--affinity_field	Use summed affinity instead of distance field.

15.6 hardware arguments

--use_gpu	use gpu if torch or mxnet with cuda installed
--check_mkl	check if mkl working
--mkldnn	for mxnet, force MXNET_SUBGRAPH_BACKEND = "MKLDNN"

15.7 development arguments

--verbose	flag to output extra information (e.g. diameter metrics) for debugging and fine-tuning parameters
--testing	flag to suppress CLI user confirmation for saving output; for test scripts
--timing	flag to output timing information for select modules

CHAPTER

SIXTEEN

AFFINITY SEGMENTATION

This is the term that I think best describes encoding an image segmentation it its most general, information-dense form: an affinity graph. To explain what this is, we will first consider two cells in contact.

16.1 The hierarchy of segmentation encoding

```
# Load image and masks
import string
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rcParams['figure.dpi'] = 600
# mpl.rcParams['facecolor'] = [0]*4

plt.rc('figure', facecolor=[0]*4)

plt.style.use('dark_background')
mpl.use('Agg')

%matplotlib inline

from pathlib import Path
import os
from cellpose_omni import io, plot
import fastremap

import omnipose
omnidir = Path(omnipose.__file__).parent.parent
basedir = os.path.join(omnidir,'docs','_static')
# name = 'ecoli_phase'
name = 'ecoli'
ext = '.tif'
image = io.imread(os.path.join(basedir,name+ext))
masks = io.imread(os.path.join(basedir,name+'_labels'+ext))
slc = omnipose.utils.crop_bbox(masks>0,pad=0)[0]
masks = fastremap.renumber(masks[slc])[0]
image = image[slc]

# Plot a few things

```

(continues on next page)

(continued from previous page)

```
import matplotlib.pyplot as plt
from omnipose.plot import apply_ncolor, plot_edges, imshow
from omnipose import utils
import numpy as np
# import matplotlib_inline
# matplotlib_inline.backend_inline.set_matplotlib_formats('svg')

from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset
from matplotlib.patches import Patch



images = [image, masks>0, apply_ncolor(masks), masks>0]
labels = ['Image\n(phase contrast)', 'Semantic\nsegmentation',
          'Instance\nsegmentation', 'Affinity\nsegmentation']

# Set up the figure and subplots
N = len(images)

f = 1
Y,X = masks.shape[-2:]
M = 1

h,w = masks.shape[-2:]

sf = w
p = 0.0035*w # needs to be defined as fraction of width for aspect ratio to work?
h /= sf
w /= sf

# Calculate positions of subplots
left = np.array([i*(w+p) for i in range(N)])*1.
bottom = np.array([0]*N)*1.
width = np.array([w]*N)*1.
height = np.array([h]*N)*1.

max_w = left[-1]+width[-1]
max_h = bottom[-1]+height[-1]

sw = max_w
sh = max_h

sf = max(sw,sh)
left /= sw
bottom /= sh
width /= sw
height /= sh

# Create figure
s = 6
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(s,s*sh/sw), frameon=False, dpi=300)
# fig.patch.set_facecolor([0]*4)

# Add subplots
axes = []
for i in range(N):
    ax = fig.add_axes([left[i], bottom[i], width[i], height[i]])
    axes.append(ax)

# Iterate over each subplot and set the image, label, and formatting
c =  [0.5]*3
fontsize = 11

# bounds = [40,20,10,10]
bounds = [11,22,8,8]
h,w = masks.shape[-2:]
extent = np.array([0,w,0,h])#-0.5


sy,sx,wy,wx = bounds
zoomslc = tuple([slice(sy,sy+wy),slice(sx,sx+wx)])


cmap='inferno'

zoom = 3
# zoom, f = 5, 0.75
color = [.75]*3
edgecol = [1/3]*3
edgecol = [.75]*3+[.5]
axcol = [0.5]*3
# edgecol = [.5,.75,0]+[2/3]

lw= .2
labelpad = 3
fontsize2 = 8

do_labels = 0

for i, ax in enumerate(axes):

    # inset axes
    axins = zoomed_inset_axes(ax, zoom, loc='lower left',
                              bbox_to_anchor=(-wx/w,-2*wy/h),
                              # bbox_to_anchor=(-wx/w*zoom/2,-zoom*wy/h),
                              # bbox_to_anchor=(-f*zoom*wy/h,-f*wx/w*zoom),


                              bbox_transform=ax.transAxes)

    if i==N-1:
        # ax.invert_yaxis()
```

(continues on next page)

(continued from previous page)

```
        dim = masks.ndim
        shape = masks.shape
        steps, inds, idx, fact, sign = utils.kernel_setup(dim)
        coords = np.nonzero(masks)
        affinity_graph = omnipose.core.masks_to_affinity(masks, coords, steps,
                                                         inds, idx, fact, sign, dim)
        neighbors = utils.get_neighbors(coords,steps,dim,shape)
        summed_affinity, affinity_cmap = plot_edges(shape,affinity_graph,neighbors,
↪coords,
                                                    figsize=1,fig=fig,ax=ax,
↪extent=extent,
                                                    edgecol=edgecol,cmap=cmap,
↪linewidth=lw
                                                   )



        axins.invert_yaxis()
        ax.invert_yaxis()

        summed_affinity, affinity_cmap = plot_edges(shape,affinity_graph,neighbors,
↪coords,
                                                figsize=1,fig=fig,ax=axins,
                                                extent=extent,
                                                edgecol=edgecol,linewidth=lw*zoom,cmap=cmap,
                                                bounds=bounds
                                               )

        axins.set_xlim(zoomslc[1].start, zoomslc[1].stop)
        axins.set_ylim(h-zoomslc[0].start, h-zoomslc[0].stop)


        loc1,loc2 = 4,2
        patch, pp1, pp2 = mark_inset(ax, axins, loc1=loc1, loc2=loc2, fc="none",␣
↪ec=color+[1],zorder=2)
        pp1.loc1 = 4
        pp1.loc2 = 1
        pp2.loc1 = 2
        pp2.loc2 = 3


        N = affinity_cmap.N
        colors = affinity_cmap.colors

        cax = inset_axes(ax, width="50%", height="100%", loc='lower right',
                bbox_to_anchor=(-.05, -0.7, 1, 1), bbox_transform=ax.transAxes,
                borderpad=0)

        # Display the color swatches as an image
        n = np.arange(3,9)
        Nc = len(n)
        cax.imshow(affinity_cmap(n.reshape(1,Nc)))#,vmin=n[0]-1,vmax=n[-1]+1)
```

(continues on next page)

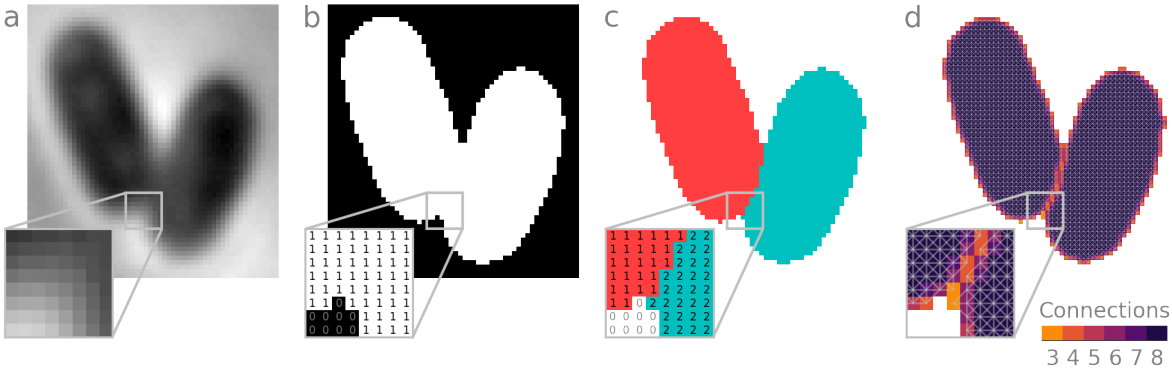
(continued from previous page)

```
        # Set the y ticks and tick labels
        cax.set_xticks(np.arange(Nc))
        nums = [str(i) for i in n]
        cax.set_xticklabels(nums,c=c,fontsize=fontsize2)
        cax.tick_params(axis='both', which='both', length=0, pad=labelpad)
        cax.set_yticks([])

        wa = .07
        ha = .05
        cax.set_aspect(ha/wa)
        cax.set_title('Connections',c=c,fontsize=fontsize2,pad=labelpad)
        for spine in cax.spines.values():
            spine.set_color(None)
    else:

        ax.imshow(images[i],cmap='gray',extent=extent)
        # axins.imshow(images[i][zoomslc],extent=extent,origin='upper')
        axins.imshow(images[i],extent=extent,cmap='gray')
        # axins.imshow(images[i])#,extent=extent)

        if i>0:
            imp = masks[::-1][zoomslc]
            if i==1:
                imp = imp>0
            for (j,k),label in np.ndenumerate(imp):
                axins.text(k+sx+0.5, j+sy+0.45, int(label), ha='center', va='center',␣
↪color=[(label==0)*0.5]*3,fontsize=4)


        axins.set_xlim(zoomslc[1].start, zoomslc[1].stop)
        axins.set_ylim(zoomslc[0].start, zoomslc[0].stop)

        mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec=color+[1],zorder=2)


    if do_labels:
        ax.set_title(labels[i],c=c,fontsize=fontsize,fontweight="bold",pad=5)
    else:
        ax.annotate(string.ascii_lowercase[i], xy=(-0.1, 1), xycoords='axes fraction',
        xytext=(0, 0), textcoords='offset points', va='top', c=axcol,
        fontsize=fontsize)

    ax.axis('off')
    axins.set_xticks([])
    axins.set_yticks([])
    axins.set_facecolor([0]*4)

    for spine in axins.spines.values():
        spine.set_color(color)


```

(continues on next page)

(continued from previous page)

```
# plt.subplots_adjust(wspace=0.15)
fig.patch.set_facecolor([0]*4)

# Display the plot
plt.show()
```



Semantic segmentation sorts pixels into semantic classes. This is often just two classes, or binary classification: foreground and background. In this case, foreground is cell and background is media. As you can see, semantic segmentation does not discern between adjacent instances of foreground objects. We store a semantic segmentation as a binary image file with the same dimensions as the image itself, with foreground pixels labeled `True` (`1`) and background `False` (`0`).

Instance segmentation assigns a unique integer to the pixels each instance of an object - in this case, each cell. This is also conveniently stored as an image file, typically uint8 (unsigned 8-bit integer) for up to $2^8 - 1 = 255$ labels or uint16 (unsigned 16-bit integer) for up to $2^{16} - 1 = 65535$ labels. Signed and/or unsigned 32- or 64-bit formats may also be used, but your OS may not be able to preview these files in its native file manager.

Note: Some instance labels use -1 as an "ignore" label. This can be in conflict with several tasks from indexing to label formatting, which assume unsigned integers, so care must be taken when working with signed formats (int) versus unsigned (uint).

16.2 Bad labels I: Semantic islands to instance labels

Semantic segmentation can be converted into instance segmentation, and this forms the basis of many instance segmentation pipelines. The general steps are:

1. Pre-process image: traditional filtering/blurring/feature extraction or DNN transformation
2. Threshold processed image: adaptive techniques are usually used on the pre-processed image to ensure that the majority of objects pixels are identified despite variations within an image and among images in a dataset. Importantly, object boundaries **must not be identified** as foreground. This allows each object to be associated with a unique island of foreground pixels.
3. These unique blobs are identified using **connected components labeling**. This is the process of building an affinity graph, where pixels are nodes and edges are formed between any adjacent foreground pixels. Adjacency can be defined most narrowly by sharing edges (1-connected in Python, 4-connected in MATLAB) or more broadly by sharing either edges or vertices (2-connected in Python, 8-connected in MATLAB). The graph is then traversed to find all connected components of the graph.

These points are illustrated below. By simulating the amount of foreground pixels detected by filtering+thresholding, we see that is is impossible to distinguish between the two cells until much of the boundary is lost, particularly when using 2-connectivity.

```
import skimage.measure

connectivity = [1,2]
cutoffs = [4,5,6]
row_labels = ['{}-connected'.format(i) for i in connectivity]
col_labels = ['cutoff = {}'.format(i) for i in cutoffs]
# Define the grid dimensions
num_rows = len(row_labels)
num_cols = len(col_labels)

# Create the grid of subplots

# f = .75
# dpi = mpl.rcParams['figure.dpi']
# Y,X = masks.shape[-2:]
# szX = max(X//dpi,2)*f
# szY = max(Y//dpi,2)*f
# fig, axes = plt.subplots(num_rows, num_cols,figsize=(szX*num_cols,szY*num_rows))




h,w = masks.shape[-2:]

sf = w
p = 0.05
h /= sf
w /= sf

# Calculate positions of subplots
N = num_cols
M = num_rows
left = np.array([5*p+i*(w+p) for i in range(N)]*M).flatten().astype(float)
# bottom = np.array([1.5*p]*N + [h+p]*N).flatten().astype(float)
bottom = np.array([h+p]*N+[3*p]*N).flatten().astype(float)
width = np.array([[w]*N]*M).flatten().astype(float)
height = np.array([[h]*N]*M).flatten().astype(float)

max_w = left[-1]+width[-1]
max_h = bottom[-1]+height[-1]

sw = max_w
sh = max_h

sf = max(sw,sh)
left /= sw
bottom /= sh
width /= sw
height /= sh
```

(continues on next page)

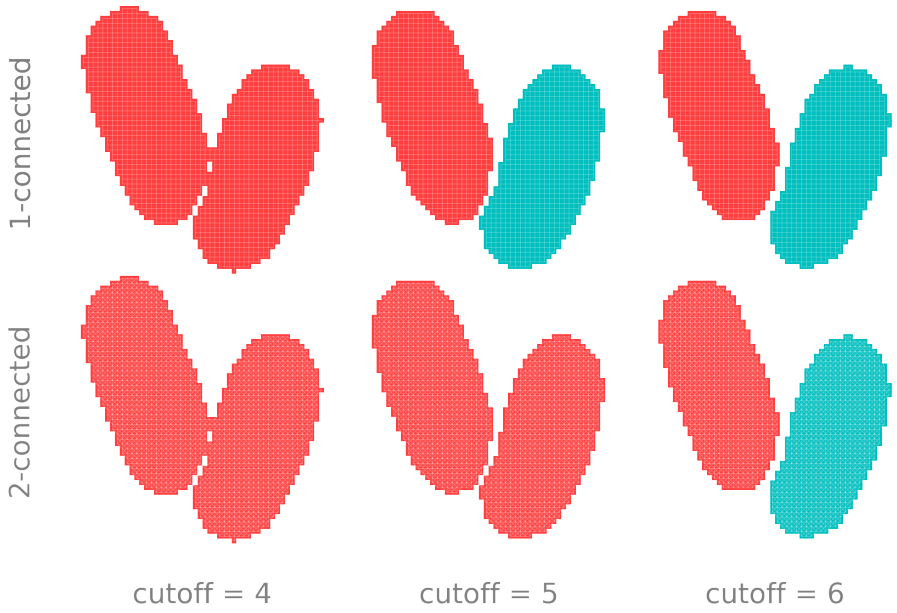
(continued from previous page)

```

# Create figure
s = 5
fig = plt.figure(figsize=(s,s*sh/sw), frameon=False, dpi=300,facecolor=[0]*4)
# fig.patch.set_facecolor([0]*4)

# Add subplots
axes = []
for i in range(N*M):
    ax = fig.add_axes([left[i], bottom[i], width[i], height[i]])
    ax.set_facecolor([0]*4)

    axes.append(ax)


fig.patch.set_facecolor([0]*4)
color = [0.5]*3
# for row,conn in enumerate(connectivity):
#     for col,cutoff in enumerate(cutoffs):

for i,ax in enumerate(axes):
    row,col= np.unravel_index(i,(M,N))

    cutoff = cutoffs[col]
    conn = connectivity[row]


    bin0 = summed_affinity>cutoff
    msk0 = skimage.measure.label(bin0,connectivity=conn)
    pic = apply_ncolor(msk0)


    dim = masks.ndim
    shape = masks.shape
    steps, inds, idx, fact, sign = utils.kernel_setup(dim)
    coords = np.nonzero(msk0)
    affinity_graph = omnipose.core.masks_to_affinity(msk0, coords, steps,
                                                     inds, idx, fact, sign, dim)
    neighbors = utils.get_neighbors(coords,steps,dim,shape)

    #choose to plot cardinal connections only
    step_inds = None if conn==2 else inds[1]

    # index = np.ravel_multi_index([[row],[col]],(N,M))
    # ax = axes[row*col]
    ax.axis('off')

    # ax.text(0,0,i)

    plot_edges(shape,affinity_graph,neighbors,coords,figsize=1,extent=extent,
               fig=fig,ax=ax,step_inds=step_inds,pic=pic,origin='lower',edgecol=[1,1,1,0.
↪5])
```

(continues on next page)

(continued from previous page)

```
    ax.invert_yaxis()

    if i<N:
        ax.annotate(col_labels[i], xy=(0.5, 1), xytext=(0, ax.xaxis.labelpad),
                xycoords=ax.xaxis.label, textcoords='offset points',
                ha='center', va='baseline',color=color,fontsize=fontsize)

    if i%M==0:
        ax.annotate(row_labels[row], xy=(0, 0), xytext=(-ax.yaxis.labelpad + 20, 0),
                    xycoords=ax.yaxis.label, textcoords='offset points',
                    ha='right', va='center', rotation=90,color=color,fontsize=fontsize)

plt.show()
```



Although such sloppy segmentation is good enough for some tasks, we have a better tools now. So in general, do not use image thresholding for segmentation.

16.3 Bad labels II: Watershed lines

While we are on the topic, missing boundary pixels also frequently arise when applying the watershed transform. As usually implemented, this ubiquitous operation returns a semantic classification of an image into watershed lines and catchment basins. As you can tell by the above example, this means that distinct basins must be separated by a 1- or 2-connected watershed line, and therefore boundary pixels are always left unclassified.

There are implementations that allow users to return instance labels *without* the gaps let by watershed lines (*e.g.*, `skimage.segmentation.watershed`), but I have yet to see a paper published using this method. Despite this fix, watershed also tends to over-segment images (even when transformed by traditional filters or DNNs). So in general, do not use watershed for instance segmentation.

16.4 Bad labels III: Self-contact boundaries

Instance labels are good enough to fully describe a lot of objects. More precisely, there is a bijective map between the affinity graph and the instance label matrix whenever all edge pixels are in contact with a non-self pixel. This assumption fails in many interesting (and biologically relevant) scenarios, including bacterial microscopy. Consider the following image containing one extremely filamentous cell and corresponding cell mask:

```
# read in files; this is an entire movie, but we will just be looking at the last frame
import tifffile

nm = 'long_10_2'
masks = tifffile.imread(os.path.join(basedir,nm+'_op_masks.tif'))
phase = tifffile.imread(os.path.join(basedir,nm+'_phase.tif'))
fluor = tifffile.imread(os.path.join(basedir,nm+'_fluor.tif'))
afnty = utils.load_nested_list(os.path.join(basedir,nm+'_affinity.npz'))

# make figure
import omnipose, cellpose_omni
im = phase[-1]
msk = masks[-1]

f = 1
c = [0.5]*3
fontsize=11

titles = [r'$\bf{image}$'+'\n(phase contrast)', r'$\bf{label}$'+'\n(single cell mask)', r
↪'$\bf{boundary}$'+'\n(from cell mask)']
ol = cellpose_omni.utils.masks_to_outlines(msk,omni=True)
# outlines = np.stack([ol]*4,axis=-1)*0.5
images = [im,
          omnipose.plot.apply_ncolor(msk),
          omnipose.plot.apply_ncolor(ol,offset=.5)]



# Set up the figure and subplots
N = len(images)
h,w = im.shape

sf = h
p = 0.5 # needs to be defined as fraction of width for aspect ratio to work?


h,w = im.shape
extent = np.array([0,w,0,h])#-0.5
sy,sx,wy,wx = [h//2.5,w//3.6,40,40]
zoomslc = tuple([slice(sy,sy+wy),slice(sx,sx+wx)])
zoom = 5
bbox_to_anchor = (-(wx/w)*zoom/(1.25),-zoom/1.5*wy/h) # inset axis


asp = h/w

```

(continues on next page)

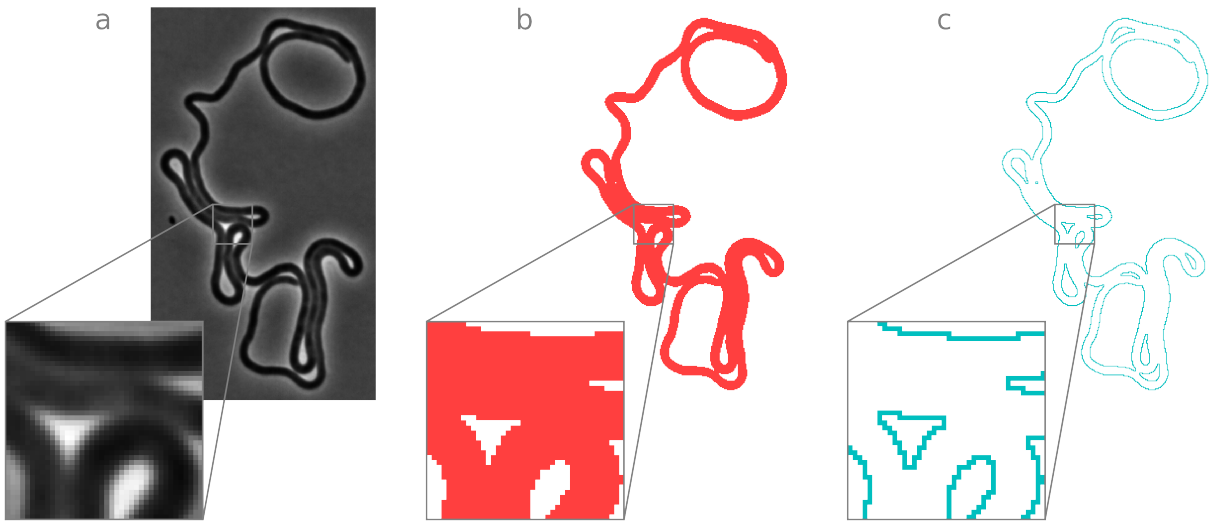
(continued from previous page)

```
h /= sf
w /= sf

oy,ox = np.abs(bbox_to_anchor)/2
# oy,ox = 0.,0.
# Calculate positions of subplots
left = np.array([ox+i*(w+p) for i in range(N)])*1.
bottom = np.array([oy]*N)*1.
width = np.array([w]*N)*1.
height = np.array([h]*N)*1.

max_w = left[-1]+width[-1]+ox
max_h = bottom[-1]+height[-1]+oy

sw = max_w
sh = max_h

sf = max(sw,sh)
left /= sw
bottom /= sh
width /= sw
height /= sh

# Create figure
s = 6.5
fig = plt.figure(figsize=(s,s*sh/sw), frameon=False, dpi=600)
# fig.patch.set_facecolor([0]*4)

# Add subplots
axes = []
for i in range(N):
    ax = fig.add_axes([left[i], bottom[i], width[i], height[i]])
    axes.append(ax)



lwa = 2/N # linewidth for axes
lw = lwa/20 # linewidth for affinity graph
labelpad = 2

for i, ax in enumerate(axes):

    ax.imshow(images[i],cmap='gray',extent=extent)
    ax.axis('off')



    # inset axes....
    # axins = ax.inset_axes([0.5, 0.5, 0.47, 0.47])
    # axins = inset_axes(ax, 1,1 , loc=2, bbox_to_anchor=(.08, 0.35))
    axins = zoomed_inset_axes(ax, zoom, loc='lower left',
                              # bbox_to_anchor=(1.1, 1.1),
```

(continues on next page)

(continued from previous page)

```
                                # bbox_to_anchor=(-.15,-.15),
                                bbox_to_anchor=bbox_to_anchor,
                                bbox_transform=ax.transAxes)



    # axins.imshow(images[i][zoomslc],extent=extent,origin='upper')
    axins.imshow(images[i],extent=extent,cmap='gray')
    # axins.imshow(images[i])#,extent=extent)
    axins.set_xlim(zoomslc[1].start, zoomslc[1].stop)
    axins.set_ylim(zoomslc[0].start, zoomslc[0].stop)

    mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec=color+[1], zorder=2, lw=lwa)
    # axins.axis('off')
    axins.set_xticks([])
    axins.set_yticks([])
    axins.set_facecolor([0]*4)


    for spine in axins.spines.values():
        spine.set_color(color)
        spine.set_linewidth(lwa)


    if do_labels:
        # ax.set_title(labels[i],c=c,fontsize=fontsize,fontweight="bold",pad=5)
        ax.set_title(titles[i],c=c,fontsize=fontsize,pad=5)

    else:
        ax.annotate(string.ascii_lowercase[i], xy=(-0.25, 1), xycoords='axes fraction',
        xytext=(0, 0), textcoords='offset points', va='top', c=axcol,
        fontsize=fontsize)



# plt.subplots_adjust(wspace=0,hspace=0)

# Display the plot
plt.show()
```



Because the label for this cell is the same integer on either side of a self-contact interface, we cannot localize the boundary of the cell at these interfaces. However, affinity segmentation encodes not only the information necessary to recontruct cell boundaries but also to traverse cell boundarues as a parametric contour.

```
import omnipose, cellpose_omni
from scipy import signal
t = -1 # last frame
im = phase[t]
msk = masks[t]

f = 1
c = [0.5]*3
fontsize = 11

titles = [#r'$\bf{image}$'+'\n(phase contrast)',
          r'$\bf{connectivity}$'+'\n(affinity graph)',
          r'$\bf{boundary}$'+'\n(from affinity graph)',
          r'$\bf{contour}$'+'\n(traced with affinity)']

# extract the addinity graph and coordinate array
aa = afnty[t]
shape = msk.shape
dim = msk.ndim
neighbors = aa[:dim]
affinity_graph = aa[dim]#.astype(bool) #VERY important to cast to bool, now done␣
↪internally
idx = affinity_graph.shape[0]//2
coords = tuple(neighbors[:,idx])

# make the boundary
ol = omnipose.core.affinity_to_boundary(msk,affinity_graph,coords)

# make the contour
contour_map, contour_list, unique_L = omnipose.core.get_contour(msk,affinity_graph,
↪coords,cardinal_only=1)
```

(continues on next page)

(continued from previous page)

```
cmap='inferno'
color = [0.5]*3



# contour_colored = np.stack([(contour_map>1).astype(np.float32)]*4,axis=-1)
contour_colored = np.zeros(contour_map.shape+(4,))

for contour in contour_list:
    # coords_t = np.unravel_index(contour,contour_map.shape)
    coords_t = np.stack([c[contour] for c in coords])
    cyclic_diff = np.diff(np.append(coords_t, coords_t[:, 0:1], axis=1), axis=1)

    a = cyclic_diff
    window_size = 11
    window = np.ones(window_size) / window_size
    cyclic_diff = signal.convolve2d(np.concatenate((a[:, -window_size+1:], a, a[:,␣
↪:window_size-1]), axis=1), np.expand_dims(window, axis=0), mode='same')[:, window_size-
↪1:-window_size+1]

    angles = np.arctan2(cyclic_diff[1], cyclic_diff[0])+np.pi


    a = 2
    r = ((np.cos(angles)+1)/a)
    g = ((np.cos(angles+2*np.pi/3)+1)/a)
    b =((np.cos(angles+4*np.pi/3)+1)/a)

    rgb = np.stack((r,g,b,np.ones_like(angles)),axis=-1)

    # v = np.array(range(len(contour)))/len(contour)
    # contour_colored[tuple(coords_t)] = ctr_cmap(v)
    contour_colored[tuple(coords_t)] = rgb


images = [#im,
          None,
          omnipose.plot.apply_ncolor(ol,offset=.5),
          contour_colored]

# N = len(images)
# A = N//2
# B = N-A

# fig, axes = plt.subplots(2,B, figsize=(szX*A,szY*B))
# fig.patch.set_facecolor([0]*4)

# inset axis


# Set up the figure and subplots
```

(continues on next page)

(continued from previous page)

```
N = len(images)
h,w = im.shape


extent = np.array([0,w,0,h])#-0.5
sy,sx,wy,wx = [h//2.5,w//3.6,40,40]
zoomslc = tuple([slice(sy,sy+wy),slice(sx,sx+wx)])
zoom = 5
bbox_to_anchor = (-(wx/w)*zoom/(1.25),-zoom/1.5*wy/h)
asp = h/w

sf = h
p = 0.5 # needs to be defined as fraction of width for aspect ratio to work?
h /= sf
w /= sf

oy,ox = np.abs(bbox_to_anchor)/2
# oy,ox = 0.,0.
# Calculate positions of subplots
left = np.array([ox+i*(w+p) for i in range(N)])*1.
bottom = np.array([oy]*N)*1.
width = np.array([w]*N)*1.
height = np.array([h]*N)*1.

max_w = left[-1]+width[-1]+ox
max_h = bottom[-1]+height[-1]+oy

sw = max_w
sh = max_h

sf = max(sw,sh)
left /= sw
bottom /= sh
width /= sw
height /= sh

# Create figure
s = 6.5
fig = plt.figure(figsize=(s,s*sh/sw), frameon=False, dpi=600)

# Add subplots
axes = []
for i in range(N):
    ax = fig.add_axes([left[i], bottom[i], width[i], height[i]])
    axes.append(ax)



h,w = im.shape


lwa = 2/N # linewidth for axes
```

(continues on next page)

(continued from previous page)

```
lw = lwa/20 # linewidth for affinity graph
labelpad = 4/3

fontsize2 = 8

for i, ax in enumerate(axes):

    # inset axes....
    # axins = ax.inset_axes([0.5, 0.5, 0.47, 0.47])
    # axins = inset_axes(ax, 1,1 , loc=2, bbox_to_anchor=(.08, 0.35))
    axins = zoomed_inset_axes(ax, zoom, loc='lower left',
                              # bbox_to_anchor=(1.1, 1.1),
                              # bbox_to_anchor=(-.15,-.15),
                              bbox_to_anchor=bbox_to_anchor,
                              bbox_transform=ax.transAxes)

    if i==(N-3):
        # ax.invert_yaxis()
        neighbors = utils.get_neighbors(coords,steps,dim,shape)

        # plot the while affinity graph
        summed_affinity, affinity_cmap = plot_edges(shape,affinity_graph,neighbors,
↪coords,
                                                    figsize=1,fig=fig,ax=ax,
↪extent=extent,
                                                    edgecol=edgecol,cmap=cmap,
↪linewidth=lw

                                                   )

        # plot the inset one
        axins.invert_yaxis()
        ax.invert_yaxis()

        summed_affinity, affinity_cmap = plot_edges(shape,affinity_graph,neighbors,
↪coords,
                                                  figsize=1,fig=fig,ax=axins,
                                                  extent=extent,
                                                  edgecol=edgecol,linewidth=lw*zoom,cmap=cmap
                                                 )

        # axins.set_xlim(zoomslc[1].start, zoomslc[1].stop)
        axins.set_xlim(zoomslc[1].start, zoomslc[1].stop)

        # axins.set_ylim(zoomslc[0].stop, zoomslc[0].start)
        # axins.set_ylim(h-zoomslc[0].stop, h-zoomslc[0].start)
        axins.set_ylim(h-zoomslc[0].start, h-zoomslc[0].stop)


        loc1,loc2 = 4,2
        patch, pp1, pp2 = mark_inset(ax, axins, loc1=loc1, loc2=loc2, fc="none",
                                     ec=color+[1],zorder=2,lw=lwa)
```

(continues on next page)

(continued from previous page)

```
        pp1.loc1 = 4
        pp1.loc2 = 1
        pp2.loc1 = 2
        pp2.loc2 = 3




        # Display the color swatches as an image
        Nc = affinity_cmap.N
        colors = affinity_cmap.colors

        wa = .07
        ha = .05
        # cax = fig.add_axes([.3975, -.08, wa, ha])
        cax = inset_axes(ax, width="60%", height="100%", loc='lower right',
                 bbox_to_anchor=(0, -0.725, 1, 1), bbox_transform=ax.transAxes,
                 borderpad=0)

        n = np.arange(3,9)
        Nc = len(n)
        cax.imshow(affinity_cmap(n.reshape(1,Nc)))#,vmin=n[0]-1,vmax=n[-1]+1)

        # Set the y ticks and tick labels
        # cax.set_yticks(np.arange(N))
        cax.set_xticks(np.arange(Nc))

        nums = [str(i) for i in n]
        cax.set_xticklabels(nums,c=c,fontsize=fontsize2)
        cax.tick_params(axis='both', which='both', length=0, pad=labelpad)
        cax.set_yticks([])

        wa = .07
        ha = .05
        cax.set_aspect(ha/wa)
        cax.set_title('Connections',c=c,fontsize=fontsize2, pad=labelpad)
        for spine in cax.spines.values():
            spine.set_color(None)

        # cax.xaxis.set_labelpad  = -10
    else:




        ax.imshow(images[i],cmap='gray',extent=extent)
        # axins.imshow(images[i][zoomslc],extent=extent,origin='upper')
        axins.imshow(images[i],extent=extent,cmap='gray')
        # axins.imshow(images[i])#,extent=extent)
        axins.set_xlim(zoomslc[1].start, zoomslc[1].stop)
        axins.set_ylim(zoomslc[0].start, zoomslc[0].stop)

        mark_inset(ax, axins, loc1=2, loc2=4, fc="none",
```

(continues on next page)

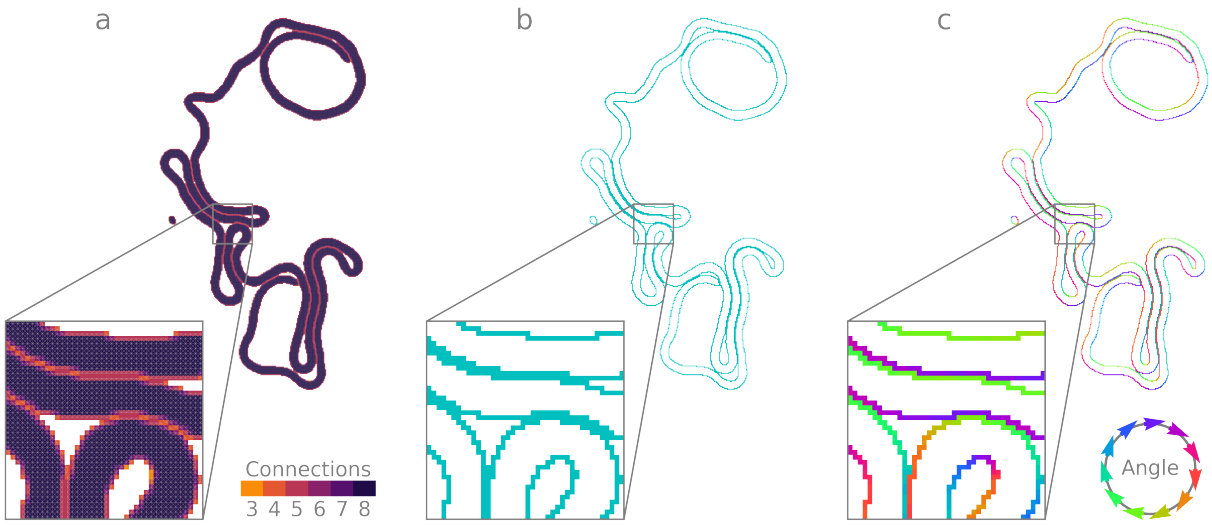
(continued from previous page)

```
                    ec=color+[1], zorder=2, lw=lwa)



    if i==(N-1):
        # ax2 = fig.add_axes([.7, -.15, .25, .25])
        ax2 = inset_axes(ax, width="60%", height="100%", loc='lower right',
             bbox_to_anchor=(0, -0.675, 1, 1), bbox_transform=ax.transAxes,
             borderpad=0)
        lw2 = 1

        # Create the circle and arrows on the second subplot
        circle = plt.Circle((0, 0), 1, fill=False, edgecolor=c,lw=lw2)
        ax2.add_artist(circle)

        # Set the number of arrows and colormap
        n_arrows = 11
        cmap = plt.get_cmap('hsv')

        for j in range(n_arrows):
            angle = j * 2 * np.pi / n_arrows
            a = 2
            r = ((np.cos(angle)+1)/a)
            g = ((np.cos(angle+2*np.pi/3)+1)/a)
            b =((np.cos(angle+4*np.pi/3)+1)/a)

            rgb = np.stack((r,g,b,np.ones_like(angle)),axis=-1)


            x, y = np.cos(angle), np.sin(angle)
            # dx, dy = -y, x
            dx, dy = y,-x

            # clr = cmap(j / n_arrows)
            ax2.quiver(x, y, dx, dy, color=rgb, angles='xy', scale_units='xy', scale=2,␣
↪width=.05)

        # Add text to the center of the circle
        ax2.text(0, 0, 'Angle', ha='center', va='center',c=c,fontsize=fontsize2)

        # Set the axis limits and aspect ratio
        ax2.set_xlim(-1.5, 1.5)
        ax2.set_ylim(-1.5, 1.5)
        ax2.set_aspect('equal')

        # Remove the axes from the second subplot
        ax2.axis('off')


    if do_labels:
        # ax.set_title(labels[i],c=c,fontsize=fontsize,fontweight="bold",pad=5)
        ax.set_title(titles[i],c=c,fontsize=fontsize,pad=5)
```

(continues on next page)

(continued from previous page)

```
    else:
        ax.annotate(string.ascii_lowercase[i], xy=(-0.25, 1), xycoords='axes fraction',
        xytext=(0, 0), textcoords='offset points', va='top', c=axcol,
        fontsize=fontsize)



    ax.axis('off')
    # axins.axis('off')
    axins.set_xticks([])
    axins.set_yticks([])
    axins.set_facecolor([0]*4)


    for spine in axins.spines.values():
        spine.set_color(color)
        spine.set_linewidth(lwa)


# plt.subplots_adjust(wspace=-0,hspace=0)
plt.subplots_adjust(wspace=-.35,hspace=.5)


# Display the plot
plt.show()
```



Pixels (or in ND, hypervoxels) may be classified as interior or boundary by their net connectivity. An ND hypervoxel connected to all $3^N - 1$ neighbors is classified as **internal** (8 in 2D, fully 2-connected to both cardinal and ordinal neighbors). Hypervoxels with fewer than $3^N - 1$ connections are classified as **boundary**. In Omnipose, hypervoxels with fewer than N connections are pruned when using affinity segmentation to avoid spurs and allow cell contours in 2D to be traced.

Because connections in an affinity graph are symmetrical, interfaces between objects are 2 hypervoxels thick. That is, the shortest path between the interiors of any two objects will pass through at least two boundary hypervoxels, one

belonging to each object. Thresholding-based methods of boundary detection do not guarantee this symmetry and thus predict too many or too few boundary hypervoxels.

CHAPTER

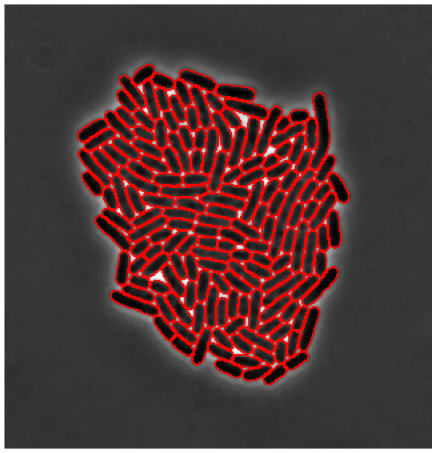
SEVENTEEN

N-COLOR

Here I will argue that many of the errors I see in ground-truth datasets can be most kindly attributed to a lack of good label visualization. To illustrate, I will use the following cell microcolony.

17.1 The insufficiency of cell outlines

```
import matplotlib.pyplot as plt
plt.style.use('dark_background')
import matplotlib as mpl
%matplotlib inline
mpl.rcParams['figure.dpi'] = 600
import numpy as np
import omnipose
from omnipose.utils import rescale, crop_bbox
from omnipose.plot import imshow
import fastremap

from pathlib import Path
import os
from cellpose_omni import io, plot
omnidir = Path(omnipose.__file__).parent.parent
basedir = os.path.join(omnidir,'docs','test_files') #first run the mono_channel_bact␣
↪notebook to generate masks
masks = io.imread(os.path.join(basedir,'masks','ec_5I_t141xy5c1_cp_masks.tif'))
img = io.imread(os.path.join(basedir,'ec_5I_t141xy5c1.tif'))
imshow(plot.outline_view(img,masks),3,interpolation='None')
```



This outline view clearly distinguishes cells from each other, and it requires just one color (one channel). As ground truth, binary maps like this are one of the easiest annotations to generate and are therefore quite common in public datasets (see MiSiC, DeLTA, and SuperSegger just for a few in the realm of bacterial microscopy).

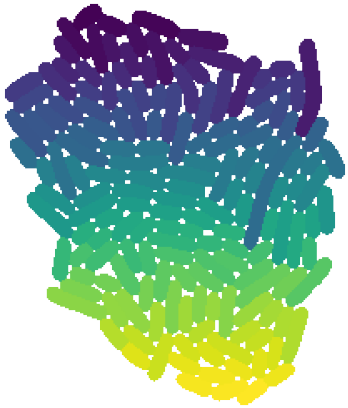
Despite the ease of drawing reasonable cell outlines, it is exceptionally difficult to guarantee that these monochromatic boundaries between cells are **precisely** 2 pixels thick. Without this property, the resulting label matrix will either exclude boundary pixels or asymmetrically incorporate them into one of the two cells. This is a primary reason why label matrices, not boundary maps, should be used to train and evaluate any segmentation algorithm (labels can fail in self-contact scenarios, but Omnipose now accepts affinity graphs or linked label matrices just for those cases).

17.2 Not enough colors to go around

However, creating and editing label matrices has its own set of issues. If you have too many cells in an image, you quickly run out of distinct colors to distinguish adjacent cells:

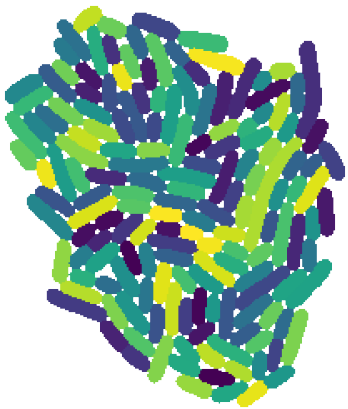
```
bbx = crop_bbox(masks) #in omni
slc = bbx[0]
m,_ = fastremap.renumber(masks[slc]) # make sure masks go from 0 to N
print('number of masks: ', np.max(m))

cmap = mpl.colormaps.get_cmap('viridis')
pic1 = cmap(rescale(m))
pic1[:,:,-1] = m>0 # alpha
imshow(pic1,3,interpolation='None')
```

```
number of masks:  161
```



This perceptually uniform color map is our best bet of distinguishing cells from each other, but some close cells are too similar to tell apart. The standard technique is to randomly shuffle the labels:

```
import fastremap
keys = fastremap.unique(m)
vals = keys.copy()
np.random.seed(42)
np.random.shuffle(keys)
d = dict(zip(keys,vals))
m_shuffle = fastremap.remap(m,d)
pic2 = cmap(rescale(m_shuffle))
pic2[:,:,-1] = m>0 # alpha
imshow(pic2,3,interpolation='None')
```



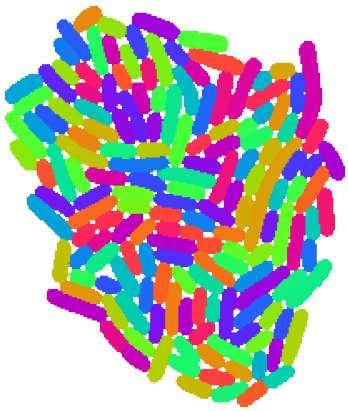
This doesn't fix the problem. You might think that adding more colors would help...

```
from omnipose.utils import sinebow
from matplotlib.colors import ListedColormap

cmap = ListedColormap([color for color in list(sinebow(m.max()).values())[1:]])
pic3 = cmap(m_shuffle)
pic3[:,:,-1] = m>0 # alpha
```

(continues on next page)

(continued from previous page)

```
imshow(pic3,3,interpolation='None')
```



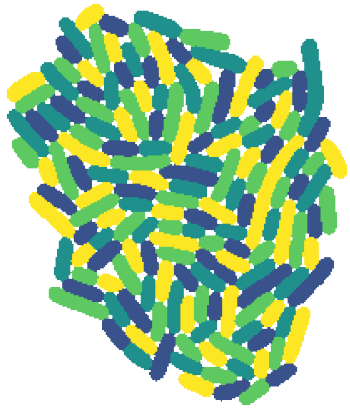
... but since even random shuffling *does not guarantee* that numerically close labels become spatially separated, adjacent labels that were hard to tell apart using a perceptually uniform color map like viridis are often *more difficult* to tell apart using any kind of unicorn-vomit color map.

Worse still, multiple similar colors can accidentally get used while editing the *wrong cell* (*e.g.*, color 11 inside cell 12 that are both shades of yellow) and ruin the segmentation despite this error being imperceptible to the human eye (this may account for many of the "errant pixels" we observe across ground-truth datasets of dense cells).

17.3 4-color in theory, N-color in practice

To solve this problem, I developed the `ncolor` package, which converts K-integer label matrices to $N \ll K$ - color labels. The four color theorem guarantees that you only need 4 unique cell labels to cover all cells, but my algorithm opts to use 5 if a solution using 4 is not found quickly. This was integral in developing the BPCIS dataset, and I subsequently incorporated it into Cellpose and Omnipose. By default, the GUI and plot commands display N-color masks for easier visualization and editing:

```
import ncolor
cmap = mpl.colormaps.get_cmap('viridis')
pic4 = cmap(rescale(ncolor.label(m)))
pic4[:,:,-1] = m>0 # alpha
imshow(pic4,3,interpolation='None')
```



Interesting note: my code works for 3D volume labels as well, but there is no analogous theorem guaranteeing any sort of upper bound $N < K$ in 3D. In 3D, you could in principle have cells that touch every other cell, in which case $N = K$ and you cannot "recolor your map". On the dense but otherwise well-behaved volumes I have tested, my algorithm ends up needing 6-7 unique labels. I am curious if some bound on N can be formulated in the context of constrained volumes, *e.g.*, packed spheres of mixed and arbitrary diameter...

Getting uniform colors for non-contacting or sparse objects

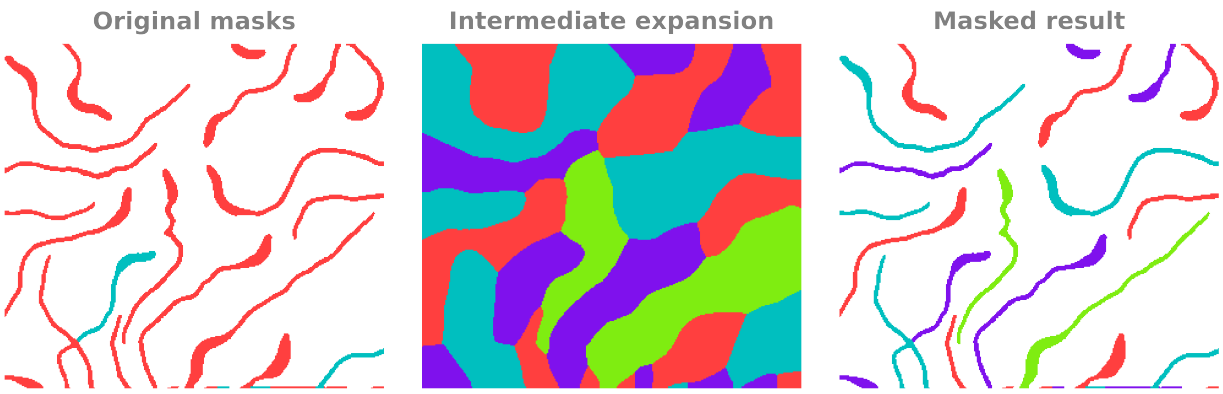
Final note: thanks to Ryan Peters for suggesting a fix for displaying segmentations that (a) are from ground-truth sets with pixel-separated (boundary-map-generated) label matrices or (b) have many sparse, disjoint objects. By expanding labels before coloring them (a step that actually takes far longer than the coloring step itself), we get a much more pleasing distribution of colors that can make it easier to assess segmentations when when images are zoomed out. For example,

```
from omnipose import plot
masks = io.imread(os.path.join(basedir,'masks','caulo_15_cp_masks.tif'))
exp = ncolor.expand_labels(masks)
ims = [plot.apply_ncolor(masks,expand=False),
       plot.apply_ncolor(exp),
       plot.apply_ncolor(masks)]

titles = ['Original masks','Intermediate expansion', 'Masked result']
N = len(titles)
f = 1.5
c = [0.5]*3
fontsize=10
dpi = mpl.rcParams['figure.dpi']
Y,X = masks.shape[-2:]
szX = max(X//dpi,2)*f
szY = max(Y//dpi,2)*f

fig, axes = plt.subplots(1,N, figsize=(szX*N,szY))
fig.patch.set_facecolor([0]*4)
for i,ax in enumerate(axes):
    ax.imshow(ims[i])
    ax.axis('off')
    ax.set_title(titles[i],c=c,fontsize=fontsize,fontweight="bold")

plt.subplots_adjust(wspace=0.1)
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



Left: ncolor applied to raw masks. Middle: ncolor expanded masks. Right: resulting ncolor masks with more uniform color distribution.

Note that the expansion step takes about 2x longer than the ncolor algorithm itself takes to run, but the extra milliseconds are worth it. If you know of any faster way to get a feature transform than `scipy.ndimage`, please let me know.

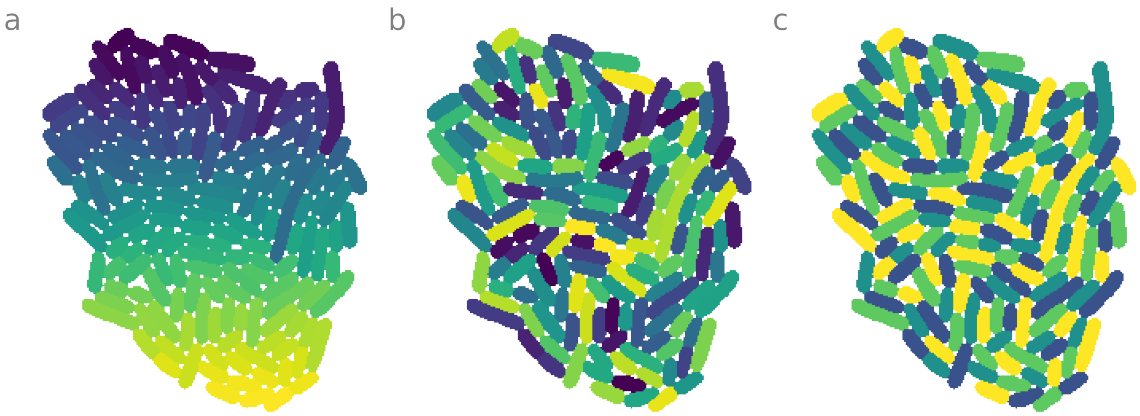
```
import string
fontsize = 11

axcol = [0.5]*3
# Set up the figure and subplots
images = [pic1,pic2,pic4]
N = len(images)
M = 1

h,w = images[0].shape[:2]

sf = w
p = 0.05 # needs to be defined as fraction of width for aspect ratio to work?
h /= sf
w /= sf
offset = 0.05
# Calculate positions of subplots
left = np.array([i*(w+p) for i in range(N)])*1.+offset
bottom = np.array([0]*N)*1.
width = np.array([w]*N)*1.
height = np.array([h]*N)*1.

max_w = left[-1]+width[-1]
max_h = bottom[-1]+height[-1]

sw = max_w
sh = max_h

sf = max(sw,sh)
left /= sw
bottom /= sh
```

(continues on next page)

(continued from previous page)

```
width /= sw
height /= sh

# Create figure
s = 6
fig = plt.figure(figsize=(s,s*sh/sw), frameon=False, dpi=600)
# fig = plt.figure(figsize=(s,s*sh/sw), frameon=False, dpi=300,constrained_layout=True)
# fig.set_constrained_layout_pads(w_pad=-0.25, h_pad=0., hspace=0., wspace=0.25)
# fig.patch.set_facecolor([0]*4)

# Add subplots
axes = []
for i in range(N):
    ax = fig.add_axes([left[i], bottom[i], width[i], height[i]])
    ax.imshow(images[i])
    axes.append(ax)

    ax.annotate(string.ascii_lowercase[i], xy=(-offset, 1), xycoords='axes fraction',
    xytext=(0, 0), textcoords='offset points', va='top', c=axcol,
    fontsize=fontsize)

    ax.axis('off')

datadir = omnidir.parent
file = os.path.join(datadir,'Dissertation','figures','ncolor.pdf')
if os.path.isfile(file): os.remove(file)
fig.savefig(file,transparent=True,pad_inches=0)#,bbox_inches='tight')

m.max(),ncolor.label(m).max()
```

```
(161, 4)
```



CHAPTER

EIGHTEEN

CELL DIAMETER

The idea of an average cell diameter sounds intuitive, but the standard implementation of this idea fails to capture that intuition. The go-to method (adopted in Cellpose) is to calculate the cell diameter as the diameter of the circle of equivalent area. As I will demonstrate, this fails for anisotropic (non-circular) cells. As an alternative, I devised the following simple diameter metric:

```
diameter = 2*(dimension+1)*np.mean(distance_field)
```

Because the distance field represents the distance to the *closest* boundary point, it naturally captures the intrinsic 'thickness' of a region (in any dimension). Averaging the field over the region (the first moment of the distribution) distills this information into a number that is intuitively proportional to the thickness of the region. For example, if a region is made up of a bungle of many thin fragments, its mean distance is far smaller than the mean distance of the circle of equivalent area. But to call it a 'diameter', I wanted this metric to match the diameter of a sphere in any dimension. So, by calculating the average of distance field of an n-sphere, we get the above expression for the the diameter of an n-sphere given the average of the distance field over the volume.

18.1 Example cells

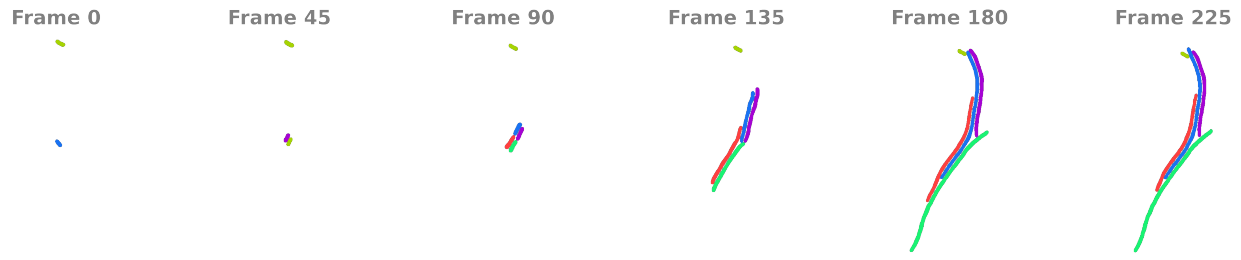
Filamenting bacterial cells often exhibit constant width but increasing length. This dataset comes from the deletion of the essential gene *ftsN* in *Acinetobacter baylyi*.

```
from pathlib import Path
from cellpose_omni import utils, plot, models, io, dynamics
import os, sys, io
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('dark_background')
import matplotlib as mpl
%matplotlib inline
mpl.rcParams['figure.dpi'] = 600

# Save a reference to the original stdout stream
old_stdout = sys.stdout

# Redirect stdout to a StringIO object
sys.stdout = io.StringIO()


import omnipose
from omnipose.plot import imshow
```

(continues on next page)

(continued from previous page)

```
import tifffile
omnidir = Path(omnipose.__file__).parent.parent
basedir = os.path.join(omnidir,'docs','_static')
nm = 'ftsZ'
masks = tifffile.imread(os.path.join(basedir,nm+'_masks.tif'))
mnc = omnipose.plot.apply_ncolor(masks)

f = 1
c = [0.5]*3
fontsize=10
dpi = mpl.rcParams['figure.dpi']
Y,X = masks.shape[-2:]
szX = max(X//dpi,2)*f
szY = max(Y//dpi,2)*f

# T = [50,80,100,150,180,240]
T = range(0,len(masks),45)
titles = ['Frame {}'.format(t) for t in T]
ims = [mnc[t] for t in T]
N = len(titles)

fig, axes = plt.subplots(1,N, figsize=(szX*N,szY))
fig.patch.set_facecolor([0]*4)

for i,ax in enumerate(axes):
    ax.imshow(ims[i])
    ax.axis('off')
    ax.set_title(titles[i],c=c,fontsize=fontsize,fontweight="bold")

plt.subplots_adjust(wspace=0.1)
plt.show()

# Restore the original stdout stream
sys.stdout = old_stdout
```



18.2 Compare diameter metrics

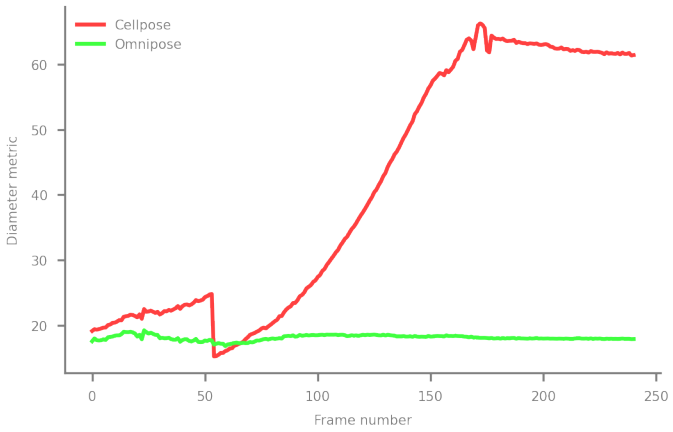
By plotting the mean diameter (averaged over all cells after being computed per-cell, of course), we find that the 'circle diameter metric' used in Cellpose rises drastically with cell length, but the 'distance diameter metric' of Omnipose remains nearly constant. If we tried to use the former to train a `SizeModel()`, images would get downsampled heavily to the point of cells being **too thin to segment**, and that is assuming that the model can reliably detect the highly nonlocal property of cell length in an image instead of the local property of cell width (at least, what we humans would point to and *call* cell width).

```
import fastremap
n = len(masks)
diam_old = []
diam_new = []
cell_num = []
x = range(n)
for k in x:
    m = masks[k]
    fastremap.renumber(m,in_place=True)
    cell_num.append(m.max())
    diam_old.append(utils.diameters(m,omni=False)[0])
    diam_new.append(utils.diameters(m,omni=True)[0])


from omnipose.utils import sinebow
golden = (1 + 5 ** 0.5) / 2
sz = 4
labelsize = 5

%matplotlib inline

plt.style.use('dark_background')
mpl.rcParams['figure.dpi'] = 300

axcol = [0.5]*3+[1]
N = 3
colors = sinebow(N,offset=0)
background_color = [0]*4

fig = plt.figure(figsize=(sz, sz/golden),frameon=False)
fig.patch.set_facecolor(None)

ax = plt.axes()

ax.plot(range(n),diam_old,c=colors[1],label='Cellpose')
ax.plot(range(n),diam_new,c=colors[N],label='Omnipose')

ax.legend(loc='best', frameon=False,labelcolor=axcol, fontsize = labelsize)
ax.tick_params(axis='both', which='major', labelsize=labelsize,length=3, direction="out",
↪colors=axcol,bottom=True,left=True)
ax.tick_params(axis='both', which='minor', labelsize=labelsize,length=3, direction="out",
↪colors=axcol,bottom=True,left=True)
ax.set_ylabel('Diameter metric', fontsize = labelsize,c=axcol)
ax.set_xlabel('Frame number', fontsize = labelsize, c=axcol)
```

(continues on next page)

(continued from previous page)

```
ax.set_facecolor(background_color)

for spine in ax.spines.values():
    spine.set_color(axcol)

ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)

plt.show()
```



CHAPTER

NINETEEN

GAMMA

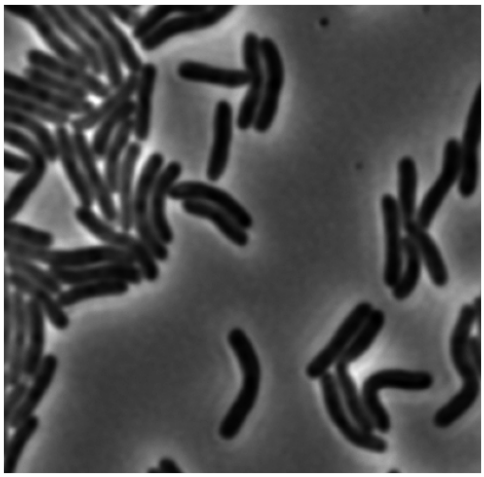
One of the more trivial uses of good binary segmentation (let alone best-in-class *instance* segmentation) is the ability to adjust an image based on foreground/background values.

19.1 Example Image

To start off, consider this example image:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
plt.style.use('dark_background')
dpi = 600
mpl.rcParams['figure.dpi'] = dpi
px = 1/plt.rcParams['figure.dpi']  # pixel in inches
import matplotlib_inline
matplotlib_inline.backend_inline.set_matplotlib_formats('png')

%matplotlib inline

import numpy as np
import omnipose
from omnipose.plot import imshow
from pathlib import Path
import os
from cellpose_omni import io, plot
omnidir = Path(omnipose.__file__).parent.parent
basedir = os.path.join(omnidir,'docs','test_files')
im = io.imread(os.path.join(basedir,'e1t1_crop.tif'))

imshow(im,1,cmap='gray')
```

```
2024-03-04 13:10:51,353        [INFO ]        [io.py      61   logger_setup     ␣
↪]        WRITING LOG OUTPUT TO /home/kcutler/.cellpose/run.log
```



This image is 16-bit and already adjusted to span the entire bit depth:

```
print(im.dtype, im.ptp()==(2**16-1))
```

```
uint16 True
```

19.2 Exposure and outliers

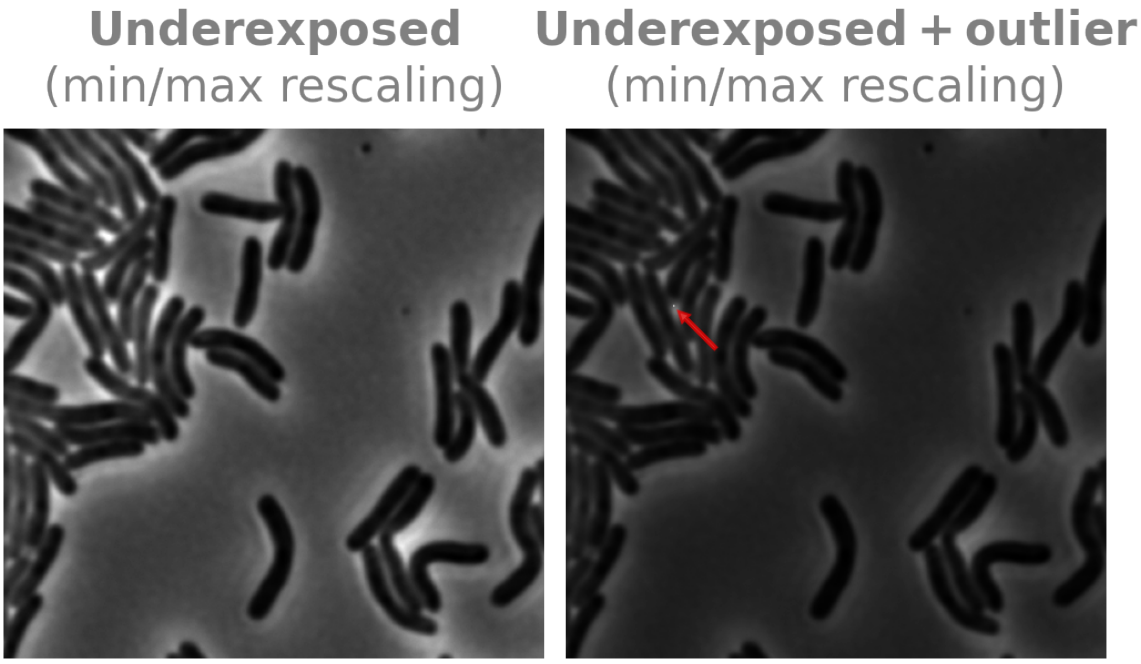
Raw data is often under- or over-exposed and can contain outliers where pixels are saturated. We can simulate this by dividing the image by 2 and adding a bright pixel:

```
im_bad = im * .5 # reduce brightness by 50%

f = 1
c = [0.5]*3
fontsize=10

# Number of subplots in the right column
n = 2
h, w = im_bad.shape[:2]

sf = w
p = 0.0001*w # needs to be defined as fraction of width for aspect ratio to work?
h /= sf
w /= sf

# Calculate positions of subplots
left = np.array([i*(w+p) for i in range(n)])*1.
bottom = np.array([0]*n)*1.
width = np.array([w]*n)*1.
height = np.array([h]*n)*1.

```

(continues on next page)

(continued from previous page)

```
max_w = left[-1]+width[-1]
max_h = bottom[-1]+height[-1]

sf = max_w - (n-1)*p
left /= sf
bottom /= sf
width /= sf
height /= sf


s = 6.5 * 2/4 # make it so that these appear the same size
fig = plt.figure(figsize=(s,s), frameon=False, dpi=300)

ax = fig.add_axes([left[0], bottom[0], width[0], height[0]])
ax.imshow(im_bad,cmap='gray')
ax.axis('off')
ax.set_title(r'$\bf{Underexposed}$' + '\n(min/max rescaling)',c=c,fontsize=fontsize)

y,x = im.shape[0]//3,im.shape[1]//5
im_bad[y,x] = im_bad.max()*2 # add a bright pixel
im_bad = omnipose.utils.rescale(im_bad)

ax = fig.add_axes([left[1], bottom[1], width[1], height[1]])
ax.imshow(im_bad,cmap='gray')
ax.axis('off')
ax.set_title(r'$\bf{Underexposed+outlier}$' + '\n(min/max rescaling)',c=c,
↪fontsize=fontsize)

scale = 50
arrow_length = 0.1*scale
dx=dy=-5
offx=offy=-5
ax.arrow(x - dx*arrow_length-offx, y - dy*arrow_length-offy, dx*arrow_length, dy*arrow_
↪length,
         width=0.01*scale, head_width=0.1*scale, head_length=0.1*scale,
            fc=None, ec=[1.,0,0,0.75],
            clip_on=False,
            length_includes_head=True)

fig.subplots_adjust(wspace=0.1)
```



The `plt.imshow` command simply maps the minimum value of the image to 0 and the maximum value of the image to 1, i.e. it applies standard *0-1 min-max normalization*. This explains the dark appearance once we add in a bright pixel, as most of the image gets mapped to the bottom half of the available color map.

This is annoying when visualizing images next to each other, but it is particularly problematic when we need to standardize the images we feed into a neural network. We can choose to make all images 0-1, 0-255, etc. (and these can go above or below the minimum and maximum by a little), but it is much harder for a network to learn foreground from background if the images are chaotically rescaled like the above example (chaotic meaning that the image darkening is highly sensitive to the particular condition of whether or not there are saturated pixels).

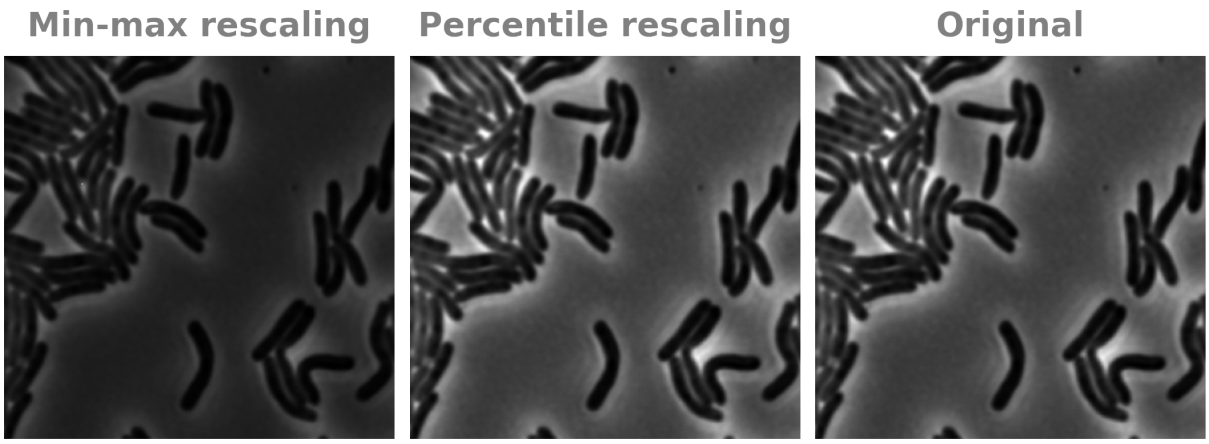
We solve this by normalizing the image not by the absolute min and max, but by percentiles. We set pixels at or below the 0.01 percentile to 0 and those at or above the 99.99th percentile to 1. (Cellpose uses 1 and 99, but this will mess up images with very few cells compared to background).

```
from omnipose.utils import normalize99

im_fixed = normalize99(im_bad)
# print('normalize99() fixes the image:')
# imshow(np.hstack((im_bad,im_fixed)),2,cmap='gray')

f = 1
c = [0.5]*3
fontsize=10

titles = ['Min-max rescaling','Percentile rescaling','Original']
ims = [im_bad,im_fixed,im]

# Number of subplots in the right column
n = len(ims)
h, w = ims[0].shape[:2]

```

(continues on next page)

(continued from previous page)

```
sf = w
p = 0.0001*w # needs to be defined as fraction of width for aspect ratio to work?
h /= sf
w /= sf

# Calculate positions of subplots
left = np.array([i*(w+p) for i in range(n)])*1.
bottom = np.array([0]*n)*1.
width = np.array([w]*n)*1.
height = np.array([h]*n)*1.

max_w = left[-1]+width[-1]
max_h = bottom[-1]+height[-1]

sf = max_w - (n-1)*p
left /= sf
bottom /= sf
width /= sf
height /= sf


s = 6.5 * 3/4 # make it so that these appear the same size
fig = plt.figure(figsize=(s,s), frameon=False, dpi=300)


for i in range(n):
    ax = fig.add_axes([left[i], bottom[i], width[i], height[i]])
    ax.imshow(ims[i],cmap='gray')
    ax.axis('off')
    ax.set_title(titles[i],c=c,fontsize=fontsize,fontweight="bold")
```



With an image that has been properly normalized from 0 to 1, we can further adjust it. Right now we cannot see a lot of detail in the dark parts of the image; what we can do is raise the image to some power, called **gammma adjustment**. Because $0^x = 0$ and $1^x = 1$, we can make the image globally brighter or darker without affecting the total range:

```
# %matplotlib inline
from omnipose.utils import sinebow
```

(continues on next page)

(continued from previous page)

```

im_gamma = []
gamma = [0.25, 0.5, 1, 2]
N = len(gamma)

dpi = 300
mpl.rcParams['figure.dpi'] = dpi
mpl.rcParams["axes.facecolor"] = [0,0,0,0]
px = 1/plt.rcParams['figure.dpi']  # pixel in inches
axcol = [0.5]*3
matplotlib_inline.backend_inline.set_matplotlib_formats('svg')

w=6
labelsize = 10
# fig1,ax = plt.subplots(figsize=(w,w/N),facecolor='#0000',frameon=False,)
fig1 = plt.figure(figsize=(w,w/N),
                  # frameon=False,
                  facecolor='#0000',
                  # tight_layout={'pad':10}
                 )
offset = 0.05
ax = fig1.add_axes([offset,0,1-offset,1])
fig1.subplots_adjust(left=offset, bottom=0, right=1, top=1, wspace=0, hspace=0)

color = sinebow(N+1)
for j,g in enumerate(gamma):
    i = im_fixed**g
    im_gamma.append(i)
    ax.hist(i.flatten(),
            bins=100,
            label='gamma = {}'.format(g),
            color=color[j+1],
            histtype='step',
            density=True)

l = ax.legend(prop={'size': labelsize},
              frameon=False,
              bbox_to_anchor=(1, 1),
              loc='upper right',
              borderaxespad=0.)
for text,c in zip(l.get_texts(),[color[i] for i in range(1,N+1)]):
    text.set_color(c)

for item in l.legend_handles:
    item.set_visible(False)

ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.patch.set_alpha(0.0)
plt.xlabel('Intensity',size=labelsize,c=axcol,fontweight='bold')
plt.ylabel('PDF',size=labelsize,c=axcol,fontweight='bold')

```

(continues on next page)

(continued from previous page)

```
ax.yaxis.set_label_coords(-offset,0.5)


ax.tick_params(axis='both', colors=axcol)
ax.spines['bottom'].set_color(axcol)
ax.spines['left'].set_color(axcol)

plt.show()


# %matplotlib inline
%config InlineBackend.figure_formats = ['png']
# mpl.use('Agg')

h,w = im.shape[-2:]

# Number of subplots in the right column
n = len(im_gamma)

sf = w
p = 0.05
h /= sf
w /= sf

# Calculate positions of subplots
left = np.array([i*(w+p) for i in range(n)])*1.
bottom = np.array([0]*n)*1.
width = np.array([w]*n)*1.
height = np.array([h]*n)*1.

max_w = left[-1]+width[-1]
max_h = bottom[-1]+height[-1]

sw = max_w
sh = max_h

sf = max(sw,sh)
left /= sw
bottom /= sh
width /= sw
height /= sh

# Create figure
s = 6
fig2 = plt.figure(figsize=(s,s*sh/sw), frameon=False, dpi=600)#,tight_layout={'pad':0})
# fig2.patch.set_facecolor([0]*4)

# Add subplots
axes = []
for i in range(n):
    ax = fig2.add_axes([left[i], bottom[i], width[i], height[i]])
    axes.append(ax)
```

(continues on next page)

(continued from previous page)

```


# fig2, axes = plt.subplots(1,4, figsize=(w,w/4))
# fig2.patch.set_facecolor([0]*4)

sz = im.shape
pad = 10
width = 30
slc = (slice(pad,pad+width),slice(sz[1]-(pad+width),sz[1]-pad),Ellipsis)

for i,(ax,ig) in enumerate(zip(axes,im_gamma)):
    ax.axis('off')
    ig = np.stack([ig]*3+[np.ones_like(ig)],axis=-1)
    ig[slc] = color[i+1]
    ax.imshow(ig)

fig2.subplots_adjust(left=0, bottom=0, right=1, top=1, wspace=0, hspace=0)
# plt.show()
```

```
<Figure size 1800x450 with 1 Axes>
```

```
<Figure size 3600x849.766 with 4 Axes>
```

```
datadir = omnidir.parent
# fig1.savefig(os.path.join(datadir,'Dissertation','figures','gammahist.pdf'),
#     transparent=True,bbox_inches='tight',pad_inches=0)

file = os.path.join(datadir,'Dissertation','figures','gammahist.pdf')
if os.path.isfile(file): os.remove(file)
fig1.savefig(file,transparent=True,bbox_inches='tight',pad_inches=0)

file = os.path.join(datadir,'Dissertation','figures','gammapics.png')
if os.path.isfile(file): os.remove(file)
fig2.savefig(file,transparent=True,pad_inches=0,bbox_inches='tight')
```

Note that fractional powers only work (without going into complex numbers) if the image values are nonnegative.

19.3 Semantic gamma normalization

We can next use image segmentation in combination with gamma adjustment to normalize image brightness. This is very handy for making figures with images coming from different microscopes or optical configurations. To demonstrate, let's load in the image set from our *mono_channel_bact* notebook and the corresponding masks we made with Omnipose.

```
from pathlib import Path
import os
from cellpose_omni import io, transforms

mask_filter='_cp_masks'
img_names = io.get_image_files(basedir,mask_filter,look_one_level_down=True)
```

(continues on next page)

(continued from previous page)

```
mask_names = io.get_label_files(img_names,subfolder='masks')
imgs =  [io.imread(i) for i in img_names]
imgs = [im if im.ndim==2 else im[...,0] for im in imgs]
masks = [io.imread(m) for m in mask_names]
```

Now we will compare standard normalization to what I am calling "semantic gamma normalization". My implementation of it can be found in `omnipose.utils`, which simply answers the question: "what is the power to which I need to raise my image such that the average background becomes equal to a given value?". From left to right, I plot `im/max(im.dtype)` (so min>=0 and max=1), 0-1 remapping of `im` (`recsale`), percentile remapping of `im` (`normalize99`), gamma normalization to background of 1/3, and gamma normalization of background to 1/2. The output has been set to use the same colormap and interpolation (`vmin` and `vmax` are otherwise set by the min and max of the image).

```
from omnipose.utils import rescale, normalize_image

textcolor = [0.5]*3

f = 1
labelsize = 8*f
fontsize = 4*f
fontsize3 = 12*f

# Assume the images are stored in a nested list
images = []
for im, mask in zip(imgs,masks):

    # format the image
    im = transforms.move_min_dim(im) # move the channel dimension last
    if len(im.shape)>2:
        im = im[:,:,1]

    im_raw = im/np.iinfo(im.dtype).max

    im_rescale = rescale(im)
    im_norm = normalize99(im)
    im_gamma_3 = normalize_image(im, mask>0, target=1/3)
    im_gamma_2 = normalize_image(im, mask>0, target=1/2)

    images.append([im_raw,im_rescale,im_norm,im_gamma_3, im_gamma_2])

images = images[::-1]
titles = ['raw','minmax','percentile','$\gamma=1/3$', '$\gamma=1/2$']

kwargs = {'cmap':'gray','vmin':0,'vmax':1}
numt = [str(i) for i in range(len(images))]
fig = omnipose.plot.image_grid(images,column_titles=titles,
                                    # row_titles=numt,
                                    fig_scale=6,
                                    outline=True,
                                    **kwargs)
plt.show()

```

(continues on next page)

(continued from previous page)

```
# fig = omnipose.plot.image_grid(images[::-1],
#                                column_titles=numt,
#                                row_titles=titles,
#                                fig_scale=4,
#                                outline=True,
#                                order='ji',
#                                **kwargs)
# plt.show()
```

```
<Figure size 1800x2376.53 with 35 Axes>
```

The first column provides an essentially 'raw' view of the image, as it has not been shifted or stretched relative to the original max and min of its data type. As noted in the segmentation notebook, that first image is super dark because it is an 8-bit image (0-255), but only takes on values from 4 to 22. My code above divides by 255 for uint8 images and 65535 for the last uint16 image.

The second and third columns do stretch the image to fill the whole 0-1 range, but you can see how the images still have different background intensity. My function in columns 4 and 5 normalize the background to a constant value. Well-exposed bacterial phase contrast images seem to have a 'natural' background value of about 1/3.

CHAPTER

TWENTY

LOGO

This is a little cute: Omnipose can "segment" text using the `bact_phase_omni` model. Semantic segmentation of uniform, disjoint shapes on a uniform background is absolutely no feat, but it is amusing that a neural network trained purely on phase contrast images of bacteria gives such reasonable output on something so different from the training set. Also, the over-segmentation at cusps hints that the network has learned to pick up on local morphology.

To make the Omnipose logo/title/favicon, I first generate some rasterized text images with roughly the same mean diameter as the bacteria in my training set:

```
# Make some text images

from PIL import Image, ImageDraw, ImageFont
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('dark_background')
import matplotlib as mpl
%matplotlib inline
mpl.rcParams['figure.dpi'] = 300

from omnipose.utils import bbox_to_slice

tsizes = [60]
texts = ["Omnipose","O"]
imgs = []
for textsize in tsizes:
    fonts = [ImageFont.truetype(f, textsize) for f in ["SFNSRounded.ttf"]]
    # fonts = [ImageFont.truetype(f, textsize) for f in ["Arial.ttf"]]
    for text in texts:
        for font in fonts:
            size = np.array([textsize*len(text)*2, textsize*2])
            im = Image.new("RGB", tuple(size), "white")
            d = ImageDraw.Draw(im)
            center = size/2
            anchor = "mm"
            d.text(center, text, fill="black", anchor=anchor, font=font)
            bbox = d.textbbox(center, text, anchor=anchor, font=font)
            bbox = [bbox[1],bbox[0],bbox[3],bbox[2]] # reverse x, y
            im = np.array(im)
            shape = im.shape[:2]
            slc = bbox_to_slice(bbox,shape,pad = 3)
            im = im[slc]
            imgs.append(im)
```

(continues on next page)

(continued from previous page)

```
            fig = plt.figure(figsize=(1,1))
            fig.patch.set_facecolor([0]*4)


            plt.imshow(im)
            plt.axis('off')
            plt.show()
```

20.1 Segmentation

I will then segment these image with the standard settings:

```
from cellpose_omni import plot, models, core
import omnipose

model_name = 'bact_phase_omni'
use_GPU = core.use_gpu()
model = models.CellposeModel(gpu=use_GPU, model_type=model_name)


chans = [0,0] #this means segment based on first channel, no second channel
nimg = len(imgs)
n = range(nimg)

# define parameters
mask_threshold = 1
verbose = 0
use_gpu = use_GPU
transparency = True
rescale=None
omni = True
flow_threshold = 0
resample = True
cluster = False

masks, flows, styles = model.eval([imgs[i] for i in n],
                                  channels=chans,
                                  rescale=rescale,
                                  mask_threshold=mask_threshold,
```

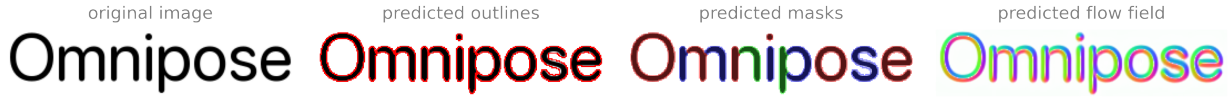
(continues on next page)

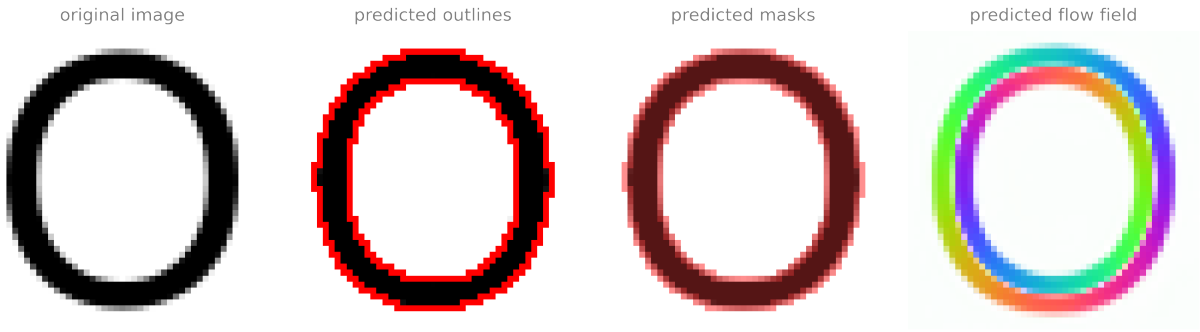
(continued from previous page)

```
                                    transparency=transparency,
                                    flow_threshold=flow_threshold,
                                    omni=omni,resample=resample,
                                    verbose=verbose,
                                    cluster=cluster)


mpl.rcParams['figure.dpi'] = 300
plt.style.use('dark_background')

for idx,i in enumerate(n):

    maski = masks[idx] # get masks
    bdi = flows[idx][-1] # get boundaries
    flowi = flows[idx][0] # get RGB flows

    # set up the output figure to better match the resolution of the images
    # f = 10
    # szX = maski.shape[-1]/mpl.rcParams['figure.dpi']*f
    # szY = maski.shape[-2]/mpl.rcParams['figure.dpi']*f
    szX,szY = 10,10
    fig = plt.figure(figsize=(szY,szX*4))
    fig.patch.set_facecolor([0]*4)

    plot.show_segmentation(fig, omnipose.utils.normalize99(imgs[i]),
                           maski, flowi, bdi, channels=chans, omni=True,␣
↪interpolation=None)

    plt.tight_layout()
    plt.show()
```

```
2023-08-03 20:42:12,194 [INFO] ** TORCH GPU version installed and working. **
2023-08-03 20:42:12,194 [INFO] >>bact_phase_omni<< model set to be used
2023-08-03 20:42:12,195 [INFO] ** TORCH GPU version installed and working. **
2023-08-03 20:42:12,195 [INFO] >>>> using GPU
```

```
  0%|          | 0/2 [00:00<?, ?it/s]
```





I landed on this font because it is one of Apple's system defaults (and therefore works well with the system fonts used on our website when viewed on Apple devices), and I chose this scale because it is very close to bacteria and showed a good amount of 'segmentation' in the M, N, P, and E from purely local morphology (cusps). This gives reasonable output at higher-resolution text (wider 'cells'), but it starts to hallucinate output between objects if the size gets too large.

20.2 Adjusting transparency

The transparency (alpha channel) is set by the flow magnitude, and the color (RGB channels) is set by the flow angle according to a shifted sinebow relation:

```
angles = np.arctan2(dP[1], dP[0])+np.pi
r = ((np.cos(angles)+1)/2)
g = ((np.cos(angles+2*np.pi/3)+1)/2)
b =((np.cos(angles+4*np.pi/3)+1)/2)
```

(a is just a constant, not alpha). The slight tinge of green comes from the fact that `np.arctan2(0,0)=0` and `((np.cos(0+2*np.pi/3)+1)/2) = 1/4`. I'll try two ways to remove it: first by removing any average background bias, second by adjusting the alpha channel so that the background alpha is 0 on average.

```
import skimage.io
import os
from omnipose.utils import normalize99, rescale
from scipy.ndimage import zoom
from pathlib import Path

omnidir = Path(omnipose.__file__).parent.parent
basedir = os.path.join(omnidir,'docs','_static')
names = ['logo.png','icon.ico']
ext = '.png'

for idx,i in enumerate(n):

    maski = masks[idx]
    flowi = flows[idx][0]
    dPi = flows[idx][1]
    bias = [np.mean(d[maski==0]) for d in dPi]
    angle = np.arctan2(bias[1], bias[0]) / np.pi
    print('avarage bias is {}, average angle is {} pi rad.'.format(bias,angle))
    dPi_new = np.stack([np.clip(d - b,-np.inf,np.inf) for d,b in zip(dPi,bias)])
```

(continues on next page)

(continued from previous page)

```
    flowi_new = plot.dx_to_circ(dPi_new,transparency=True)

    flowi_3 = flowi.copy()
    alpha = flowi_3[...,-1]
    flowi_3[...,-1] = rescale(np.clip(alpha-np.mean(alpha[maski==0]),0,np.inf))*255

    f = 30
    szX = maski.shape[-1]/mpl.rcParams['figure.dpi']*f
    szY = maski.shape[-2]/mpl.rcParams['figure.dpi']*f
    fig = plt.figure(figsize=(szY,szX*4))
    fig.patch.set_facecolor([0]*4)

    plt.imshow(np.hstack([flowi,flowi_new,flowi_3]))
    plt.axis('off')
    plt.show()
    # aplha channel correction is the winner
    # also rescale the image without interpolation so that, when displayed as favicon␣
↪etc., it is not as smoothed out - we want to show real output
    skimage.io.imsave(os.path.join(basedir,names[idx]),zoom(flowi_3,(3,)*(flowi.ndim-
↪1)+(1,),order=0))

```

```
avarage bias is [0.06506971, 0.06305661], average angle is 0.24499917011957278 pi rad.
```


```
avarage bias is [0.07233106, 0.06707774], average angle is 0.23801084309752654 pi rad.
```

Turns out that subtracting off the flow component bias introduces some over-correction in places, leading to some discoloration. So, alpha adjustment it is. It might be hard for you to see it, but I can. This level of pixel-peeping is how I made my ground-truth data ;-)

20.3 Exporting

Favicons need to be a particular resolution. For now I am making a multi-scale .ico, but that isn't working properly on Safari (too pixelated). Seems like multiple separate PNGs is the way to go moving forward.

```
from PIL import Image
filename = os.path.join(basedir,names[-1])
zimgs = []

for j,sz in enumerate([(32,32), (128,128), (180,180), (192,192)]):
    scale = np.array(sz)/np.array(flowi_3.shape[0:2])
    zimg = zoom(flowi_3,tuple(scale)+(1,),order=(np.max(scale)<1))
    zimgs.append(zimg)
    # plt.imshow(zimg)
    # plt.axis('off')
    # plt.show()
    # zimg.shape

icon = Image.fromarray(zimgs[0], 'RGBA')
icon.save(filename,append_images=[Image.fromarray(z, 'RGBA') for z in zimgs[1:]])
```

PYTHON MODULE INDEX

INDEX

N

O

P

R

S

T

U

W