# BASICS:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Installations</td>
<td>3</td>
</tr>
<tr>
<td>1.1 Built-in model directory</td>
<td>3</td>
</tr>
<tr>
<td>1.2 M1 Mac installation</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Common issues</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Dependencies</td>
<td>4</td>
</tr>
<tr>
<td>2 GUI</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Starting the GUI</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Using the GUI</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Segmentation options</td>
<td>6</td>
</tr>
<tr>
<td>2.4 Training your own cellpose model</td>
<td>6</td>
</tr>
<tr>
<td>2.5 Contributing training data</td>
<td>7</td>
</tr>
<tr>
<td>2.6 Keyboard shortcuts</td>
<td>8</td>
</tr>
<tr>
<td>3 Inputs</td>
<td>9</td>
</tr>
<tr>
<td>3.1 3D segmentation</td>
<td>9</td>
</tr>
<tr>
<td>4 Settings</td>
<td>11</td>
</tr>
<tr>
<td>4.1 Channels</td>
<td>11</td>
</tr>
<tr>
<td>4.2 Diameter</td>
<td>12</td>
</tr>
<tr>
<td>4.3 Resample</td>
<td>12</td>
</tr>
<tr>
<td>4.4 Flow threshold (aka model fit threshold in GUI)</td>
<td>12</td>
</tr>
<tr>
<td>4.5 Mask threshold</td>
<td>13</td>
</tr>
<tr>
<td>4.6 3D settings</td>
<td>13</td>
</tr>
<tr>
<td>5 Outputs</td>
<td>15</td>
</tr>
<tr>
<td>5.1 _seg.npy output</td>
<td>15</td>
</tr>
<tr>
<td>5.2 PNG output</td>
<td>16</td>
</tr>
<tr>
<td>5.3 ROI manager compatible output for ImageJ</td>
<td>16</td>
</tr>
<tr>
<td>5.4 Plotting functions</td>
<td>17</td>
</tr>
<tr>
<td>6 Models</td>
<td>19</td>
</tr>
<tr>
<td>6.1 Model Zoo</td>
<td>19</td>
</tr>
<tr>
<td>6.2 Full built-in models</td>
<td>19</td>
</tr>
<tr>
<td>6.2.1 Cytoplasm model (‘cyto’)</td>
<td>20</td>
</tr>
<tr>
<td>6.2.2 Nucleus model (‘nuclei’)</td>
<td>20</td>
</tr>
<tr>
<td>6.2.3 Cytoplasm 2.0 model (‘cyto2’)</td>
<td>20</td>
</tr>
<tr>
<td>6.3 Other built-in models</td>
<td>20</td>
</tr>
<tr>
<td>6.3.1 TissueNet models</td>
<td>20</td>
</tr>
<tr>
<td>6.3.2 LiveCell models</td>
<td>20</td>
</tr>
<tr>
<td>6.4 User-trained models</td>
<td>21</td>
</tr>
</tbody>
</table>
cellpose is an anatomical segmentation algorithm written in Python 3 by Carsen Stringer and Marius Pachitariu. For support, please open an issue.

We make pip installable releases of cellpose, here is the pypi. You can install it as `pip install cellpose[gui]`. You can try it out without installing at cellpose.org. Also check out these resources:

**Cellpose 2.0**
- paper on bioRxiv
- `talk <https://www.youtube.com/watch?v=3ydtAhfq6H0>`_
- twitter thread
- human-in-the-loop training protocol video

**Cellpose 1.0**
- paper on bioRxiv (see figure 1 below) and in nature methods
- twitter `thread`
- Marius’s talk
Figure 1: Model architecture. a, Procedure for transforming manually annotated masks into a vector flow representation that can be predicted by a neural network. A simulated diffusion process started at the center of the mask is used to derive spatial gradients that point towards the center of the cell, potentially indirectly around corners. The X and Y gradients are combined into a single normalized direction from 0° to 360°. b, Example spatial flows for cells from the training dataset. c, A neural network is trained to predict the horizontal and vertical flows, as well as whether a pixel belongs to any cell. The three predicted maps are combined into a flow field. d shows the details of the neural network which contains a standard backbone neural network that downsamples and then upsamples the feature maps, contains skip connections between layers of the same size, and global skip connections from the image styles, computed at the lowest resolution, to all the successive computations. e, At test time, the predicted flow fields are used to construct a dynamical system with fixed points whose basins of attraction represent the predicted masks. Informally, every pixel “follows the flows” along the predicted flow fields towards their eventual fixed point. f, All the pixels that converge to the same fixed point are assigned to the same mask.
For basic install instructions, look up the main github readme.

1.1 Built-in model directory

By default, the pretrained cellpose models are downloaded to $HOME/.cellpose/models/. This path on linux would look like /home/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’d like to download the models to a different directory, and are using the command line or the GUI, before you run python -m cellpose ..., you will need to always set the environment variable CELLPOSE_LOCAL_MODELS_PATH (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 cellpose in a jupyter notebook, run this code at the beginning of your notebook before you import cellpose:

```python
import os
os.environ['CELLPOSE_LOCAL_MODELS_PATH'] = '/PATH_FOR_MODELS/
```

1.2 M1 Mac installation

Please use the instructions provided on image.sc <https://forum.image.sc/t/cellpose-on-macos-m1-pro-apple-silicon-arm64/68018/4> by Peter Sobolewski.

1.3 Common issues

If you are having issues with CUDA on Windows, or want to use Cuda Toolkit 10, please follow these instructions:

```bash
conda create -n cellpose pytorch=1.8.2 cudatoolkit=10.2 -c pytorch-lts
conda activate cellpose
pip install cellpose
```

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 3.7 and not python 3.8 installed.

If you are on Yosemite Mac OS or earlier, PyQt doesn’t work and you won’t be able to use the graphical interface for cellpose. More recent versions of Mac OS are fine. The software has been heavily tested on Windows 10 and Ubuntu 18.04, and less well tested on Mac OS. Please post an issue if you have installation problems.

1.4 Dependencies

cellpose relies on the following excellent packages (which are automatically installed with conda/pip if missing):

- pytorch
- pyqtgraph
- PyQt5
- numpy (>=1.16.0)
- numba
- numba
- scipy
- scikit-image
- natsort
- matplotlib
2.1 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:

```python
python -m cellpose
```

The first time cellpose runs it downloads the latest available trained model weights from the website.

You can drag and drop images (.tif, .png, .jpg, .gif) into the GUI and run Cellpose, and/or manually segment them. 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 you opened the GUI with. For example data, see cellpose website. For best accuracy and runtime performance, resize images so cells are less than 100 pixels across.

For multi-channel, multi-Z tiff’s, the expected format is Z x channels x Ly x Lx.

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

**Note:** Since the output file is in the same folder as the loaded image with _seg.npy appended, make sure you have WRITE access in the folder, otherwise the file will not save.

2.2 Using the GUI

The GUI serves two main functions:

1. Running the segmentation algorithm.
2. Manually labelling data.
3. (NEW) Fine-tuning a pretrained cellpose model on your own data.

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.

Note: 3D labelling will fill in unlabelled z-planes so that you do not have to densely label, for example you can skip some planes.

2.3 Segmentation options

SIZE: you can manually enter the approximate diameter for your cells, or press “calibrate” to let the model estimate it. The size is represented by a disk at the bottom of the view window (can turn this disk off by unchecking “scale disk on”).

use GPU: if you have installed the cuda version of mxnet, then you can activate this, but it won’t give huge speedups when running single images in the GUI.

MODEL: there is a cytoplasm model and a nuclei model, choose what you want to segment
CHAN TO SEG: this is the channel in which the cytoplasm or nuclei exist
CHAN2 (OPT): if cytoplasm model is chosen, then choose the nuclear channel for this option

2.4 Training your own cellpose model

Check out this video to learn the process.

1. Drag and drop an image from a folder of images with a similar style (like similar cell types).
2. Run the built-in models on one of the images using the “model zoo” and find the one that works best for your data. Make sure that if you have a nuclear channel you have selected it for CHAN2.
3. Fix the labelling by drawing new ROIs (right-click) and deleting incorrect ones (CTRL+click). The GUI autosaves any manual changes (but does not autosave after running the model, for that click CTRL+S). The segmentation is saved in a _seg.npy file.
4. Go to the “Models” menu in the File bar at the top and click “Train new model…” or use shortcut CTRL+T.
5. Choose the pretrained model to start the training from (the model you used in #2), and type in the model name that you want to use. The other parameters should work well in general for most data types. Then click OK.
6. The model will train (much faster if you have a GPU) and then auto-run on the next image in the folder. Next you can repeat #3-#5 as many times as is necessary.
7. The trained model is available to use in the future in the GUI in the “custom model” section and is saved in your image folder.

If you have 3D data, please save random XY, YZ and XZ slices through your 3D data, ideally sufficiently spaced from each other so the information each slice has is distinct. Then put these slices into a folder and start the human-in-the-loop training. You can then use the new custom model on new 3D data.

**Note:** You can only start training with one of the built-in Cellpose models or from scratch. When you start training from a built-in model or from scratch each time, then you are training the network on all the previously labelled images in the folder and weighting them equally in your training set.

If you restart from a previous retraining, you are biasing the network towards the earlier images it has already been trained on. Conversely, if you have created a custom model with different images, and you retrain that model, then you are downweighting the images that you have already trained on and excluded from your new training set. Therefore, we recommend having all images that you want to be trained for the same model in the same folder so they are all used.

See the Models doc for info on the new model zoo and suggestion mode.

### 2.5 Contributing training data

We are very excited about receiving community contributions to the training data and re-training the cytoplasm model to make it better. Please follow these guidelines:

1. Run cellpose on your data to see how well it does. Try varying the diameter, which can change results a little.
2. If there are relatively few mistakes, it won’t help much to contribute labelled data.
3. If there are consistent mistakes, your data is likely very different from anything in the training set, and you should expect major improvements from contributing even just a few manually segmented images.
4. For images that you contribute, the cells should be at least 10 pixels in diameter, and there should be **at least** several dozens of cells per image, ideally ~100. If your images are too small, consider combining multiple images into a single big one and then manually segmenting that. If they are too big, consider splitting them into smaller crops.
5. For the manual segmentation, please try to outline the boundaries of the cell, so that everything (membrane, cytoplasm, nucleus) is inside the boundaries. Do not just outline the cytoplasm and exclude the membrane, because that would be inconsistent with our own labelling and we wouldn’t be able to use that.
6. Do not use the results of the algorithm in any way to do contributed manual segmentations. This can reinforce a vicious circle of mistakes, and compromise the dataset for further algorithm development.

If you are having problems with the nucleus model, please open an issue before contributing data. Nucleus images are generally much less diverse, and we think the current training dataset already covers a very large set of modalities. Additionally, you can run a non-nuclear model on nuclear data such as cyto.
### 2.6 Keyboard shortcuts

<table>
<thead>
<tr>
<th>Keyboard shortcuts</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL+H</td>
<td>help</td>
</tr>
<tr>
<td>=/+ // -</td>
<td>zoom in // zoom out</td>
</tr>
<tr>
<td>CTRL+Z</td>
<td>undo previously drawn mask/stroke</td>
</tr>
<tr>
<td>CTRL+0</td>
<td>clear all masks</td>
</tr>
<tr>
<td>CTRL+L</td>
<td>load image (can alternatively drag and drop image)</td>
</tr>
<tr>
<td>CTRL+S</td>
<td>SAVE MASKS IN IMAGE to _seg.npy file</td>
</tr>
<tr>
<td>CTRL+T</td>
<td>start model training using _seg.npy files</td>
</tr>
<tr>
<td>CTRL+P</td>
<td>load _seg.npy file (note: it will load automatically with image if it exists)</td>
</tr>
<tr>
<td>A/D or LEFT/RIGHT</td>
<td>cycle through images in current directory</td>
</tr>
<tr>
<td>W/S or UP/DOWN</td>
<td>change color (RGB/gray/red/green/blue)</td>
</tr>
<tr>
<td>R / G / B</td>
<td>press to toggle RGB and Red or Green or Blue</td>
</tr>
<tr>
<td>PAGE-UP / PAGE-DOWN</td>
<td>change to flows and cell prob views (if segmentation computed)</td>
</tr>
<tr>
<td>X</td>
<td>turn masks ON or OFF</td>
</tr>
<tr>
<td>Z</td>
<td>toggle outlines ON or OFF</td>
</tr>
<tr>
<td>, / .</td>
<td>increase / decrease brush size for drawing</td>
</tr>
</tbody>
</table>
CHAPTER THREE

INPUTS

You can use tiffs or PNGs or JPEGs. We use the image loader from scikit-image. Single plane images can read into data as nY x nX x channels or channels x nY x nX. Then the channels settings will take care of reshaping the input appropriately for the network. Note the model also rescales the input for each channel so that 0 = 1st percentile of image values and 1 = 99th percentile.

If you want to run multiple images in a directory, use the command line or a jupyter notebook to run cellpose.

3.1 3D segmentation

Tiffs with multiple planes and multiple channels are supported in the GUI (can drag-and-drop tiffs) and supported when running in a notebook. Multiplane images should be of shape nplanes x channels x nY x nX or as nplanes x nY x nX. You can test this by running in python

```python
import tifffile
data = tifffile.imread('img.tif')
print(data.shape)
```

If drag-and-drop of the tiff into the GUI does not work correctly, then it’s likely that the shape of the tiff is incorrect. If drag-and-drop works (you can see a tiff 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.

When running cellpose in a notebook, set do_3D=True to enable 3D processing. 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 tiffs in the folder as 3D tiffs if possible).

If the 3D 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.

If drag-and-drop doesn't work because of the shape of your tiff, you need to transpose the tiff and resave 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 an 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. 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.
The important settings are described on this page. See the *Cellpose class* for all run options.

Here is an example of calling the Cellpose class and running a list of images for reference:

```python
from cellpose import models
from cellpose.io import imread

# model_type='cyto' or model_type='nuclei'
model = models.Cellpose(gpu=False, model_type='cyto')

files = ['img0.tif', 'img1.tif']
imgs = [imread(f) for f in files]
masks, flows, styles, diams = model.eval(imgs, diameter=None, channels=[0,0],
                                        flow_threshold=0.4, do_3D=False)
```

You can make lists of channels/diameter for each image, or set the same channels/diameter for all images as shown in the example above.

### 4.1 Channels

There are two channels inputs. The first channel is the channel you want to segment. The second channel is an optional channel that is helpful in models trained with images with a nucleus channel. See more details in the models page.

1. 0=grayscale, 1=red, 2=green, 3=blue
2. 0=None (will set to zero), 1=red, 2=green, 3=blue

Set channels to a list with each of these elements, e.g. `channels = [0,0]` if you want to segment cells in grayscale or for single channel images, or `channels = [2,3]` if you green cells with blue nuclei.

On the command line the above would be `--chan 0 --chan2 0` or `--chan 2 --chan2 3`. 
4.2 Diameter

The cellpose models have been trained on images which were rescaled to all have the same diameter (30 pixels in the case of the cyto model and 17 pixels in the case of the nuclei model). Therefore, cellpose needs a user-defined cell diameter (in pixels) as input, or to estimate the object size of an image-by-image basis.

The automated estimation of the diameter is a two-step process using the style vector from the network, a 64-dimensional summary of the input image. We trained a linear regression model to predict the size of objects from these style vectors on the training data. On a new image the procedure is as follows.

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 ROIs. Take the final estimated size as the median diameter of the predicted ROIs.

For automated estimation set diameter = None. However, if this estimate is incorrect please set the diameter by hand.

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

4.3 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 determining the flows (dX, dY, cellprob), 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 ROIs when the cells are large but will be slower in case; resample=False will find more ROIs when the cells are small but will be slower in this case. By default in versions >=1.0 resample=True.

The nuclear model in cellpose is trained on two-channel images, where the first channel is the channel to segment, and the second channel is always set to an array of zeros. Therefore set the first channel as 0=grayscale, 1=red, 2=green, 3=blue; and set the second channel to zero, e.g. channels = [0,0] if you want to segment nuclei in grayscale or for single channel images, or channels = [3,0] if you want to segment blue nuclei.

If the nuclear model isn’t working well, try the cytoplasmic model.

4.4 Flow threshold (aka model fit threshold in GUI)

Note there is nothing keeping the neural network from predicting horizontal and vertical flows that do not correspond to any real shapes at all. In practice, most predicted flows are consistent with real shapes, because the network was only trained on image flows that are consistent with real shapes, but sometimes when the network is uncertain it may output inconsistent flows. To check that the recovered shapes after the flow dynamics step are consistent with real ROIs, we recompute the flow gradients for these putative predicted ROIs, and compute the mean squared error between them and the flows predicted by the network.

The flow_threshold parameter is the maximum allowed error of the flows for each mask. The default is flow_threshold=0.4. Increase this threshold if cellpose is not returning as many ROIs as you’d expect. Similarly, decrease this threshold if cellpose is returning too many ill-shaped ROIs.
4.5 Mask threshold

The network predicts 3 outputs: flows in X, flows in Y, and cell “probability”. The predictions the network makes of the probability are the inputs to a sigmoid centered at zero ($1 / (1 + e^{-x})$), so they vary from around -6 to +6. The pixels greater than the `cellprob_threshold` are used to run dynamics and determine ROIs. The default is `cellprob_threshold=0.0`. Decrease this threshold if cellpose is not returning as many ROIs as you’d expect. Similarly, increase this threshold if cellpose is returning too ROIs particularly from dim areas.

4.6 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 ROIs 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.
Internally, the network predicts 3 (or 4) outputs: (flows in Z), flows in Y, flows in X, and cellprob. The predictions the network makes of cellprob are the inputs to a sigmoid centered at zero \((1 / (1 + e^{-x}))\), so they vary from around -6 to +6.

5.1 _seg.npy output

*_seg.npy* files have the following fields:

- **filename**: filename of image
- **img**: image with chosen channels (nchan x Ly x Lx) (if not multiplane)
- **masks**: each pixel in the image is assigned to an ROI \((0 = \text{NO ROI}; 1,2,\ldots = \text{ROI labels})\)
- **colors**: colors for ROIs
- **outlines**: outlines of ROIs \((0 = \text{NO outline}; 1,2,\ldots = \text{outline labels})\)
- **chan_choose**: channels that you chose in GUI \((0=\text{gray/none}, 1=\text{red}, 2=\text{green}, 3=\text{blue})\)
- **ismanual**: element \(k\) = whether or not mask \(k\) was manually drawn or computed by the cellpose algorithm
- **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 ROIs)

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

```python
import numpy as np
from cellpose import plot, utils
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 = utils.outlines_list(dat['masks'])
```

(continues on next page)
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 import io
io.masks_flows_to_seg(images, masks, flows, diams, file_name, channels)
```

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

### 5.2 PNG output

You can save masks to PNG in the GUI.

To save masks (and other plots in PNG) using the command line, add the flag `--save_png`.

Or use the function below if running in a notebook

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

### 5.3 ROI manager compatible output for ImageJ

You can save the outlines of ROIs in a text file that’s compatible with ImageJ ROI Manager in the GUI File menu.

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

Or use the function below if running in a notebook

```
from cellpose import io, utils

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

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

```python
from cellpose 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()
```
from cellpose import models

Cellpose 2.0 now has a model zoo and options for user model training. Each model will be downloaded automatically to your models.MODELS_DIR (see Installation instructions for more details on MODELS_DIR). See paper for more details on the model zoo. You can also directly download a model by going to the URL, e.g.:

https://www.cellpose.org/models/MODEL_NAME

### 6.1 Model Zoo

All built-in models were trained with the ROIs resized to a diameter of 30.0 (diam\_mean = 30), except the 'nuclei' model which was trained with a diameter of 17.0 (diam\_mean = 17). The models will internally take care of rescaling the images given a user-provided diameter (or with the diameter from auto-diameter estimation in full models).

There is a suggestion button below the model zoo in the GUI. This runs a general model that has been trained on Cellpose, TissueNet, and LiveCell to obtain the style of the image. It uses this style to suggest which model would be best for the given image (see info in Cellpose 2.0 paper, and runs the suggested model on the image. Make sure the diameter is set to the approximate diameter of the ROIs in the image before clicking the button to ensure best performance.

### 6.2 Full built-in models

These models have a size model and 4 different training versions, each trained starting from 4 different random initial parameter sets. This means you can run with diameter=0 or --diameter 0 and the model can estimate the ROI size. Also you can set net\_avg=True or --net\_avg to average the results of the 4 models.

These models can be loaded and used in the notebook with models.Cellpose(model_type='cyto') or in the command line with python -m cellpose --pretrained_model cyto.

These models’ names (to download all the models for a class run with --net\_avg): * 'cyto': cytotorch\_0, cytotorch\_1, cytotorch\_2, cytotorch\_3, size\_cytotorch\_0.npy * 'nuclei': nucleitorch\_0, nucleitorch\_1, nucleitorch\_2, nucleitorch\_3, size\_nucleitorch\_0.npy * 'cyto2': cyto2torch\_0, cyto2torch\_1, cyto2torch\_2, cyto2torch\_3, size\_cyto2torch\_0.npy
6.2.1 Cytoplasm model (‘cyto’)  
The cytoplasm model in cellpose is trained on two-channel images, where the first channel is the channel to segment, and the second channel is an optional nuclear channel. Here are the options for each: 1. 0=grayscale, 1=red, 2=green, 3=blue. 0=None (will set to zero), 1=red, 2=green, 3=blue

Set channels to a list with each of these elements, e.g. channels = [0,0] if you want to segment cells in grayscale or for single channel images, or channels = [2,3] if you green cells with blue nuclei.

6.2.2 Nucleus model (‘nuclei’)  
The nucleus model in cellpose is trained on two-channel images, where the first channel is the channel to segment, and the second channel is always set to an array of zeros. Therefore set the first channel as 0=grayscale, 1=red, 2=green, 3=blue; and set the second channel to zero, e.g. channels = [0,0] if you want to segment nuclei in grayscale or for single channel images, or channels = [3,0] if you want to segment blue nuclei.

6.2.3 Cytoplasm 2.0 model (‘cyto2’)  
The cytoplasm 2.0 model in cellpose is trained on two-channel images, where the first channel is the channel to segment, and the second channel is an optional nuclear channel, as the cytoplasm model.

In addition to the training data in our dataset, it was trained with user-submitted images.

6.3 Other built-in models

These models do not have a size model and 4 different training versions. If the diameter is set to 0.0, then the model uses the default diam_mean for the diameter (30.0).

These models can be loaded and used in the notebook with e.g. models.CellposeModel(model_type='tissuenet') or models.CellposeModel(model_type='LC2'), or in the command line with python -m cellpose --pretrained_model tissuenet.

These models’ names are the same as their strings in the GUI.

6.3.1 TissueNet models

The 'tissuenet' model was trained on all training images from the tissuenet dataset. These images have a cytoplasm channel and a nuclear channel. The other tissuenet models ('TN1', 'TN2', and 'TN3') were trained on subsets of the tissuenet dataset that had similar characteristics.

6.3.2 LiveCell models

The 'livecell' model was trained on all training images from the livecell dataset. These images only have a cytoplasm channel. The other livecell models ('LC1', 'LC2', 'LC3', and 'LC4') were trained on subsets of the livecell dataset that had similar characteristics.
6.4 User-trained models

By default, models are trained with the ROIs resized to a diameter of 30.0 (\(\text{diam\_mean} = 30\)) – this is necessary if you want to start from a pretrained cellpose model. If you want to use a different diameter and use pretraining, we recommend performing training yourself on the cellpose dataset with that diameter so the model learns objects at that size. All user-trained models will save the \(\text{diam\_mean}\) so it will be loaded automatically along with the model weights.

Each model also saves the \(\text{diam\_labels}\) which is the mean diameter of the ROIs in the training images. This value is auto-loaded into the GUI for use with the model, or will be used if the diameter is 0 (\(\text{diameter}=0\) or \(--\text{diameter} 0\)).

These models can be loaded and used in the notebook with e.g. `models.CellposeModel(model_type='name_in_gui')` or with the full path `models.CellposeModel(pretrained_model='/full/path/to/model')`. If you trained in the GUI, you can automatically use the `model_type` argument. If you trained in the command line, you need to first add the model to the cellpose path either in the GUI in the Models menu, or using the command line: `python -m cellpose --add_model /full/path/to/model`.

Or these models can be used in the command line with `python -m cellpose --pretrained_model name_in_gui` or `python -m cellpose --pretrained_model /full/path/to/model`.
At the beginning of training, Cellpose computes the flow field representation for each mask image (dynamics.labels_to_flows).

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.

If you choose to train from scratch, you can set the median diameter you want to use for rescaling with the --diam_mean flag. We trained all model zoo models with a diameter of 30.0 pixels, except the nuclei model which used a diameter of 17 pixels, so if you want to start with a pretrained model, it will default to those values.

The models will be saved in the image directory (--dir) in a folder called models/.

The same channel settings apply for training models.

Note Cellpose expects the labelled masks (0=no mask, 1,2...,=masks) in a separate file, e.g:

```
wells_000.tif
wells_000_masks.tif
```

You can use a different ending from _masks with the --mask_filter option, e.g. --mask_filter _masks_2022.

Also, you can train a model using the labels from the GUI (_seg.npy) by using the following option --mask_filter _seg.npy.

If you use the --img_filter option (--img_filter _img in this case):

```
wells_000_img.tif
wells_000_masks.tif
```

**Warning:** The path given to --dir and --test_dir should be an absolute path.

To train on cytoplasmic images (green cyto and red nuclei) starting with a pretrained model from Cellpose (one of the model zoo models), we also have included the recommended training parameters in the command below:

```
python -m cellpose --train --dir ~/images_cyto/train/ --test_dir ~/images_cyto/test/ --pretrained_model cyto --chan 2 --chan2 1 --learning_rate 0.1 --weight_decay 0.0001 --n_epochs 100
```

You can train from scratch as well:

```
python -m cellpose --train --dir ~/images_nuclei/train/ --pretrained_model None
```

To train the cyto model from scratch using the same parameters we did, download the dataset and run
python -m cellpose --train --train_size --use_gpu --dir ~/cellpose_dataset/train/ --test_ ˓→dir ~/cellpose_dataset/test/ --img_filter _img --pretrained_model None --chan 2 -- ˓→chan2 1

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

python -m cellpose --dir ~/images_cyto/test/ --pretrained_model ~/images_cyto/test/model/ ˓→cellpose_35_0 --save_png

Training arguments

```
--train train network using images in dir
--train_size train size network at end of training
--test_dir TEST_DIR folder containing test data (optional)
--mask_filter MASK_FILTER end string for masks to run on. Default: _masks
--diam_mean DIAM_MEAN mean diameter to resize cells to during training -- if starting from pretrained models it cannot be changed from 30.0
--learning_rate LEARNING_RATE learning rate. Default: 0.2
--weight_decay WEIGHT_DECAY weight decay. Default: 1e-05
--n_epochs N_EPOCHS number of epochs. Default: 500
--batch_size BATCH_SIZE batch size. Default: 8
--min_train_masks MIN_TRAIN_MASKS minimum number of masks a training image must have to be used. Default: 5
--residual_on RESIDUAL_ON use residual connections
--style_on STYLE_ON use style vector
--concatenation CONCATENATION concatenate downsampled layers with upsampled layers (off by default which means they are added)
--save_every 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 comparison
```
OpenVINO is an optional backend for Cellpose which optimizes deep learning inference for Intel Architectures. It should be installed in the same environment with Cellpose by the following command:

```
pip install --no-deps openvino
```

Using `openvino_utils.to_openvino`, convert PyTorch model to OpenVINO one:

```python
from cellpose.contrib import openvino_utils

model = models.CellposeModel(...)

model = openvino_utils.to_openvino(model)
```
import numpy as np
import matplotlib.pyplot as plt
from cellpose import models
from cellpose.io import imread

# model_type='cyto' or 'nuclei' or 'cyto2'
model = models.Cellpose(model_type='cyto')

# list of files
# PUT PATH TO YOUR FILES HERE!
files = ['/media/carsen/DATA1/TIFFS/onechan.tif']

imgs = [imread(f) for f in files]
nimg = len(imgs)

# define CHANNELS to run segmentation on
# grayscale=0, R=1, G=2, B=3
# channels = [cytoplasm, nucleus]
# if NUCLEUS channel does not exist, set the second channel to 0
channels = [[0,0]]
# IF ALL YOUR IMAGES ARE THE SAME TYPE, you can give a list with 2 elements
# channels = [0,0] # IF YOU HAVE GRAYSCALE
# channels = [2,3] # IF YOU HAVE G=cytoplasm and B=nucleus
# channels = [2,1] # IF YOU HAVE G=cytoplasm and R=nucleus

# if diameter is set to None, the size of the cells is estimated on a per image basis
# you can set the average cell `diameter` in pixels yourself (recommended)
# diameter can be a list or a single number for all images

masks, flows, styles, diams = model.eval(imgs, diameter=None, channels=channels)

See full notebook at run_cellpose.ipynb.
10.1 Input settings

- **dir**: (string)
  directory of images

- **img_filter**: (string)
  (optional) ending of filenames (excluding extension) for processing

10.2 Run settings

These are the same settings, but set up for the command line, e.g. `channels = [chan, chan2].`

- **chan**: (int) channel to segment, ones-based because zero is gray (average of all channels)
  0 = grayscale; 1 = red; 2 = green; 3 = blue

- **chan2**: (int) nuclear or other channel, ones-based because zero means set to all zeros
  (optional); 0 = None (will be set to zero); 1 = red; 2 = green; 3 = blue

- **pretrained_model**: (string)
  cyto = cellpose cytoplasm model; nuclei = cellpose nucleus model; can also specify absolute path to model file

- **diameter**: (float)
  average diameter of objects in image, if 0 cellpose will estimate for each image, default is 30

- **use_gpu**: (bool)
  run network on GPU

- **save_outlines**: FLAG
  save outlines as text file for ImageJ

- **save_png**: FLAG
  save masks as png

- **save_tif**: FLAG
  save masks as tif

- **no_npy**: FLAG
  turn off saving of _seg.npy file

- **batch_size**: (int, optional 8)
  batch size to run tiles of size 224 x 224
10.3 Command line examples

Run python -m cellpose and specify parameters as below. 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):

```bash
given path to images
python -m cellpose --dir ~/images_cyto/test/ --pretrained_model cyto --chan 2 --chan2 3 --save_png
```

You can specify the diameter for all the images or set to 0 if you want the algorithm to estimate it on an image by image basis. Here is how to run on nuclear data (grayscale) where the diameter is automatically estimated:

```bash
python -m cellpose --dir ~/images_nuclei/test/ --pretrained_model nuclei --diameter 0. --save_png
```

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

10.4 Options

You can run the help string and see all the options:

```bash
usage: __main__.py [-h] [--use_gpu] [--gpu_device GPU_DEVICE] [--check_mkl] [--dir DIR]...
                [--look_one_level_down] [--img_filter IMG_FILTER] [--channel_axis CHANNEL_AXIS] [--z_axis Z_AXIS]...
                [--chan CHAN] [--chan2 CHAN2] [--invert] [--all_channels] [--pretrained_model...
                [--do_3D] [--diameter DIAMETER] [--stitch_threshold STITCH_THRESHOLD] [--fast_mode]
                [--flow_threshold FLOW_THRESHOLD] [--cellprob_threshold CELLPROB_THRESHOLD]
                [--anisotropy ANISOTROPY] [--exclude_on_edges] [--save_png] [--save_tif]...
                [--diam_mean DIAM_MEAN] [--learning_rate LEARNING_RATE] [--weight_decay...
                [--residual_on RESIDUAL_ON] [--style_on STYLE_ON] [--concatenation]
                [--save_every SAVE_EVERY] [--save_each] [--verbose]

cellpose parameters
optional arguments:
    -h, --help            show this help message and exit
```
(continues on next page)
--verbose show information about running and settings and save to log

**hardware arguments:**

--use_gpu use gpu if torch with cuda installed
--gpu_device GPU_DEVICE which gpu device to use
--check_mkl check if mkl working

**input image arguments:**

--dir DIR folder containing data to run or train on.
--look_one_level_down run processing on all subdirectories of current folder
--img_filter IMG_FILTER end string for images to run on
--channel_axis CHANNEL_AXIS axis of image which corresponds to image channels
--z_axis Z_AXIS axis of image which corresponds to Z dimension
--chan CHAN channel to segment; 0: GRAY, 1: RED, 2: GREEN, 3: BLUE. Default: 0
--chan2 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

**model arguments:**

--pretrained_model PRETRAINED_MODEL model to use for running or starting training
--unet run standard unet instead of cellpose flow output
--nclasses NCLASSES if running unet, choose 2 or 3; cellpose always uses 3

**algorithm arguments:**

--no_resample disable dynamics on full image (makes algorithm faster for images with large diameters)
--net_avg run 4 networks instead of 1 and average results
--no_interp do not interpolate when running dynamics (was default)
--no_norm do not normalize images (normalize=False)
--do_3D process images as 3D stacks of images (nplanes x nchan x Ly x Lx
--diameter DIAMETER cell diameter, if 0 will use the diameter of the training labels used in the model, or with
--stitch_threshold STITCH_THRESHOLD compute masks in 2D then stitch together masks with IoU>0.9 across planes
--fast_mode now equivalent to --no_resample; make code run faster by turning off resampling
--flow_threshold FLOW_THRESHOLD flow error threshold, 0 turns off this optional QC step. Default: 0.4
--cellprob_threshold CELLPROB_THRESHOLD cellprob threshold, default is 0, decrease to find more and
--larger masks
--anisotropy ANISOTROPY
anisotropy of volume in 3D

--exclude_on_edges discard masks which touch edges of image

output arguments:
--save_png save masks as png and outlines as text file for ImageJ
--save_tif save masks as tif and outlines as text file for ImageJ
--no_npy suppress saving of npy
--savedir 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)

training arguments:
--train train network using images in dir
--train_size train size network at end of training
--test_dir TEST_DIR folder containing test data (optional)
--mask_filter MASK_FILTER end string for masks to run on. use "_seg.npy" for manual annotations
--diam_mean DIAM_MEAN mean diameter to resize cells to during training -- if starting from pretrained models it cannot be changed from 30.0
--learning_rate LEARNING_RATE learning rate. Default: 0.2
--weight_decay WEIGHT_DECAY weight decay. Default: 1e-05
--n_epochs N_EPOCHS number of epochs. Default: 500
--batch_size BATCH_SIZE batch size. Default: 8
--min_train_masks MIN_TRAIN_MASKS minimum number of masks a training image must have to be used. Default: 5
--residual_on RESIDUAL_ON use residual connections
--style_on STYLE_ON use style vector
--concatenation CONCATENATION concatenate downsampled layers with upsampled layers (off by default)
--save_every 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 comparison
11.1 Cellpose class

class cellpose.models.Cellpose(gpu=False, model_type='cyto', net_avg=False, device=None)

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; 'cyto2'=cytoplasm model with additional user images

• **net_avg** (bool (optional, default False)) – loads the 4 built-in networks and averages them if True, loads one network if False

• **device** (torch device (optional, default None)) – device used for model running/training (torch.device('cuda') or torch.device('cpu')), overrides gpu input, recommended if you want to use a specific GPU (e.g. torch.device('cuda:1'))

```python
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=False, augment=False, tile=True, tile_overlap=0.1, resample=True, interp=True, flow_threshold=0.4, cellprob_threshold=0.0, min_size=15, stitch_threshold=0.0, rescale=None, progress=None, 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 False)) – 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)
- **cellprob_threshold** (float (optional, default 0.0)) – all pixels with value above threshold kept for masks, decrease to find more and larger masks
- **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
- **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] = XY flows at each pixel flows[k][2] = cell probability (if > cellprob_threshold, pixel used for dynamics) flows[k][3] = final pixel locations after Euler integration

• **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))

### 11.2 CellposeModel

**class** cellpose.models.CellposeModel(gpu=False, pretrained_model=False, model_type=None, net_avg=False, diam_mean=30.0, device=None, residual_on=True, style_on=True, concatenation=False, nchan=2)

**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)) – full path to pretrained cellpose model(s), if None or False, no model loaded
- **model_type** (str (optional, default None)) – any model that is available in the GUI, use name in GUI e.g. ‘livecell’ (can be user-trained or model zoo)
- **net_avg** (bool (optional, default False)) – loads the 4 built-in networks and averages them if True, loads one network if False
- **diam_mean** (float (optional, default 30.)) – mean ‘diameter’, 30. is built in value for ‘cyto’ model; 17. is built in value for ‘nuclei’ model; if saved in custom model file (cellpose>=2.0) then it will be loaded automatically and overwrite this value
- **device** (torch device (optional, default None)) – device used for model running / training (torch.device(‘cuda’) or torch.device(‘cpu’)), overrides gpu input, recommended if you want to use a specific GPU (e.g. torch.device(‘cuda:1’))
- **residual_on** (bool (optional, default True)) – use 4 conv blocks with skip connections per layer instead of 2 conv blocks like conventional u-nets
- **style_on** (bool (optional, default True)) – use skip connections from style vector to all upsampling layers
- **concatenation** (bool (optional, default False)) – if True, concatenate down-sampling block outputs with upsampling block inputs; default is to add
- **nchan** (int (optional, default 2)) – number of channels to use as input to network, default is 2 (cyto + nuclei) or (nuclei + zeros)

**eval**(x, batch_size=8, channels=None, channel_axis=None, z_axis=None, normalize=True, invert=False, rescale=None, diameter=None, do_3D=False, anisotropy=None, net_avg=False, augment=False, tile=True, tile_overlap=0.1, resample=True, interp=True, flow_threshold=0.4, cellprob_threshold=0.0, compute_masks=True, min_size=15, stitch_threshold=0.0, progress=None, loop_run=False, model_loaded=False)

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

• **diameter** *(float (optional, default None))* – diameter for each image, if diameter is None, set to diam_mean or diam_train if available

• **rescale** *(float (optional, default None))* – resize factor for each image, if None, set to 1.0; (only used if diameter is None)

• **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 False))* – 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)

• **cellprob_threshold** *(float (optional, default 0.0))* – all pixels with value above threshold kept for masks, decrease to find more and larger masks
- **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

- **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] = XY flow in HSV 0-255 flows[k][1] = XY flows at each pixel flows[k][2] = Cell probability (if > cellprob_threshold, pixel used for dynamics) flows[k][3] = Final pixel locations after Euler integration

- **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

**train** *(train_data, train_labels, train_files=None, test_data=None, test_labels=None, test_files=None, channels=None, 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, nimg_per_epoch=None, rescale=True, min_train_masks=5, model_name=None)*

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_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))* – Labels for test_data, where 0=no masks; 1, 2, ...=mask labels; can include flows as additional images

- **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 224x224 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 False 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
- **model_name** (*str (default, None]*) — name of network, otherwise saved with name as params + training start time

### 11.3 SizeModel

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

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** (*torch device (optional, default None]*) — device used for model running / training (torch.device('cuda') or torch.device('cpu')), overrides gpu input, recommended if you want to use a specific GPU (e.g. torch.device('cuda:1'))
- **pretrained_size** (*str*) — path to pretrained size model

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

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)
11.4 Metrics

cellpose.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

cellpose.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

cellpose.metrics.boundary_scores(masks_true, masks_pred, scales)

boundary precision / recall / Fscore

cellpose.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

```python
cellpose.metrics.mask_ious(masks_true, masks_pred)
```

return best-matched masks

## 11.5 Flows to masks

```python
cellpose.dynamics.compute_masks(dP, cellprob, p=None, niter=200, cellprob_threshold=0.0, flow_threshold=0.4, do_3D=False, min_size=15, resize=None, use_gpu=False, device=None)
```

compute masks using dynamics from dP, cellprob, and boundary

```python
cellpose.dynamics.follow_flows(dP, mask=None, niter=200, interp=True, use_gpu=True, device=None)
```

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 None))* – 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** *(float32, 3D or 4D array)* – final locations of each pixel after dynamics; [axis x Ly x Lx] or [axis x Lz x Ly x Lx]

- **inds** *(int32, 3D or 4D array)* – indices of pixels used for dynamics; [axis x Ly x Lx] or [axis x Lz x Ly x Lx]

```python
cellpose.dynamics.get_masks(p, iscell=None, rpad=20)
```

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.

```python
create masks using pixel convergence after running dynamics

Parameters

- **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

**cellpose.dynamics.labels_to_flows** *(labels, files=None, use_gpu=False, device=None, redo_flows=False)*

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

**cellpose.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)

**cellpose.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

**cellpose.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.

Parameters

- `masks` *(labelled masks 0=NO masks; 1,2,…=mask labels)*

Return type

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

cellpose.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

cellpose.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

cellpose.dynamics.steps2D(p, dP, inds, niter)

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

11.5. Flows to masks
cellpose.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

11.6 Image transforms

cellpose.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]
cellpose.transforms.convert_image(x, channels, channel_axis=None, z_axis=None, do_3D=False, normalize=True, invert=False, nchan=2)
return image with z first, channels last and normalized intensities

cellpose.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

- **img** (`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 n tiles x nchan x bsize x bsize
- **ysub** (`list`) – list of arrays with start and end of tiles in Y of length n tiles
- **xsub** (`list`) – list of arrays with start and end of tiles in X of length n tiles

```python
cellpose.transforms.move_axis(img, m_axis=-1, first=True)
```
move axis m_axis to first or last position

```python
cellpose.transforms.move_min_dim(img, force=False)
```
move minimum dimension last as channels if < 10, or force==True

```python
cellpose.transforms.normalize99(Y, lower=1, upper=99)
```
normalize image so 0.0 is 1st percentile and 1.0 is 99th percentile

```python
cellpose.transforms.normalize_img(img, axis=-1, invert=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`) –
- **invert** (`invert image (useful if cells are dark instead of bright)`) –

Returns

- **img** – normalized image of same size

Return type

`ND-array, float32`

```python
cellpose.transforms.pad_image_ND(img0, div=16, extra=1)
```
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

11.6. Image transforms 45
cellpose.transforms.random_rotate_and_resize(X, Y=None, scale_range=1.0, xy=(224, 224),
do_flip=True, rescale=None, unet=False,
random_per_image=True)

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): param X: list of image arrays of size [nchan x Ly x Lx] or [Ly x Lx] :type X: LIST of ND-arrays, float :param Y: 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.

Parameters

- 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)) –
- random_per_image (bool (optional, default True)) – different random rotate and resize per image

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

cellpose.transforms.reshape(data, channels=[0, 0], chan_first=False)

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].
- invert (bool) – invert intensities

Returns

data

Return type

numpy array that’s (Z x ) Ly x Lx x nchan (if chan_first==False)
cellpose.transforms.reshape_and_normalize_data(train_data, test_data=None, channels=None, normalize=True)

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 \([Ly \times Lx]\), \([nchan \times Ly \times Lx]\), or \([Ly \times Lx \times nchan]\)

• **test_data** *(list of ND-arrays, float (optional, default None))* – list of testing images of size \([Ly \times Lx]\), \([nchan \times Ly \times Lx]\), or \([Ly \times Lx \times nchan]\)

• **channels** *(list of int of length 2 (optional, default None))* – First element of list is the channel to segment \((0=\text{grayscale}, 1=\text{red}, 2=\text{green}, 3=\text{blue})\). Second element of list is the optional nuclear channel \((0=\text{none}, 1=\text{red}, 2=\text{green}, 3=\text{blue})\). 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 \([2 \times Ly \times Lx]\)

• **test_data** *(list of ND-arrays, float (optional, default None))* – list of testing images of size \([2 \times Ly \times Lx]\)

• **run_test** *(bool)* – whether or not test_data was correct size and is useable during training

check sizes and reshape train and test data for training

cellpose.transforms.reshape_train_test(train_data, train_labels, test_data, test_labels, channels, normalize=True)

resize image for computing flows / unresize for computing dynamics

Parameters

• **img0** *(ND-array)* – image of size \([Y \times X \times nchan]\) or \([Lz \times Y \times X \times nchan]\) or \([Lz \times Y \times X]\)

• **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 \([Ly \times Lx \times nchan]\) or \([Lz \times Ly \times Lx \times nchan]\)

Return type

ND-array

cellpose.transforms.unaugment_tiles(y, unet=False)

reverse test-time augmentations for averaging

Parameters
cellpose, Release 0.7.2

- **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*

### 11.7 Plot functions

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

returns pixels of disk with radius *r* and center *med*

cellpose.plot.dx_to_circ(*dP, transparency=False, mask=None*)

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

cellpose.plot.image_to_rgb(*img0, channels=[0, 0]*)

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

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

get patch of size bsize x bsize with most masks

cellpose.plot.mask_overlay(*img, masks, colors=None*)

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*

cellpose.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
cellpose, Release 0.7.2

Returns

RGB – array of masks overlaid on grayscale image

Return type

uint8, 3D array

cellpose.plot.outline_view(img0, maski, color=[1, 0, 0], mode='inner')

Generates a red outline overlay onto image.

cellpose.plot.show_segmentation(fig, img, maski, flowi, channels=[0, 0], file_name=None)

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.eval, where 0=NO masks; 1,2,...=mask labels
- **flowi** (int, 2D array) – for image k, flows[k][0] output from Cellpose.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
- **seg_norm** (bool (optional, default False)) – improve cell visibility under labels

11.8 I/O functions

cellpose.io.add_model(filename)

add model to .cellpose models folder to use with GUI or CLI

cellpose.io.get_image_files(folder, mask_filter, imf=None, look_one_level_down=False)

find all images in a folder and if look_one_level_down all subfolders

cellpose.io.imread(filename)

read in image with tif or image file type supported by cv2

cellpose.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.eval, where 0=NO masks; 1,2,...=mask labels
- **flows** (list of list of ND arrays) – flows output from Cellpose.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

`cellpose.io.remove_model(filename, delete=False)`

remove model from .cellpose custom model list

`cellpose.io.save_masks(images, masks, flows, file_names, png=True, tif=False, channels=[0, 0], suffix='', save_flows=False, save_outlines=False, save_ncolor=False, dir_above=False, in_folders=False, savedir=None, save_txt=True)`

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'

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.eval, where 0=NO masks; 1,2,...=mask labels
- **flows** (*list of list of ND arrays*) – flows output from Cellpose.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,...).

`cellpose.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
cellpose, Release 0.7.2

- **filename** *(str (optional, default None)) – if no GUI, send this file to server*

```
cellpose.io.save_to_png(images, masks, flows, file_names)
```

deprecated (runs io.save_masks with png=True)

does not work for 3D images

## 11.9 Utils functions

### class cellpose.utils.TqdmToLogger(logger, level=None)

Output stream for TQDM which will output to logger module instead of the StdOut.

- **flush**
  
  Flush write buffers, if applicable.
  
  This is not implemented for read-only and non-blocking streams.

- **write(buf)**
  
  Write string to file.
  
  Returns the number of characters written, which is always equal to the length of the string.

### cellpose.utils.circleMask(d0)

Creates array with indices which are the radius of that x,y point inputs:

- **d0** (patch of (-d0,d0+1) over which radius computed)

**outputs:**

- rs: array (2*d0+1,2*d0+1) of radii
- dx, dy: indices of patch

### cellpose.utils.distance_to_boundary(masks)

Get distance to boundary of mask pixels

**Parameters**

- **masks** *(int, 2D or 3D array) – size [Ly x Lx] or [Lz x Ly x Lx]. 0=NO masks; 1,2,...=mask labels*

**Returns**

- **dist_to_bound** – size [Ly x Lx] or [Lz x Ly x Lx]

**Return type**

2D or 3D array

### cellpose.utils.download_url_to_file(url, dst, progress=True)

Download object at the given URL to a local path.

Thanks to torch, slightly modified

**Parameters**

- **url** *(string) – URL of the object to download*
- **dst** *(string) – Full path where object will be saved, e.g. /tmp/temporary_file*
- **progress** *(bool, optional) – whether or not to display a progress bar to stderr Default: True*
cellpose.fill_holes_and_remove_small_masks(masks, min_size=15)
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

might have issues at borders between cells, todo: check and fix

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 15)) – minimum number of pixels per mask, can turn off with -1

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

cellpose.get_mask_perimeters(masks)
get perimeters of masks

cellpose.get_masks_unet(output, cell_threshold=0, boundary_threshold=0)
create masks using cell probability and cell boundary

cellpose.get_perimeter(points)
perimeter of points - npoints x ndim

cellpose.masks_to_edges(masks, threshold=1.0)
get edges of masks as a 0-1 array

Parameters

masks (int, 2D or 3D array) – size [Ly x Lx] or [Lz x Ly x Lx], 0=NO masks; 1,2,...=mask labels

Returns

dges – size [Ly x Lx] or [Lz x Ly x Lx], True pixels are edge pixels

Return type
2D or 3D array

cellpose.masks_to_outlines(masks)
get outlines of masks as a 0-1 array

Parameters

masks (int, 2D or 3D array) – size [Ly x Lx] or [Lz x Ly x Lx], 0=NO masks; 1,2,...=mask labels

Returns

dlines – size [Ly x Lx] or [Lz x Ly x Lx], True pixels are outlines

Return type
2D or 3D array

cellpose.outlines_list(masks)
get outlines of masks as a list to loop over for plotting
cellpose.utils.remove_edge_masks(masks, change_index=True)

remove masks with pixels on edge of image

Parameters

- **masks** *(int, 2D or 3D array)* – size [Ly x Lx] or [Lz x Ly x Lx], 0=NO masks; 1,2,...=mask labels
- **change_index** *(bool (optional, default True))* – if True, after removing masks change indexing so no missing label numbers

Returns

- **outlines** – size [Ly x Lx] or [Lz x Ly x Lx], 0=NO masks; 1,2,...=mask labels

Return type

2D or 3D array

cellpose.utils.stitch3D(masks, stitch_threshold=0.25)

stitch 2D masks into 3D volume with stitch_threshold on IOU

### 11.10 Network classes

class cellpose.resnet_torch.CPnet(nbase, nout, sz, residual_on=True, style_on=True, concatenation=False, mkldnn=False, diam_mean=30.0)

**forward**(data)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cellpose.resnet_torch.batchconvstyle(in_channels, out_channels, style_channels, sz, concatenation=False)

**forward**(style, x, mkldnn=False, y=None)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cellpose.resnet_torch.convdown(in_channels, out_channels, sz)

**forward**(x)

Defines the computation performed at every call.

Should be overridden by all subclasses.
Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cellpose.resnet_torch.convup\((in\_channels, out\_channels, style\_channels, sz, concatenation=False)\)

    forward\((x, y, style, mkldnn=False)\)
    Defines the computation performed at every call.
    Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cellpose.resnet_torch.downsample\((nbase, sz, residual\_on=True)\)

    forward\((x)\)
    Defines the computation performed at every call.
    Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cellpose.resnet_torch.make_style

    forward\((x0)\)
    Defines the computation performed at every call.
    Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cellpose.resnet_torch.resdown\((in\_channels, out\_channels, sz)\)

    forward\((x)\)
    Defines the computation performed at every call.
    Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cellpose.resnet_torch.resup\((in\_channels, out\_channels, style\_channels, sz, concatenation=False)\)
**forward**\((x, y, style, mkldnn=False)\)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class cellpose.resnet_torch.upsample(nbase, sz, residual_on=True, concatenation=False)

**forward**\((style, xd, mkldnn=False)\)

Defines the computation performed at every call.

Should be overridden by all subclasses.

**Note:** Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

### 11.11 Core functions

cellpose.core.use_gpu(gpu_number=0, use_torch=True)

check if gpu works

### 11.12 All models functions

class cellpose.models.Cellpose(gpu=False, model_type='cyto', net_avg=False, device=None)

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; 'cyto2'=cytoplasm model with additional user images*

- **net_avg** *(bool (optional, default False)) – loads the 4 built-in networks and averages them if True, loads one network if False*

- **device** *(torch device (optional, default None)) – device used for model running / training (torch.device('cuda') or torch.device('cpu')), overrides gpu input, recommended if you want to use a specific GPU (e.g. torch.device('cuda:1'))*

**eval**\((x, batch_size=8, channels=None, channel_axis=None, z_axis=None, invert=False, normalize=True, diameter=30.0, do_3D=False, anisotropy=None, net_avg=False, augment=False, tile=True, tile_overlap=0.1, resample=True, interp=True, flow_threshold=0.4, cellprob_threshold=0.0, min_size=15, stitch_threshold=0.0, rescale=None, progress=None, model_loaded=False)\)

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 False)) – 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)

- cellprob_threshold (float (optional, default 0.0)) – all pixels with value above threshold kept for masks, decrease to find more and larger masks

- 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

• **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] = XY flows at each pixel flows[k][2] = cell probability (if > cellprob_threshold, pixel used for dynamics) flows[k][3] = final pixel locations after Euler integration

• **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))**

**class cellpose.models.CellposeModel (gpu=False, pretrained_model=False, model_type=None, net_avg=False, diam_mean=30.0, device=None, residual_on=True, style_on=True, concatenation=False, nchan=2)**

**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))** – full path to pretrained cellpose model(s), if None or False, no model loaded

• **model_type (str (optional, default None))** – any model that is available in the GUI, use name in GUI e.g. ‘livecell’ (can be user-trained or model zoo)

• **net_avg (bool (optional, default False))** – loads the 4 built-in networks and averages them if True, loads one network if False

• **diam_mean (float (optional, default 30.0))** – mean ‘diameter’, 30. is built in value for ‘cyto’ model; 17. is built in value for ‘nuclei’ model; if saved in custom model file (cellpose>=2.0) then it will be loaded automatically and overwrite this value

• **device (torch device (optional, default None))** – device used for model running / training (torch.device(‘cuda’) or torch.device(‘cpu’), overrides gpu input, recommended if you want to use a specific GPU (e.g. torch.device(‘cuda:1’))

• **residual_on (bool (optional, default True))** – use 4 conv blocks with skip connections per layer instead of 2 conv blocks like conventional u-nets

• **style_on (bool (optional, default True))** – use skip connections from style vector to all upsampling layers

• **concatenation (bool (optional, default False))** – if True, concatenate downsampling block outputs with upsampling block inputs; default is to add
**nchan** *(int (optional, default 2)) – number of channels to use as input to network, default is 2 (cyto + nuclei) or (nuclei + zeros)*

**eval** *(x, batch_size=8, channels=None, channel_axis=None, z_axis=None, normalize=True, invert=False, rescale=None, diameter=None, do_3D=False, anisotropy=None, net_avg=False, augment=False, tile=True, tile_overlap=0.1, resample=True, interp=True, flow_threshold=0.4, cellprob_threshold=0.0, compute_masks=True, min_size=15, stitch_threshold=0.0, progress=None, loop_run=False, model_loaded=False)*

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
- **diameter**(float (optional, default None)) – diameter for each image, if diameter is None, set to diam_mean or diam_train if available
- **rescale**(float (optional, default None)) – resize factor for each image, if None, set to 1.0; (only used if diameter is None)
- **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 False)) – 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
cellpose, Release 0.7.2

- **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)

- **cellprob_threshold** (float (optional, default 0.0)) – all pixels with value above threshold kept for masks, decrease to find more and larger masks

- **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

- **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] = XY flow in HSV 0-255 flows[k][1] = XY flows at each pixel flows[k][2] = cell probability (if > cellprob_threshold, pixel used for dynamics) flows[k][3] = final pixel locations after Euler integration

- **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

**train** (train_data, train_labels, train_files=None, test_data=None, test_labels=None, test_files=None, channels=None, 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, nimg_per_epoch=None, rescale=True, min_train_masks=5, model_name=None)

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_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

11.12. All models functions 59
- **test_labels** *(list of arrays (2D or 3D)) – labels for test_data, where 0=no masks; 1,2,...=mask labels; can include flows as additional images*
- **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 224x224 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*
- **model_name** *(str (default, None)) – name of network, otherwise saved with name as params + training start time*

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

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** *(torch device (optional, default None)) – device used for model running / training (torch.device(‘cuda’) or torch.device(‘cpu’)), overrides gpu input, recommended if you want to use a specific GPU (e.g. torch.device(‘cuda:1’))*
- **pretrained_size** *(str) – path to pretrained size model*

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

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

```python
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)
C

cellpose.core, 55
cellpose.dynamics, 41
cellpose.io, 49
cellpose.metrics, 40
cellpose.models, 55
cellpose.plot, 48
cellpose.resnet_torch, 53
cellpose.transforms, 44
cellpose.utils, 51
INDEX

A
add_model() (in module cellpose.io), 49
aggregated_jaccard_index() (in module cellpose.metrics), 40
average_precision() (in module cellpose.metrics), 40
average_tiles() (in module cellpose.transforms), 44

B
batchconvstyle (class in cellpose.resnet_torch), 53
boundary_scores() (in module cellpose.metrics), 40

C
Cellpose (class in cellpose.models), 33, 55
cellpose.core
  module, 55
cellpose.dynamics
  module, 41
cellpose.io
  module, 49
cellpose.metrics
  module, 40
cellpose.models
  module, 55
cellpose.plot
  module, 48
cellpose.resnet_torch
  module, 53
cellpose.transforms
  module, 44
cellpose.utils
  module, 51
CellposeModel (class in cellpose.models), 35, 57
circleMask() (in module cellpose.utils), 51
compute_masks() (in module cellpose.dynamics), 41
downsample (class in cellpose.resnet_torch), 54
dx_to_circ() (in module cellpose.plot), 48
distance_to_boundary() (in module cellpose.utils), 51
download_url_to_file() (in module cellpose.utils), 51

E
eval() (cellpose.models.Cellpose method), 33, 55
eval() (cellpose.models.CellposeModel method), 35, 58
eval() (cellpose.models.SizeModel method), 38, 60

F
fill_holes_and_remove_small_masks() (in module cellpose.utils), 51
flow_error() (in module cellpose.metrics), 40
flush() (cellpose.utils.TqdmToLogger method), 51
follow_flows() (in module cellpose.dynamics), 41
forward() (cellpose.resnet_torch.batchconvstyle method), 53
forward() (cellpose.resnet_torch.convdown method), 53
forward() (cellpose.resnet_torch.convup method), 54
forward() (cellpose.resnet_torch.CPnet method), 53
forward() (cellpose.resnet_torch.downsample method), 54
forward() (cellpose.resnet_torch.make_style method), 54
forward() (cellpose.resnet_torch.resdown method), 54
forward() (cellpose.resnet_torch.resup method), 54
forward() (cellpose.resnet_torch.upsample method), 55

G
get_image_files() (in module cellpose.io), 49
get_mask_perimeters() (in module cellpose.utils), 52
get_masks() (in module cellpose.dynamics), 41
get_masks_unet() (in module cellpose.utils), 52
get_perimeter() (in module cellpose.utils), 52

I
image_to_rgb() (in module cellpose.plot), 48
imread() (in module cellpose.io), 49
interesting_patch() (in module cellpose.plot), 48
L
labels_to_flows() (in module cellpose.dynamics), 42
loss_fn() (cellpose.models.CellposeModel method), 37, 59

M
make_style (class in cellpose.resnet_torch), 54
make_tiles() (in module cellpose.transforms), 44
map_coordinates() (in module cellpose.dynamics), 42
mask_ious() (in module cellpose.metrics), 41
mask_overlay() (in module cellpose.metrics), 48
mask_rgb() (in module cellpose.plot), 42
masks_flows_to_seq() (in module cellpose.io), 49
masks_to_edges() (in module cellpose.dynamics), 52
masks_to_flows() (in module cellpose.dynamics), 42
masks_to_flows_cpu() (in module cellpose.dynamics), 42
masks_to_flows_gpu() (in module cellpose.dynamics), 43
masks_to_outlines() (in module cellpose.dynamics), 52
module
cellpose.core, 55
cellpose.dynamics, 41
cellpose.io, 49
cellpose.metrics, 40
cellpose.models, 55
cellpose.plot, 48
cellpose.resnet_torch, 53
cellpose.transforms, 44
cellpose.utils, 51
move_axis() (in module cellpose.transforms), 45
move_min_dim() (in module cellpose.transforms), 45

N
normalize99() (in module cellpose.transforms), 45
normalize_img() (in module cellpose.transforms), 45

O
outline_view() (in module cellpose.plot), 49
outlines_list() (in module cellpose.utils), 52

P
pad_image_ND() (in module cellpose.transforms), 45

R
random_rotate_and_resize() (in module cellpose.transforms), 45
remove_bad_flow_masks() (in module cellpose.dynamics), 43
remove_edge_masks() (in module cellpose.utils), 52
remove_model() (in module cellpose.io), 50
resdown (class in cellpose.resnet_torch), 54
reshape() (in module cellpose.transforms), 46
reshape_and_normalize_data() (in module cellpose.transforms), 46
reshape_train_test() (in module cellpose.transforms), 47
resize_image() (in module cellpose.transforms), 47
resup (class in cellpose.resnet_torch), 54

S
save_masks() (in module cellpose.io), 50
save_server() (in module cellpose.io), 50
save_to_png() (in module cellpose.io), 51
show_segmentation() (in module cellpose.plot), 49
SizeModel (class in cellpose.models), 38, 60
steps2D() (in module cellpose.dynamics), 43
steps3D() (in module cellpose.dynamics), 43
stitch3D() (in module cellpose.utils), 53

T
TqdmToLogger (class in cellpose.utils), 51
train() (cellpose.models.CellposeModel method), 37, 59
train() (cellpose.models.SizeModel method), 39, 61

U
unaugment_tiles() (in module cellpose.transforms), 47
upsample (class in cellpose.resnet_torch), 55
use_gpu() (in module cellpose.core), 55

W
write() (cellpose.utils.TqdmToLogger method), 51