

# Deep Learning Content-Aware Image Restoration

Anna Sim  
4D CELL Biology Lab

# Background

ARTICLES

<https://doi.org/10.1038/s41592-018-0216-7>

nature methods

## Content-aware image restoration: pushing the limits of fluorescence microscopy

Martin Weigert<sup>1,2\*</sup>, Uwe Schmidt<sup>1,2</sup>, Tobias Boothe<sup>1,2</sup>, Andreas Müller<sup>3,4,5</sup>, Alexandr Dibrov<sup>1,2</sup>, Akanksha Jain<sup>1,2</sup>, Benjamin Wilhelm<sup>1,6</sup>, Deborah Schmidt<sup>1</sup>, Coleman Broaddus<sup>1,2</sup>, Siân Culley<sup>1,7,8</sup>, Mauricio Rocha-Martins<sup>1,2</sup>, Fabián Segovia-Miranda<sup>2</sup>, Caren Norden<sup>1,2</sup>, Ricardo Henriques<sup>1,7,8</sup>, Marino Zerial<sup>2</sup>, Michele Solimena<sup>2,3,4,5</sup>, Jochen Rink<sup>2</sup>, Pavel Tomancak<sup>1,2</sup>, Loïc Royer<sup>1,2,9\*</sup>, Florian Jug<sup>1,2\*</sup> and Eugene W. Myers<sup>1,2,10</sup>

Fluorescence microscopy is a key driver of discoveries in the life sciences, with observable phenomena being limited by the optics of the microscope, the chemistry of the fluorophores, and the maximum photon exposure tolerated by the sample. These limits necessitate trade-offs between imaging speed, spatial resolution, light exposure, and imaging depth. In this work we show how content-aware image restoration based on deep learning extends the range of biological phenomena observable by microscopy. We demonstrate on eight concrete examples how microscopy images can be restored even if 60-fold fewer photons are used during acquisition, how near isotropic resolution can be achieved with up to tenfold under-sampling along the axial direction, and how tubular and granular structures smaller than the diffraction limit can be resolved at 20-times-higher frame rates compared to state-of-the-art methods. All developed image restoration methods are freely available as open source software in Python, FIJI, and KNIME.

Fluorescence microscopy is an indispensable tool in the life sciences for investigating the spatio-temporal dynamics of cells, tissues, and developing organisms. Recent advances such as light-sheet microscopy<sup>1</sup>, structured illumination microscopy<sup>2</sup>, and super-resolution microscopy<sup>3–5</sup> enable time-resolved volumetric imaging of biological processes within cells at high resolution. The quality at which these processes can be faithfully recorded, however, is determined not only by the spatial resolution of the optical device used, but also by the desired temporal resolution, the total duration of an experiment, the required imaging depth, the achievable fluorophore density, bleaching, and photo-toxicity<sup>6,7</sup>. These aspects cannot all be optimized at the same time—trade-offs must be made, for example, by sacrificing signal-to-noise ratio (SNR) by reducing exposure time to gain imaging speed. Such trade-offs are often depicted by a design space that has resolution, speed, light exposure, and imaging depth as its dimensions (Fig. 1a), with the volume being limited by the maximal photon budget compatible with sample health<sup>8,9</sup>.

These trade-offs can be addressed through optimization of the microscopy hardware, yet there are physical limits that cannot easily be overcome. Therefore, computational procedures to improve the quality of acquired microscopy images are becoming increasingly important. Super-resolution microscopy<sup>10–12</sup>, deconvolution<sup>13–15</sup>, surface projection algorithms<sup>16–18</sup>, and denoising methods<sup>19–21</sup> are examples of sophisticated image restoration algorithms that can push the limit of the design space, and thus allow the recovery of important biological information that would be inaccessible by imaging alone. However, most common image restoration problems

have multiple possible solutions, and require additional assumptions to select one solution as the final restoration. These assumptions are typically general, for example, requiring a certain level of smoothness of the restored image, and therefore are not dependent on the specific content of the images to be restored. Intuitively, a method that leverages available knowledge about the data at hand ought to yield superior restoration results.

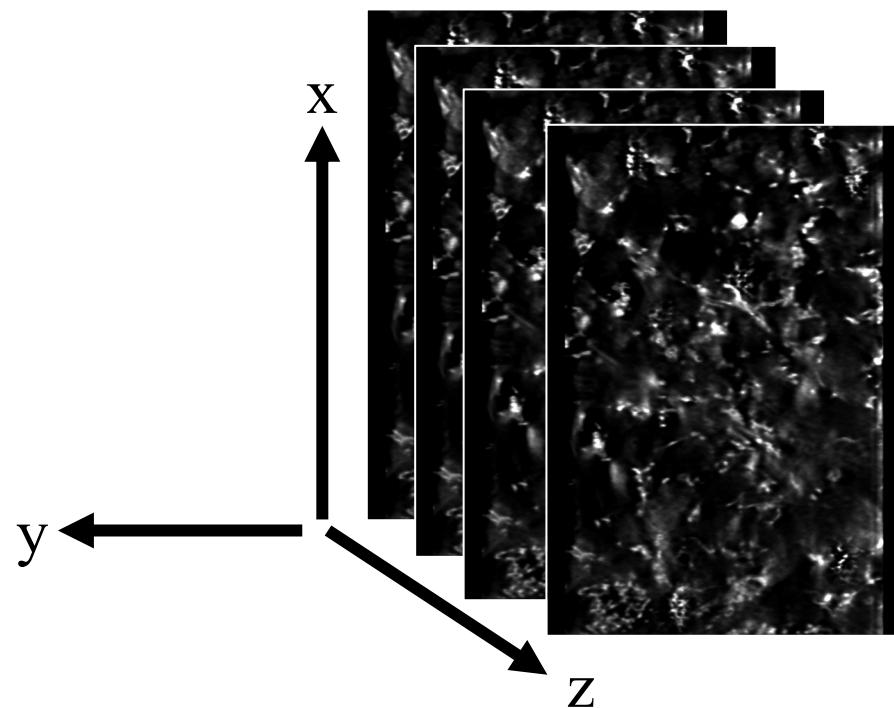
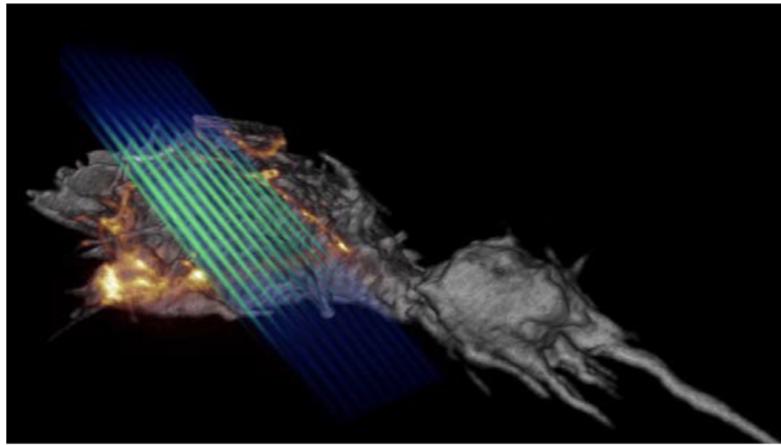
Deep learning is such a method, because it can learn to perform complex tasks on specific data by employing multilayered artificial neural networks trained on a large body of adequately annotated example data<sup>22–26</sup>. In biology, deep learning methods have, for instance, been applied to the automatic extraction of connectomes from large electron microscopy data<sup>27</sup>, for classification of image-based high-content screens<sup>28</sup>, fluorescence signal prediction from label-free images<sup>29,30</sup>, resolution enhancement in histopathology<sup>31</sup>, or for single-molecule localization in super-resolution microscopy<sup>32,33</sup>. However, the direct application of deep learning methods to image restoration tasks in fluorescence microscopy is complicated by the absence of adequate training data and the fact that it is impossible to generate them manually.

We present a solution to the problem of missing training data for deep learning in fluorescence microscopy by developing strategies to generate such data. This enables us to apply common convolutional neural network architectures (U-Nets<sup>34</sup>) to image restoration tasks, such as image denoising, surface projection, recovery of isotropic resolution, and the restoration of sub-diffraction structures. We show, in a variety of imaging scenarios, that trained

\*Center for Systems Biology Dresden, Dresden, Germany. <sup>1</sup>Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany. <sup>2</sup>Molecular Diabetology, University Hospital and Faculty of Medicine Carl Gustav Carus, TU Dresden, Dresden, Germany. <sup>3</sup>Paul Langerhans Institute Dresden of the Helmholtz Center Munich at the University Hospital Carl Gustav Carus and Faculty of Medicine of the TU Dresden, Dresden, Germany. <sup>4</sup>German Center for Diabetes Research (DZD e.V.), Neuherberg, Germany. <sup>5</sup>University of Konstanz, Konstanz, Germany. <sup>6</sup>MRC Laboratory for Molecular Cell Biology, University College London, London, UK. <sup>7</sup>The Francis Crick Institute, London, UK. <sup>8</sup>C2 Bishub, San Francisco, CA, USA. <sup>9</sup>Department of Computer Science, Technical University Dresden, Dresden, Germany. \*e-mail: mweigert@mpi-cbg.de; loic.royer@c2biobhub.org; jug@mpi-cbg.de

- Nature | Methods
- Martin Weigert et al.
- Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany.
- Published: 26 November 2018

# Background



## Spatial Resolution & Temporal Resolution

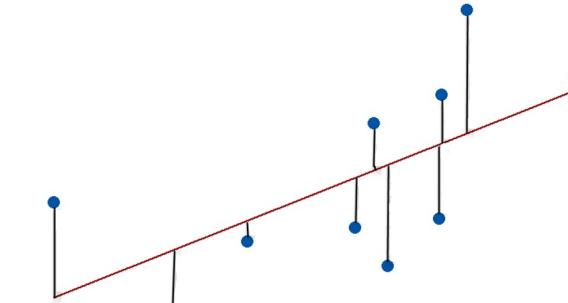
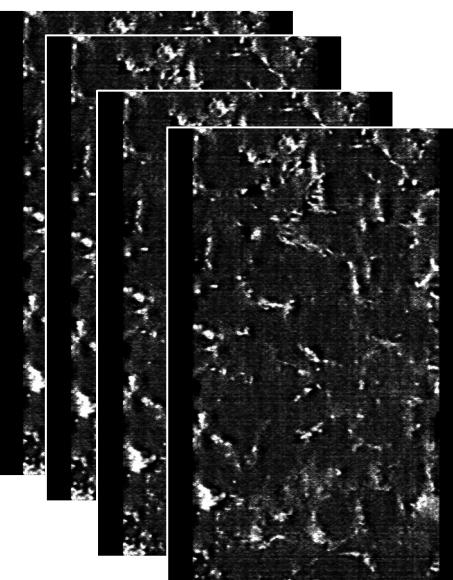
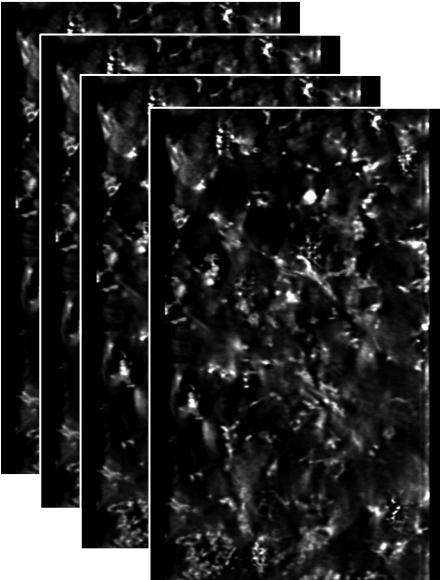
### Goals:

- High spatial resolution
- Prolong time of the experiment
- Required Imaging Depth
- Addressing Bleaching and Toxicity

# Background

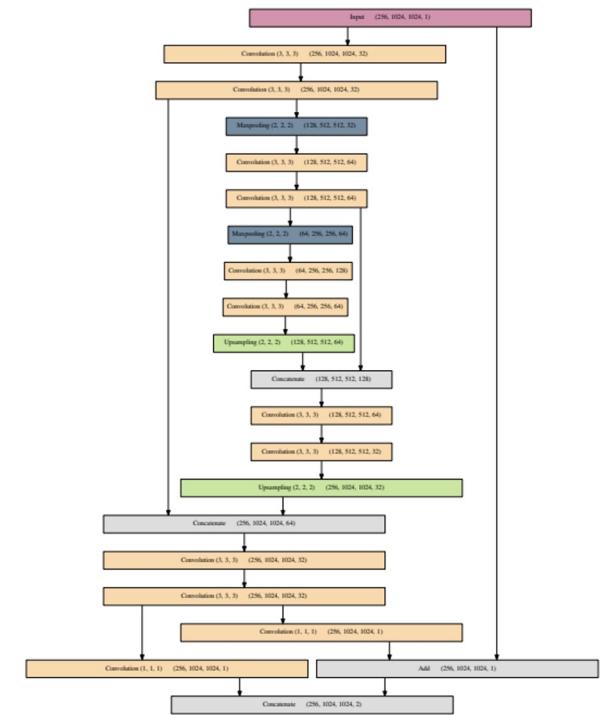
Deep Learning methods are based on:

Large number of Annotated Examples   Mathematical Models   Artificial Neural Network

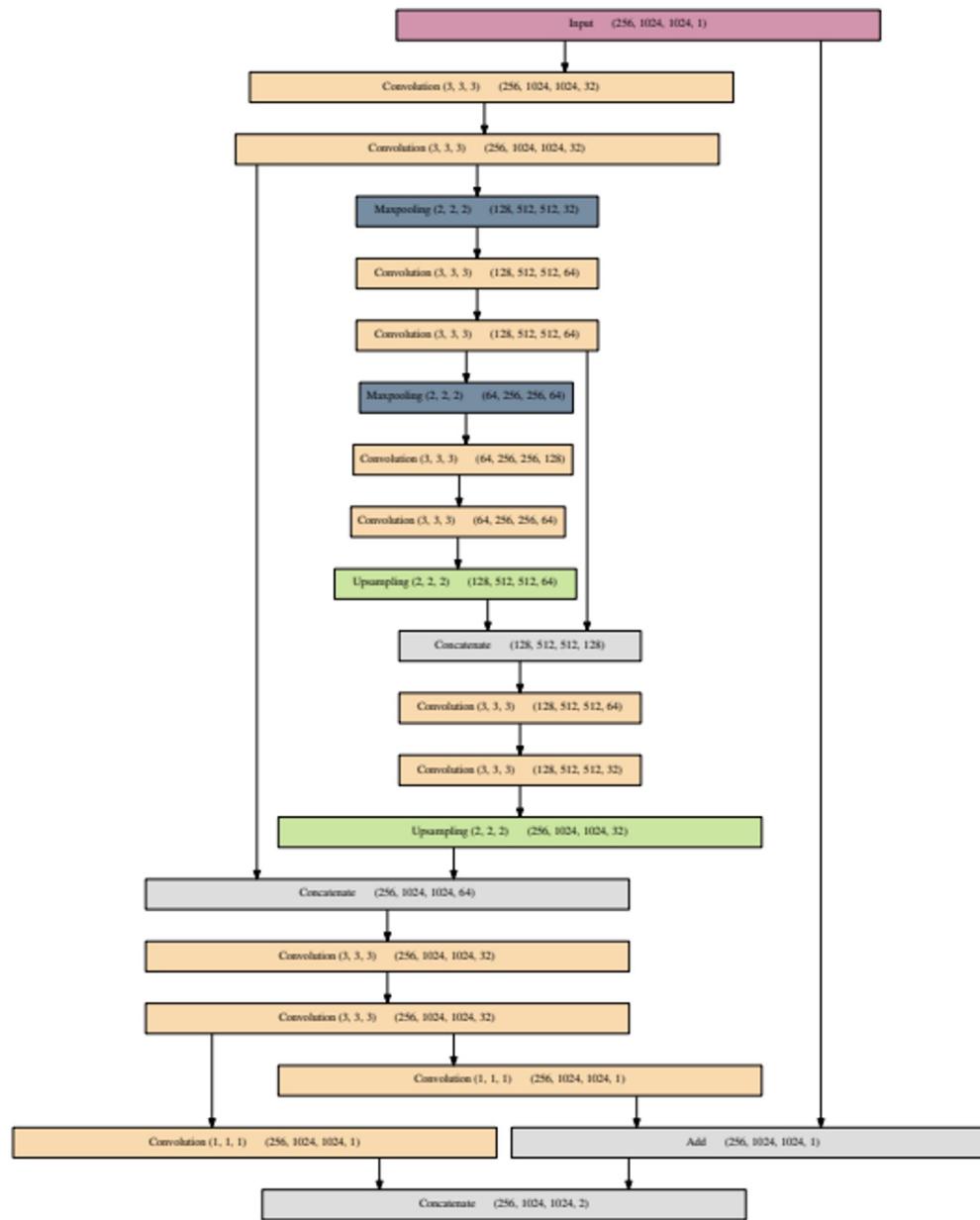


$$\text{MSE}(u, v) = \frac{1}{N} \sum_{i=1}^N (u_i - v_i)^2$$

Supplementary Figures



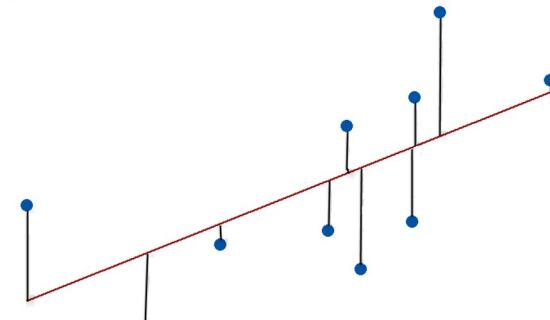
## Supplementary Figures



# Background

Deep Learning methods are based on:

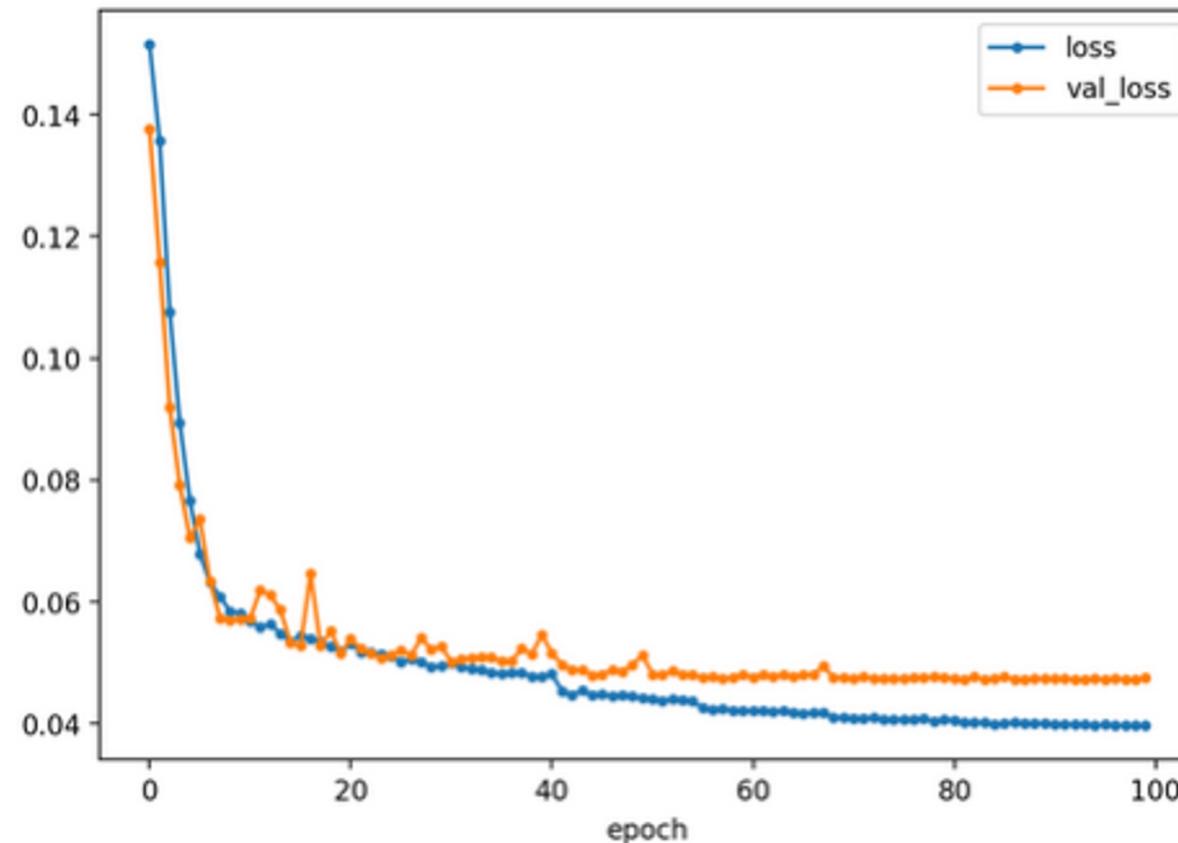
## Mathematical Models



$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (\text{Error} \quad \text{Squared})$$

Mean Error Squared

The equation shows the Mean Squared Error (MSE) calculation. It consists of three parts: 'Mean' (the average), 'Error' (the difference between the observed value  $Y_i$  and the predicted value  $\hat{Y}_i$ ), and 'Squared' (the error squared). The entire formula is enclosed in a large square bracket.



# Three essential parts of any Machine Learning Model

**Demo: Training data generation for denoising of *Tribolium castaneum***

This notebook demonstrates training data generation for a 3D denoising task, where corresponding pairs of low and high-quality stacks can be acquired. Each pair should be registered, which is best achieved by acquiring both stacks interleaved, i.e. as different channels that correspond to the different exposure/laser settings.

We will use a single Tribolium stack pair for training data generation, whereas in your application you should aim to acquire at least 10-50 stacks from different developmental stages to ensure a well-trained model.

More documentation is available at <http://tibebot.readthedocs.com/en/latest/>.

---

Parameters to change:

In [1]:

```
namedByUserFolder = root = 'dataMito'
patchesName = 'training_dataMito'
```

---

In [2]:

```
from future import print_function, unicode_literals, absolute_import, division
import numpy as np
import os
import tifffile
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

from tifffile import imread
from tibebot.util import download_and_extract_zip_file, plot_some
from tibebot.data import RawData, create_patches
```

In [3]:

```
import os
```

We can plot the training stack pair via maximum projection:

In [4]:

```
%load_ext autoreload
%autoreload 2
```

In [6]:

# Data Generation

---

## Generate training data for CARE

We first need to create a `RawData` object, which defines how to get the pairs of low/high-SNR stacks and the semantics of each axis (e.g. which one is considered a color channel, etc.).

Here we have two folders "low" and "GT", where corresponding low and high-SNR stacks are TIFF images with identical filenames. For this case, we can simply use `RawData` and set `axes = 'ZYX'` to indicate the semantic order of the image axes.

In [7]:

```
raw_data = RawData.from_folder(
    basepath = root + '/train',
    source_dirs = ['low'],
    target_dir = 'GT',
    axes = 'ZYX',
)
```

From corresponding stacks, we now generate some 3D patches. As a general rule, use a patch size that is a power of two along XYZ, or at least divisible by 8.

Typically, you should use more patches than the more training stacks you have. By default, patches are sampled from non-background regions (i.e. that are above a relative threshold), see the documentation of `create_patches`, for details.

Note that returned values (`X`, `Y`, `X_low`, `Y_low`) by `create_patches`, are not to be confused with the image axes X and Y. By convention, the variable `name` (`X`, `Y`, `X_low`, `Y_low`) refers to an input variable for a machine learning model, whereas `Y` for `y` indicates an output variable.

In [8]:

```
if not os.path.exists('patches'):
    os.makedirs('patches')
```

In [9]:

```
X, Y, XY_axes = create_patches(
    raw_data,
    patch_size = (16, 64, 64),
    n_patches_per_image = 1000,
    save_file = os.path.join('patches', patchesName + '.npz'),
)
```

In [10]:

```
dataMito.train/lib/w0/p5p2 to 1020_mitodecon.tif dataMito/train/G7/s8/p5p2 to_1020_mitodecon.tif
33% [██████████] | 00:01:00:42, 10.51/11
dataMito/train/G7/s4/p5p2 to_1020_mitodecon.tif dataMito/train/G7/s4/p5p2 to_1020_mitodecon.tif
50% [██████████] | 00:01:00:31, 18.45/11
dataMito/train/low/l1/p5p2 to_1020_mitodecon.tif dataMito/train/G7/l1/p5p2 to_1020_mitodecon.tif
67% [██████████] | 00:01:00:20, 18.43/11
dataMito/train/low/s9/p5p2 to_1020_mitodecon.tif dataMito/train/ct/r7/s9/p5p2 to_1020_mitodecon.tif
83% [██████████] | 00:01:00:10, 18.27/11
dataMito/train/low/s18/p5p2 to_1020_mitodecon.tif dataMito/train/G7/r10/p5p2 to_1020_mitodecon.tif
100% [██████████] | 00:01:00:00, 18.34/11
Saving data to 'patches/training_dataMito.npz'.
```

In [10]:

```
assert X.shape == Y.shape
print("shape of X, Y, X_low, Y_low: ", X.shape)
print("patch size: ", patch_size)
print("n_patches per image: ", n_total_patches)
print("\naxis's: ", axes)
shape of X, Y, X_low, Y_low:  (6144, 1, 16, 64, 64)
patch size:  (16, 64, 64)
n_patches per image:  1000
axis's:  ZCYX
S = n_total_patches
C = n_channels
```

jupyter \_2\_training Lab: Checkpoint: 02/13/2023 (unresolved changes) No timer Trusted Python 3 (ipython)

---

Parameters to change:

```
In [1]: patchesName = "training_dataMito"
modelName = "model_mito"
trainBatchSize = 162
trainEpochs = 32
```

```
In [2]: from future import print_function, unicode_literals, absolute_import, division
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
%config InlineBackend.figure_format = 'retina'

from tensorflow import read
from cdeepseed.utils import axes, dict, plot_som, plot_history
from cdeepseed import CDeepseed
from cdeepseed.io import load_training_data
from cdeepseed import CDeepseed
from cdeepseed.utils import limit_gpu_memory
limit_gpu_memory(None, allow_growth=True)

In [3]: import os

The Tensorflow backend uses all available GPU memory by default, hence it can be useful to limit it.

In [4]: # limit gpu memory(fraction=1/2)

In [5]: import tensorflow as tf
tf.test.is_gpu_available()
```

## Training data

Load training data generated via `_downsample`, use 10% as validation data.

```
In [6]: [Y_val, X_val], axes = load_training_data(os.path.join('patches', patchesName + '.npz'), validation_split=0.1, verbose=True)
axes['C'] = X.shape[1], Y.shape[1]
```

```
In [7]: plt.figure(figsize=(12,2))
plot_som(X_val[:10], Y_val[:10])
plt.suptitle("5 examples")
```

## CARE model

Before we construct the act:

- parameters for the model
- the learning rate
- the number of parameter updates per epoch.
- the loss function,
- whether the model is probabilistic or not.

The defaults should be sensible in many cases, so a change should only be necessary if the training process fails.

**Important:** Note that for this notebook we use a very small number of update steps per epoch for immediate feedback, whereas this number should be increased considerably (e.g. `train_steps_per_epoch=400`) to obtain a well-trained model.

```
In [8]: config = Config(axes, n_channel_in, n_channel_out, train_steps_per_epoch = trainStepsPerEpoch, train_batch_size=trainBatchSize, learning_rate=0.001, loss_fn='mse', probab=False)
print(config)
vars(config)
```

We now create a CARE model with the chosen configuration:

```
In [9]: model = CARE(config, modelName, basedir='models')
```

## Prediction

## Training

Training the model will likely take some time. We recommend to monitor the progress with `TensorBoard` (example below), which allows you to inspect the losses during training. Furthermore, you can look at the predictions for some of the validation images, which can be helpful to recognize problems early on.

You can start TensorBoard from the current working directory with `tensorboard --logdir=...`. Then connect to <http://localhost:6006> with your browser.

```
In [10]: time 1:10
3:45
```

```
In [11]: history = model.train(X,Y, validation_data=(X_val,Y_val))
```

```
In [12]: print(sorted(list(history.history.keys())))
plt.figure(figsize=(16,5))
plot_history(history,['loss','val_loss'],['mse','val_mse','mae','val_mae']);
```

Plot final training history (available in TensorBoard during training):

```
In [13]: print(sorted(list(history.history.keys())))
graph=plt.figure(figsize=(16,5))
plot_history(history,['loss','val_loss'],['mse','val_mse','mae','val_mae']);
graph.figure.savefig('figures.png')
```

## Evaluation

Example results for validation images

```
In [14]: plt.figure(figsize=(12,7))
P = model.predict(X_val[100])
if config.probabilistic:
    P = np.argmax(P, axis=1)[100]
plot_som(X_val[100], Y_val[100], P, pmax=99.5)
plt.suptitle("5 example validation patches")
for i in range(5):
    for j in range(5):
        middle_row = target[j][i]
        top_row = prediction[j][i]
        bottom_row = error[j][i]
        plt.subplot(3, 5, i * 5 + j + 1)
```

## ~~Three~~ Four essential parts of any Machine Learning Model

**Folder Architecture**

**Functionality:**

1. Takes the data from the data folder directory.
2. Creates the folder architecture inside denoising3D.
  - A. Train
  - GT
  - low
- B. Test
- C. GT
- low

3. Randomly distributes the corresponding files to the correct folders 70% train, 30% test
- A. Changes the names of the files (same name for the corresponding files GT - low)

If not satisfied with file distribution best practice is to delete the root folder and re-run the notebook

The distribution is based on randomly generated number

```
This software assumes
membrane channel is 560nm
mito channel is 488nm

Highest laser power percentage 20%


488-20_560-10_her_CamA_ch0_stack0000_488nm_0000000msec_002277875msecAbs_01x_01y_01z_00000
488-20_560-10_ch0_488nm_stack0000_000000msec_decon
488_p0_560_p2_her_560nm_stack0000_000000msec_noDecon

In [13]: import glob
import os
from random import *
import random
import shutil
import re
```

```

Please give your input folder directory

Folder Architecture

In [16]: root = 'nameOfYourFolder'
midfolders = ['train', 'test']
endfolders = ['GT', 'low']

In [17]: # If not os.path.exists(root):
#     for midfolder in midfolders:
#         for endfolder in endfolders:
#             os.makedirs(os.path.join(root, midFolder, endFolder))

In [18]: #for dirs in glob.glob(, recursive = True):
#       if a == 65:
#           for files in glob.glob(currentdir, recursive = True):
#               if re.search('20_([0-9]{3})_files_and_re_search('500mb')'):
#                   shutil.copy(files, os.path.join(root, 'test/GT', 's' + sample + 'p5p2_to_1020_mitodecon.tif'))
#                   print(os.path.join(root, 'test/GT', 's' + sample + 'p5p2_to_1020_mitodecon.tif'))
#               elif re.search('20_([0-9]{3})_files_and_re_search('500mb')'):
#                   shutil.copy(files, os.path.join(root, 'test/low', 's' + sample + 'p5p2_to_1020_mitodecon.tif'))
#                   print(os.path.join(root, 'test/low', 's' + sample + 'p5p2_to_1020_mitodecon.tif'))
#               elif re.search('20_([0-9]{3})_files_and_re_search('500mb')'):
#                   shutil.copy(files, os.path.join(root, 'test/tif', 's' + sample + 'p5p2_to_1020_mitodecon.tif'))
#                   print(os.path.join(root, 'test/tif', 's' + sample + 'p5p2_to_1020_mitodecon.tif'))
#               else:
#                   print("No matching files")
#
#       print("Training Completed")

In [19]: 0.173831646745230527
4
/home/annasri/CARE/Mitodecon/Data/4
dataMtr07/train/lw/s4_p5p2_to_1020_mitodecon.tif
dataMtr07/train/tif/s7_p5p2_to_1020_mitodecon.tif
No matching files
```

Jupyter \_dtagen Last Checkpoint: 02/13/2023 (autosaved) Trusted | Python 3 (pyenv)

File Edit View Insert Cell Kernel Widgets Help

Demo: Training data generation for denoising of *Tribolium castaneum*

This notebook demonstrates training data generation for a 2D denoising task, where corresponding pairs of low and high quality stacks can be acquired. Each stack should be registered, which is best achieved by acquiring both stacks interleaved, i.e. as different channels that correspond to the different exposure settings.

We will use a single Tribolium stack pair for training data generation, whereas in your application you should aim to acquire at least 10-50 stacks from different developmental timepoints to ensure a well trained model.

More documentation is available at [http://tribolium\\_homing\\_phenotype.readthedocs.io](http://tribolium_homing_phenotype.readthedocs.io)

---

Parameters to change:

In [1]:

```
nanoflyYourFolder = root = "dataMito"  
patchesName = "training_dataMito"
```

```
In [2]: from __future__ import print_function, unicode_literals, absolute_import, division
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

from tensorflow import nnadad
from csbdeep.utils import download_and_extract_zip_file, plot_some
from csbdeep.data import RawData, create_patches

In [3]: %load_ext autoreload
%autoreload 2

We can plot the training stack via maxmin projection:

In [4]: %load_ext autoreload
%autoreload 2

In [6]: # y = imread('/home/annassis/CARE/CSBDeep_duplicate/examples/denoising3D/data/l/train/s1/p5p2_to_1620.nrrdcon.tif'
# x = imread('/home/annassis/CARE/CSBDeep_duplicate/examples/denoising3D/data/l/train/s1/p5p2_to_1620.nrrdcon.tif'
# print('Image size %s, %s' % y.shape)
```

# Data Generation

jupyter 2\_training Last Checkpoint: 02/13/2023 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

No kernel Trusted Python 3 (ipython)

---

Parameters to change:

```
In [1]: batchSize = 100000
dataDir = 'train'
dataName = 'model_aitec'
trainStepsPerEpoch = 192
trainEpochs = 10
```

```
In [2]: from future import print_function, unicode_literals, absolute_import, division
import numpy as np
import tensorflow as tf
import tensorflow.python.platform as pl
pl.logging.set_verbosity(pl.logging.INFO)
#config = tf.ConfigProto()
#config.IntraOpParallelismThreads = 1
#config.format = 'retina'

from tensorflow import keras
from tensorflow.keras import layers, models, optimizers, losses, metrics, datasets, utils
from c3ddeep.utils import import_limit_gpu_memory
from c3ddeep import C3DModel
from c3ddeep import Config
from c3ddeep.utils import import_limit_gpu_memory
utils.give_memory_usage_allow_growth=True
```

```
In [3]: import os
```

```
In [1]: # Import GPU memory if it's available
In [1]: import tensorflow as tf
        tf.test.is_gpu_available()



## Training data


Load training data generated via downconv, use 10% as validation data.

In [1]: (X_train, axes) = load_training_data(os.path.join('patches', patchesName + '.npz'), validation_split=0.1, verbose=True)
        n_taxes = len(taxes)
        n_channel = int(X.shape[1], Y.shape[1])

In [1]: plt.figure(figsize=(12, 2))
        plot_some(X_val[:10], Y_val[:10])
        plt.suptitle('5 example validation patches (top row: source, bottom row: target)');
```

Before we construct the algorithm:

- parameters of the unit
- the learning rate
- the number of patches
- the batch size
- whether the model is trained

The defaults should be set

**Important:** Note that for this notebook we use a very small number of update steps per epoch for immediate feedback, whereas this number should be increased considerably (e.g. `train_steps_per_epoch=400`) to obtain a well-trained model.

```
In [1]: config = Config(axes, n_channel_in, n_channel_out, train_steps_per_epoch = trainStepsPerEpoch, train_batch_size = batchSize, validation=config[1])  
print(config)
```

We now create a CARE model with the chosen configuration:

```
In [2]: model = CARE(config, modelName, basedir="models")
```

## Training

Training the model will likely take some time. We recommend to monitor the progress with [TensorBoard](#) (example below), which allows you to inspect the losses during training. Furthermore, you can look at the predictions for some of the validation images, which can be helpful to recognize problems early on.

You can start TensorBoard from the current working directory with `tensorboard --logdir=.`. Then connect to <http://localhost:6006> with your browser.

```
time 1:10  
3:45
```

```
In [3]: history = model.train(X,Y, validation_data=(X_val,Y_val))  
  
In [4]: print(sorted(list(history.history.keys())))
plt.figure(figsize=(16,5))
plot_history(history, ['loss','val_loss'], ['mse','val_mse','mae','val_mae']);
```

Plot first training history (available in TensorBoard during training):

```
In [5]: print(sorted(list(history.history.keys())))
graph, plt = figure(figsize=(16,5))
plot_history(history, ['loss','val_loss'], ['mse','val_mse','mae','val_mae']);
gdrive_saver.save(gdrive, "train_hist.png");
```

---

## Evaluation

Example results for validation images.

```
In [6]: plt.figure(figsize=(12,7))
if model.keras_model.predict(X_val[:10]):
    for i in range(10):
        P = np.zeros((P.shape[1]-1)/2)
        plot_P = P[1::(P.shape[1]-1)/2]
        plot_P[0] = 1
        plot_P[-1] = 1
        plt.imshow(np.concatenate([X_val[i], P], axis=1), interpolation='nearest')
        plt.title('3 example validation patches\\n' + 'top row: input source\\n' + 'middle row: target ground truth,\\n' + 'bottom row: predicted from source');

```

jupyter \_training Last Checkpoint: 02/13/2023 (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Python 3 (systematic)

---

Parameters to change:

```
In [1]: patchName = "training_dataMito"
modelName = "model_mito"
trainEpochs = 100
trainBatchSize = 32
```

```
In [2]: future = import print_function, unicode_literals, absolute_import, division
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
%config InlineBackend.figure_format = 'retina'

from tensorflow import rint
from tensorflow import keras, layers, plot_hist, plot_some, plot_history
from cibber.utils.tf import limit_gpu_memory
from cibber import load_data, save_data
from cibber.models import Config, Caps
from cibber.utils.tf import limit_gpu_memory
limit_gpu_memory(0, AllowGrowth=True)

In [3]: import os
```

```
In [1]: # init gpu memory if fraction < 1/2
In [1]: import tensorflow as tf
        tf.test.is_gpu_available()
```

---

## Training data

Load training data generated via [downsample](#), use 10% as validation data.

```
In [1]: X, Y_val, axes = load_training_data(os.path.join('patches', patchesName + '.npz'), validation_split=0.1, verbose=True)
      , timesteps[1]
      , X_channel_out = X.shape[c], Y.shape[c]
```

```
In [1]: plt.figure(figsize=(12,12))
        plt.imshow(X_val[10], Y_val[10])
        plt.suptitle('5 example validation patches (top row: source, bottom row: target)');
```

## CARE model

Before we construct the actual CARE parameters of the underlying net the learning rate, the momentum, the learning rate update the loss function, and whether the model is probabilistic.

The defaults should be sensible in many cases, so a change should only be necessary if the training process fails.

**Important:** Note that in this notebook we use a very small number of update steps per epoch for immediate feedback, whereas this number should be increased considerably (e.g. `train_steps_per_epoch=400`) to obtain a well-trained model.

```
In [1]: config = Config(axes, n_channel_in, n_channel_out, train_steps_per_epoch = trainStepsPerEpoch, train_batch_size=trainBatchSize)
```

We now create a CARE model with the chosen configuration:

```
In [1]: model = CARE(config, modelname, basedir=r'models')
```

## Training

Training the model will likely take some time. We recommend to monitor the progress with `TensorBoard` (example below), which allows you to inspect the losses during training. Furthermore, you can look at the predictions for some of the validation images, which can be helpful to recognize problems early on.

You can start TensorBoard from the current working directory with `tensorboard --logdir=.`. Then connect to <http://localhost:6006> with your browser.

```
time 1:10
```

```
3:45
```

```
In [1]: history = model.train(X,Y, validation_data=(X_val,Y_val))
```

```
In [1]: print(sorted(list(history.history.keys())))
print('plotting training and validation losses');
plot_history(history,['loss','val_loss'],['mse','val_mse','mae','val_mae']);
```

Plot final training loss (available in TensorBoard during training):

```
In [1]: print(sorted(list(history.history.keys())))
graphics=plt.figure(figsize=(16,5))
plt.plot(history.history['loss'],label='train loss',linestyle='solid')
plt.plot(history.history['val_loss'],label='val loss',linestyle='dashed')
graphics.savefig('figures.png');
```

## Evaluation

Example results for validation images.

```
In [1]: plt.figure(figsize=(12,7))
P = model.keras_model.predict(X_val[10])
# crop to original size
P = P[...,(P.shape[-1]/2):(P.shape[-1]),...]
plot_mosaic(P, nrow=5, ncol=2, title='5 example validation patches\\n'
            'top row: input image\\n'
            'middle row: target (ground truth);\\n'
            'bottom row: predicted from source');
```

# Results

Three models trained:

PreProcessed Data



Denoised Data

Raw Data



PreProcessed and Denoise Data

Raw Mito Data



PreProcessed and Denoise Data

# Results

## Full pipeline

Iterate:

- Date
  - Sample
    - Location
    - Channel
    - Frame



background subtraction



bleach correction

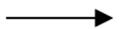


**Zachary  
Wang**

deconvolution



deskew



rotation



processed data

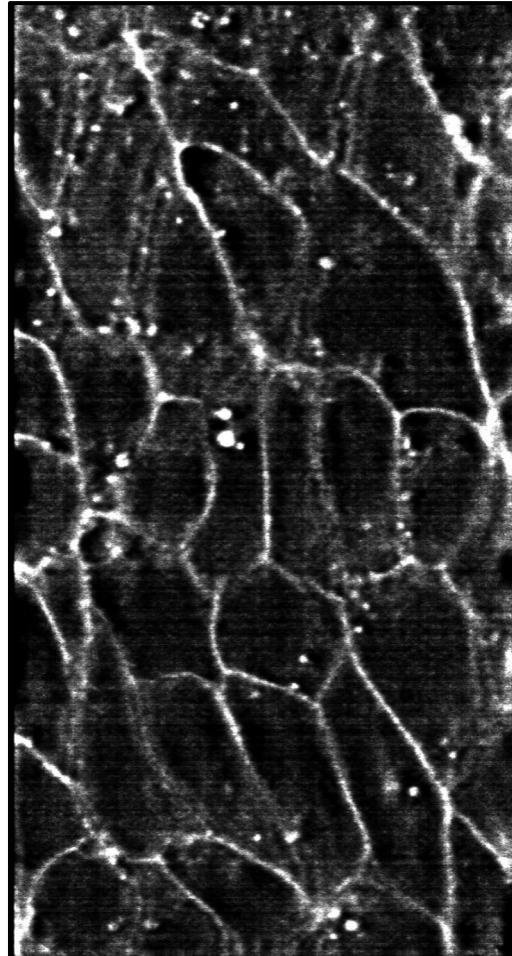


Save the data under a different folder with same file structure

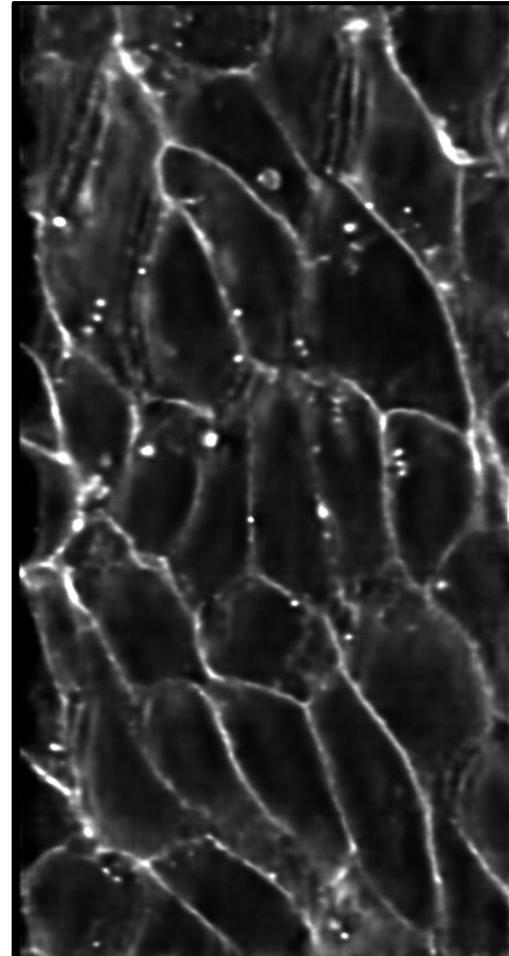
# Model\_1.0 PreProcessed Data – Denoised Data

sample 10

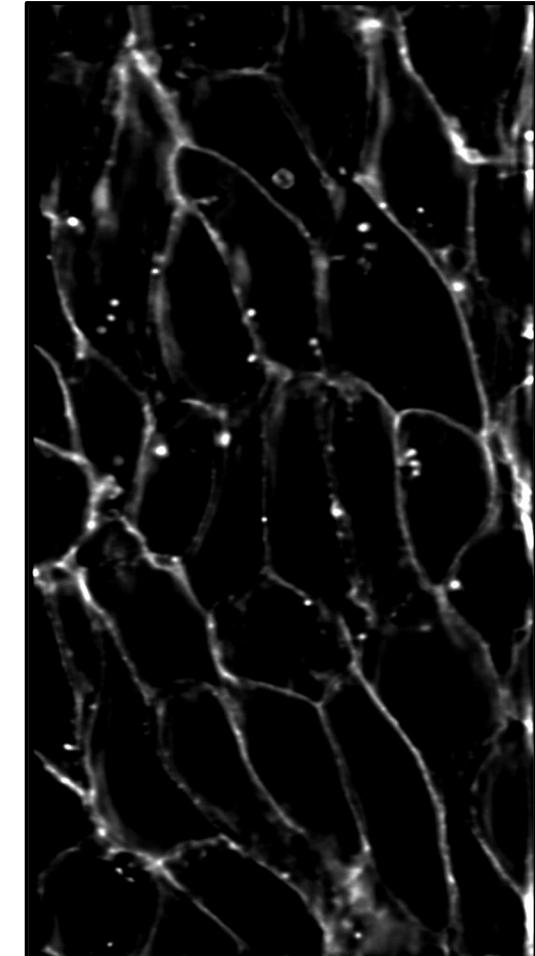
Membrane



Low – Laser Power (0.5 % | 0.2%)

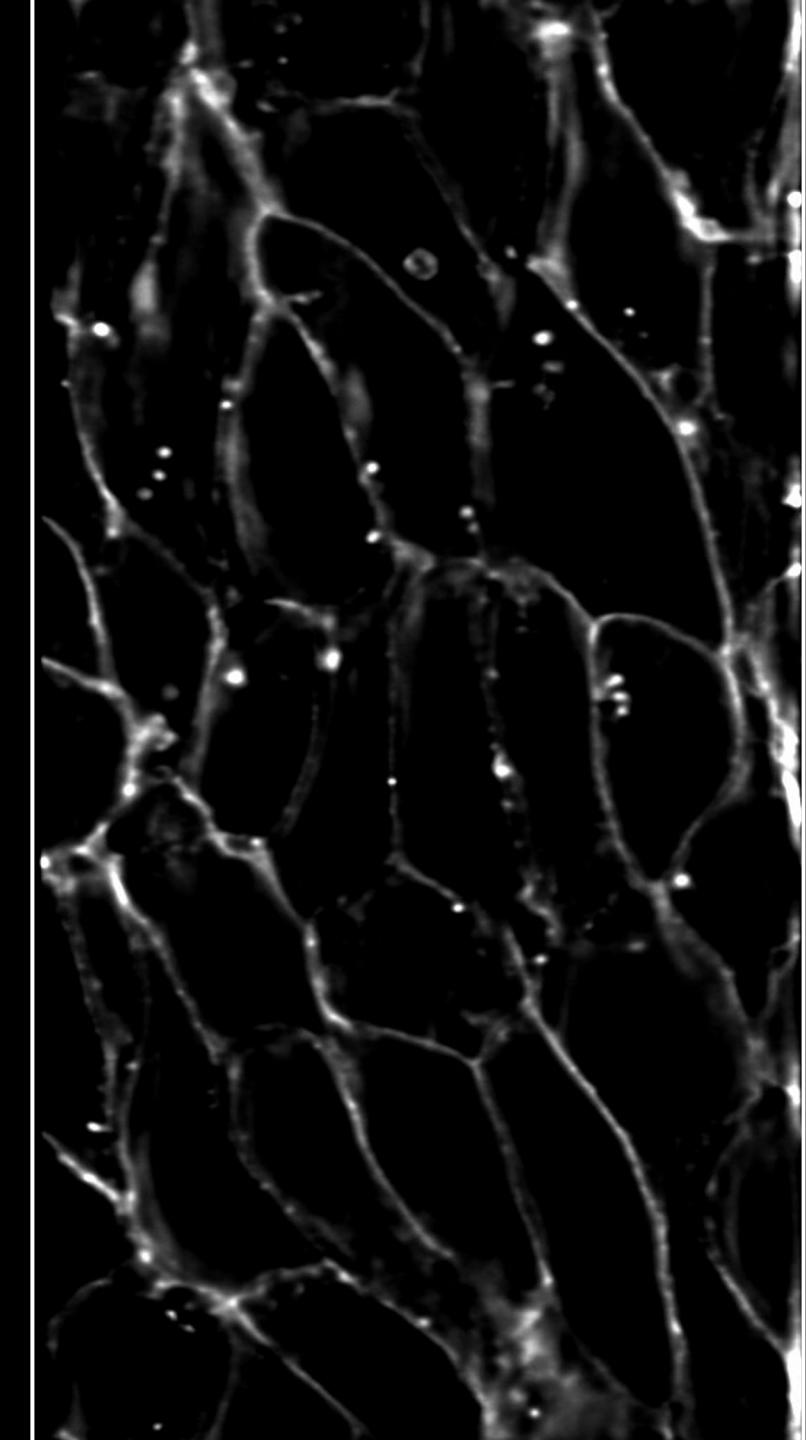
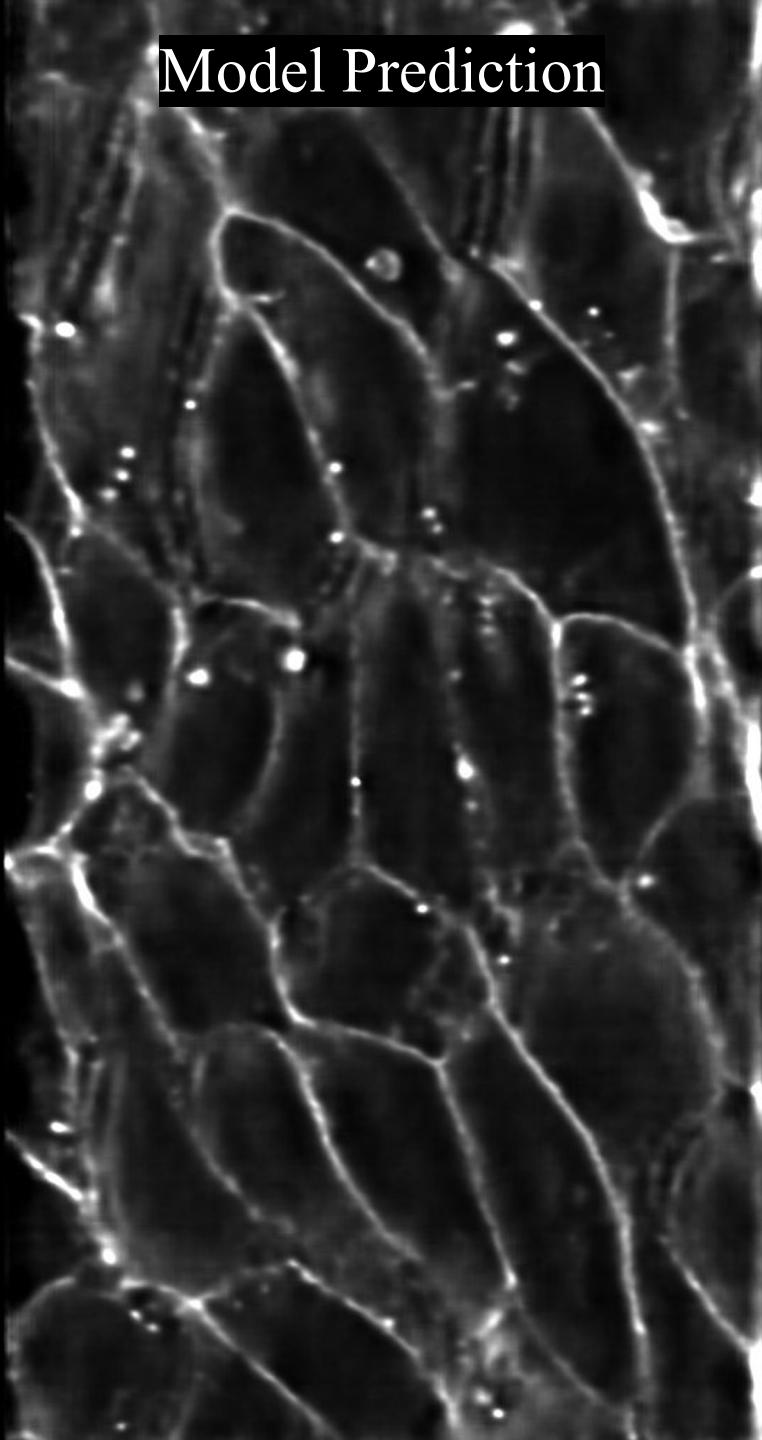
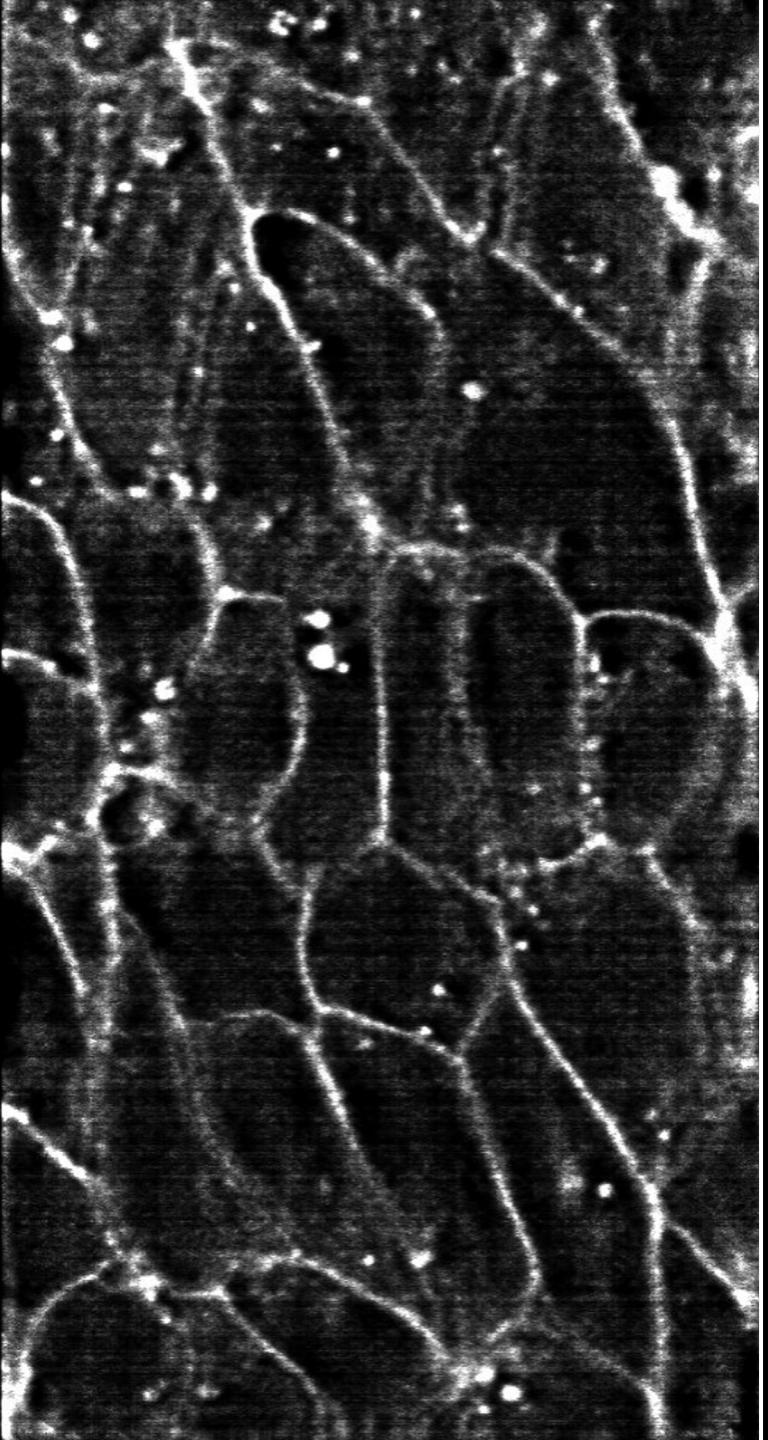


Prediction



GT – Laser Power (20% | 10%)

Model Prediction



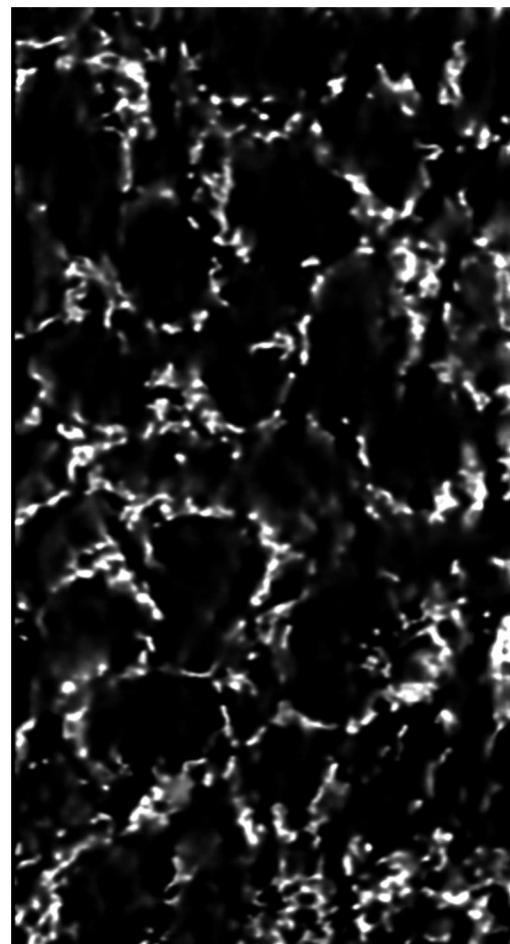
# Model\_1.0 PreProcessed Data – Denoised Data

sample 6

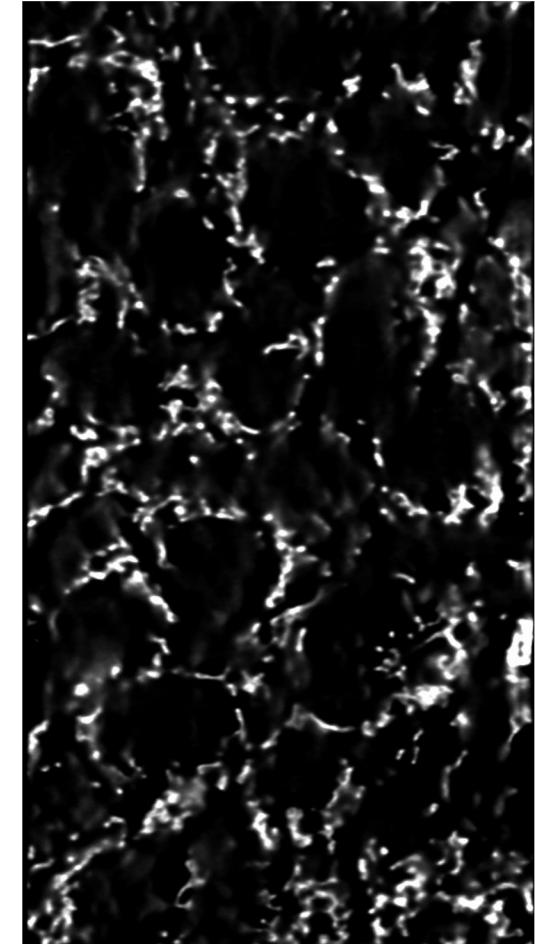
Mitochondria



Low – Laser Power (0.5 | 0.2)

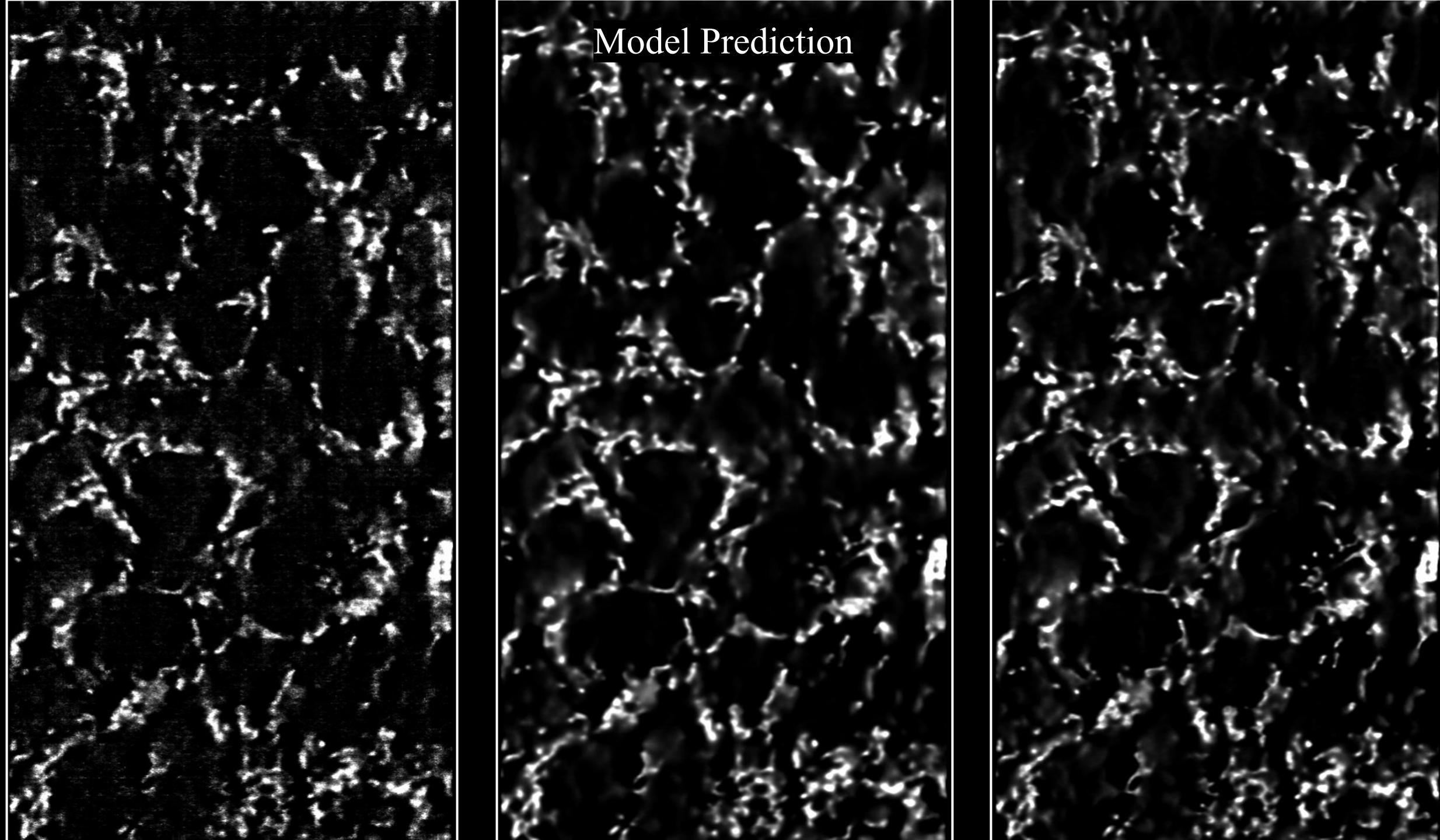


Prediction



GT – Laser Power (20% | 10%)

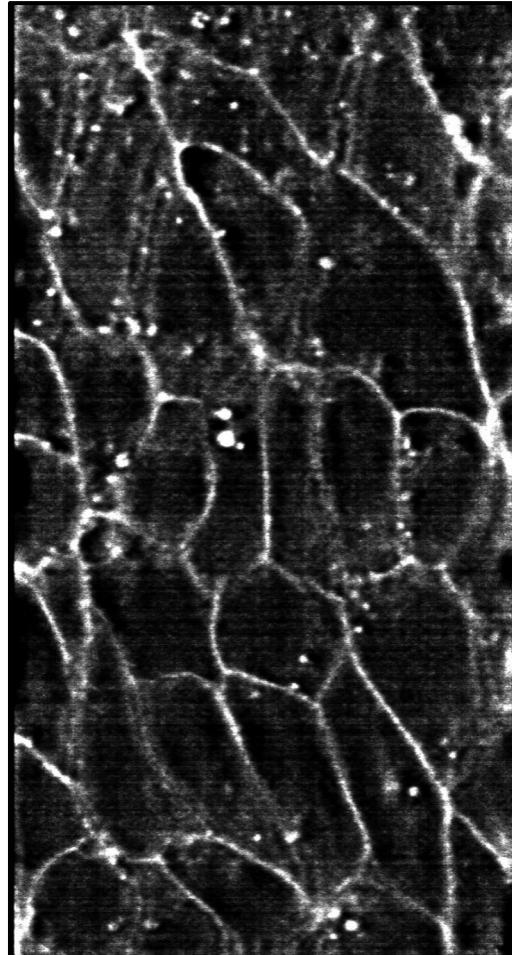
Model Prediction



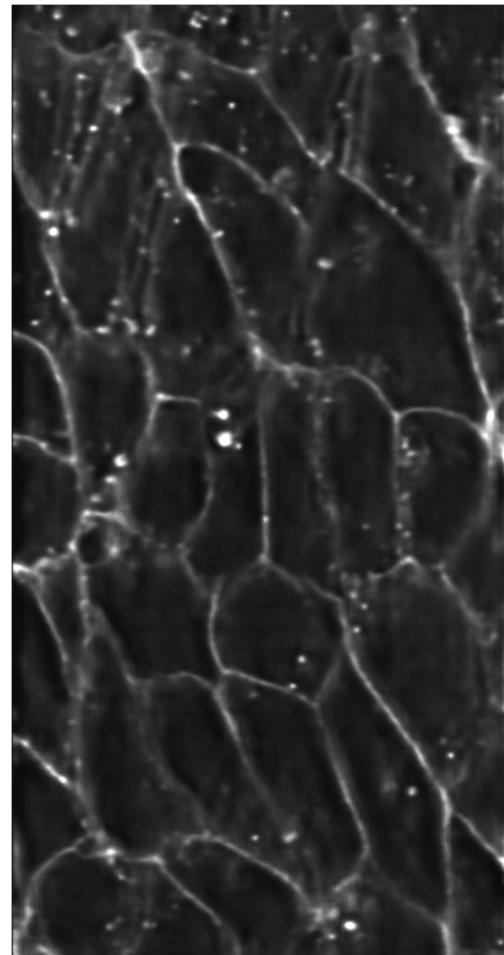
# Model\_1.1 Raw Data – Denoised Data

sample 10

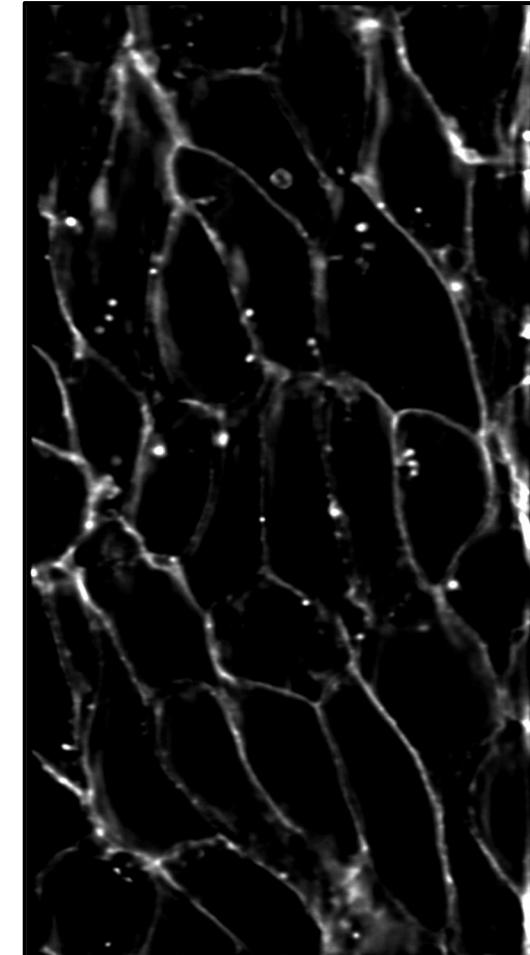
Membrane



Low – Laser Power (0.5 | 0.2)

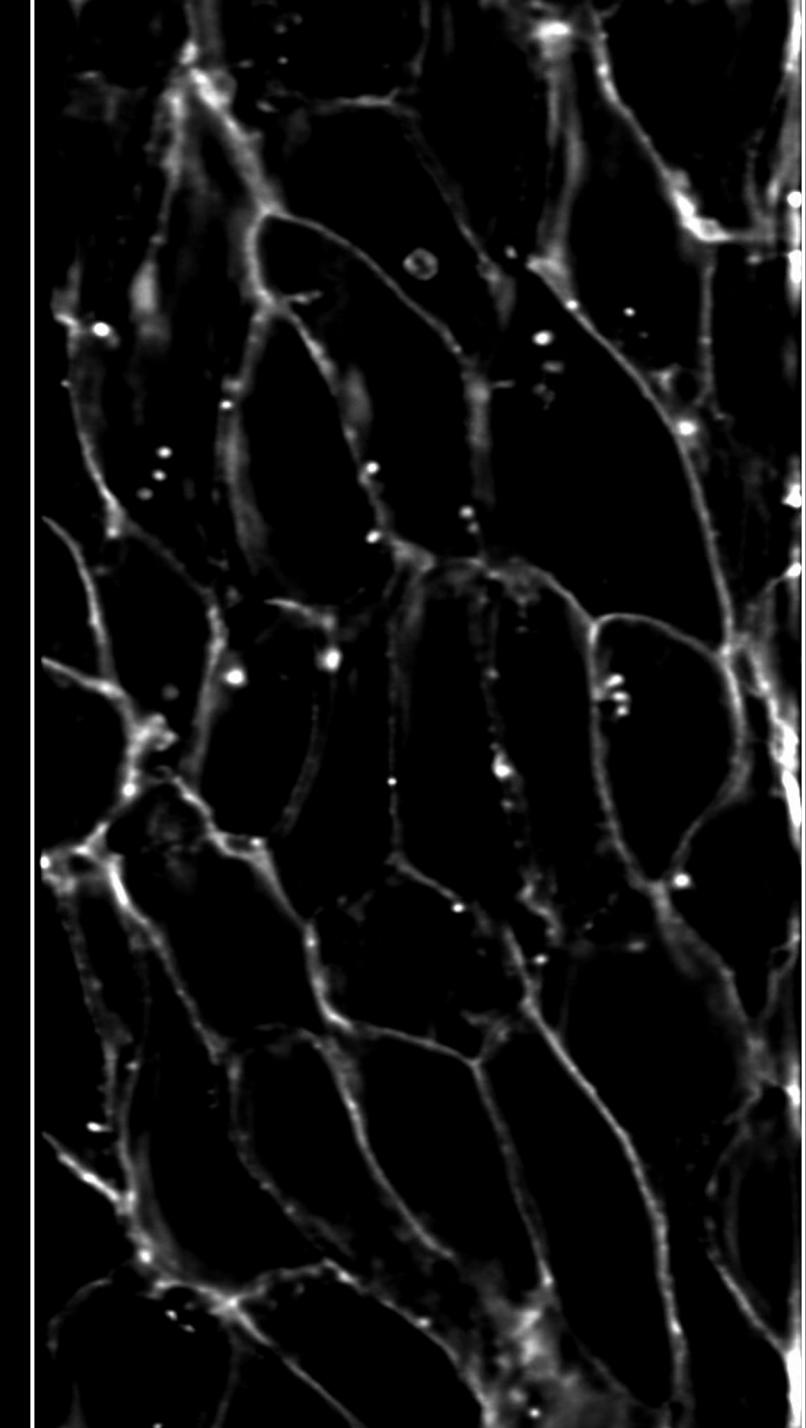
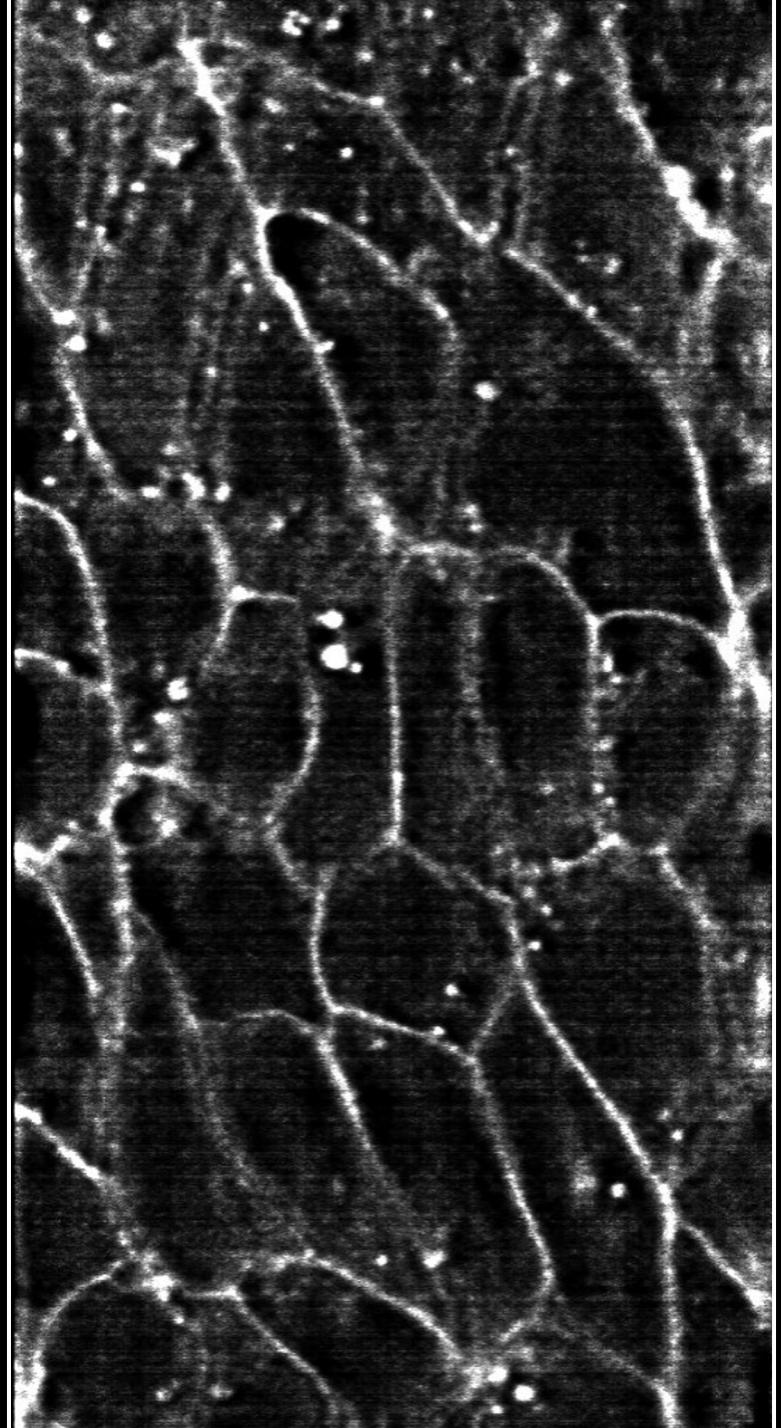
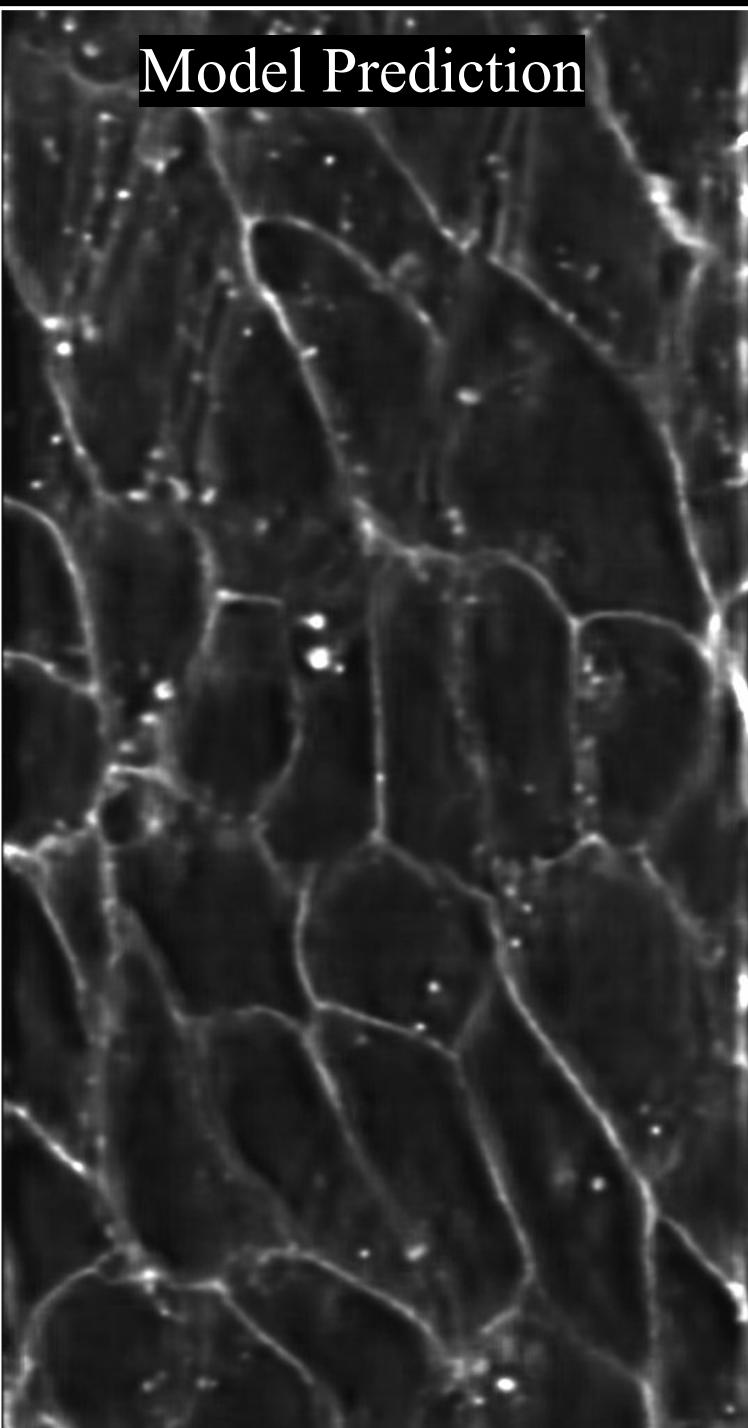


Prediction



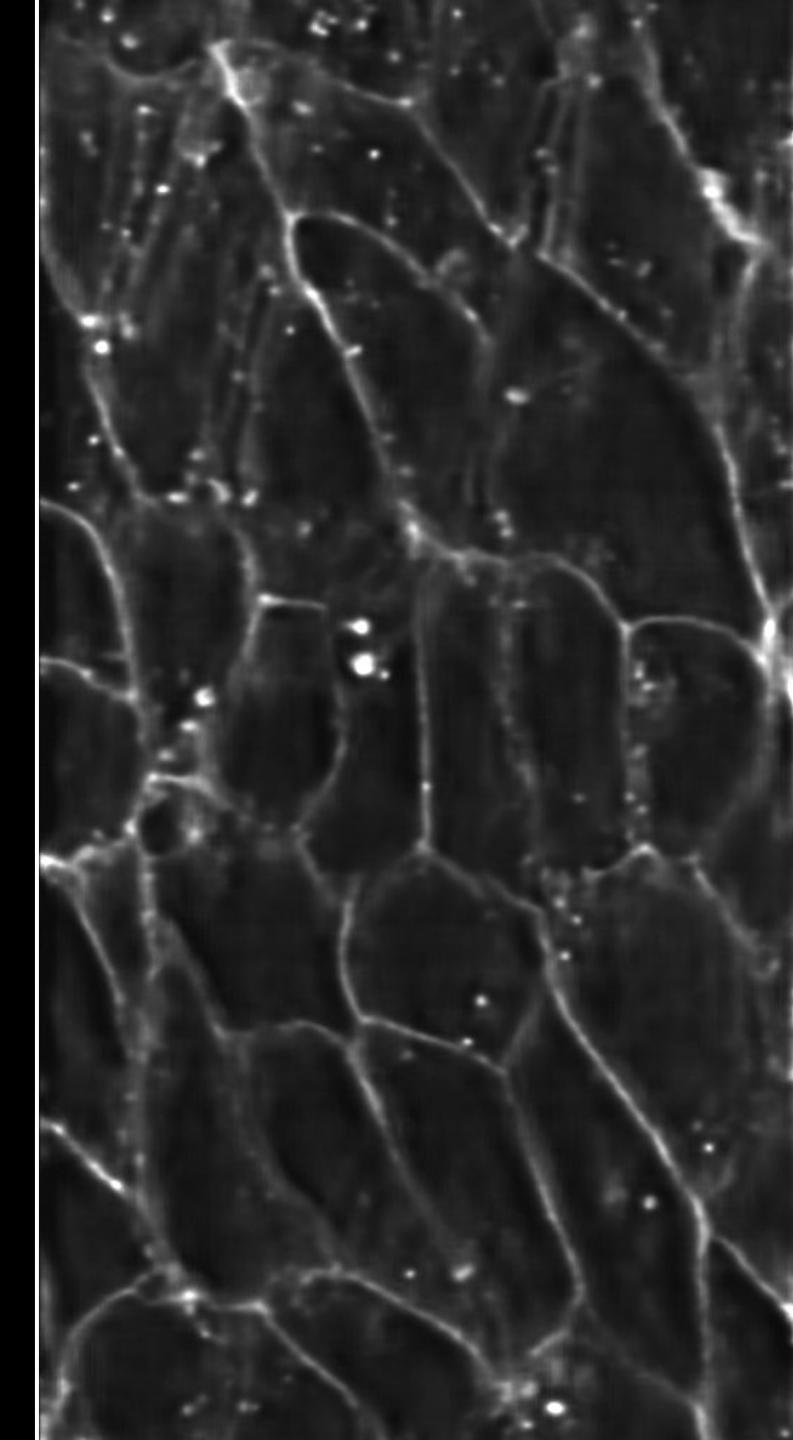
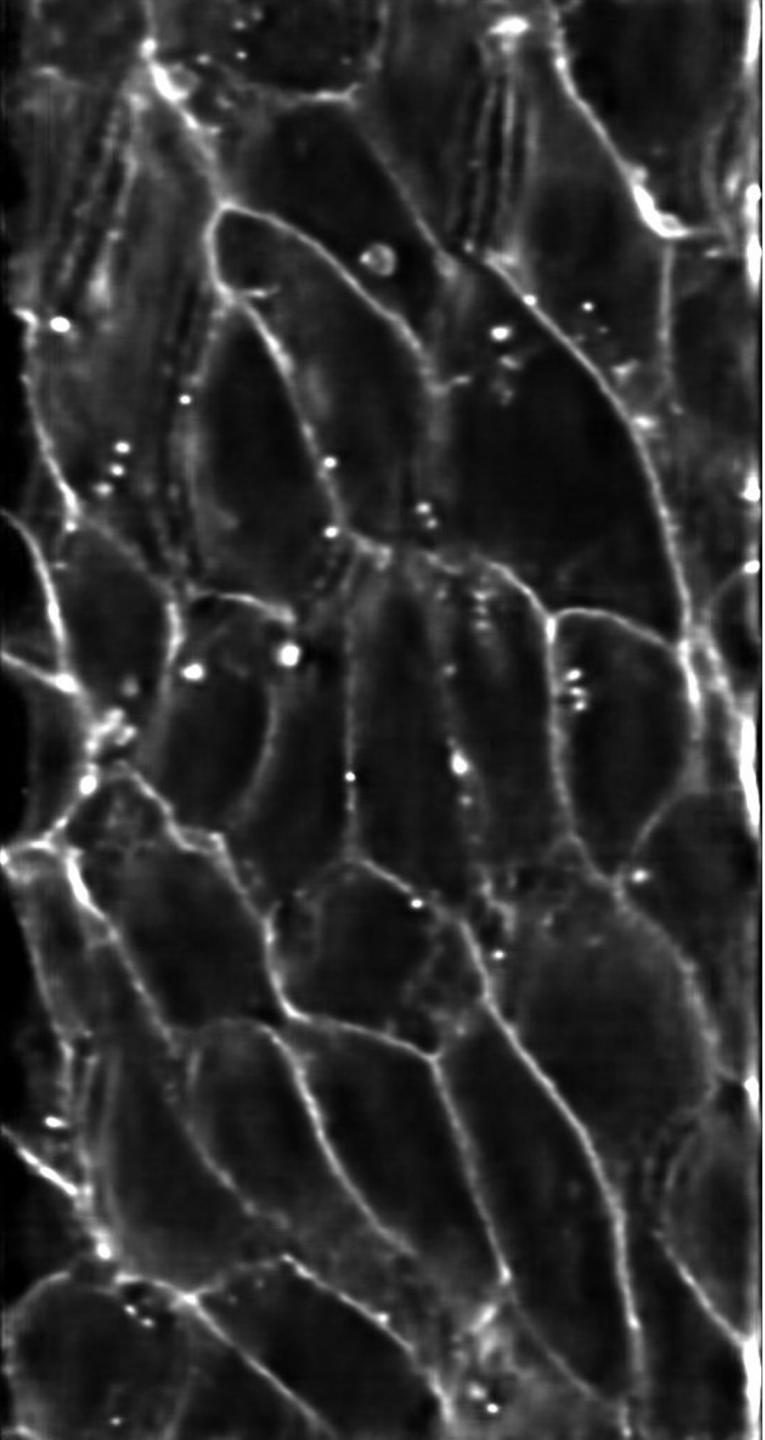
GT – Laser Power (20% | 10%)

Model Prediction



Sample 10

Model 1.0

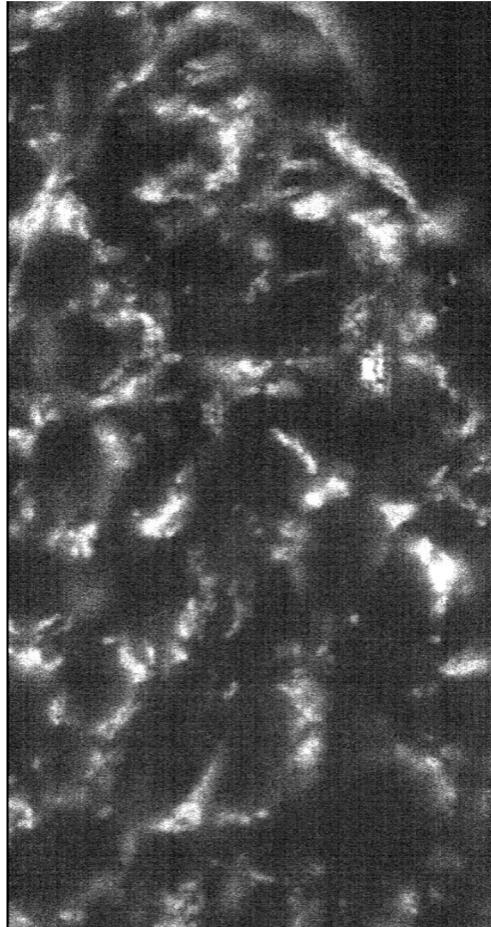


Model 1.1

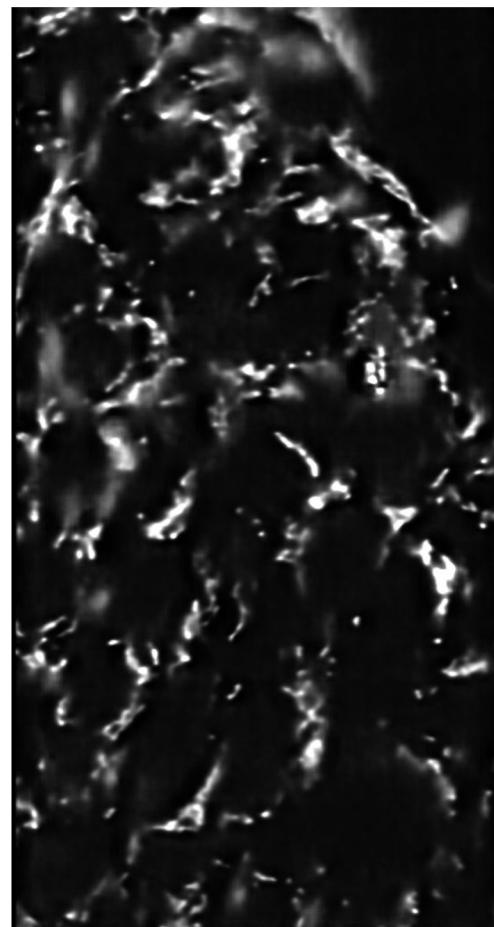
# Model\_1.1 Raw Data – Denoised Data

sample 8

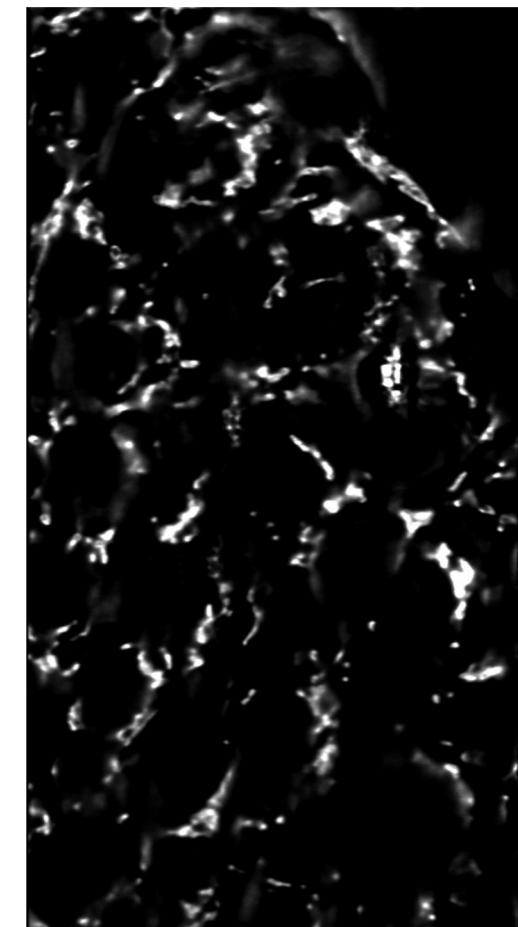
Mitochondria



Low – Laser Power (0.5 | 0.2)

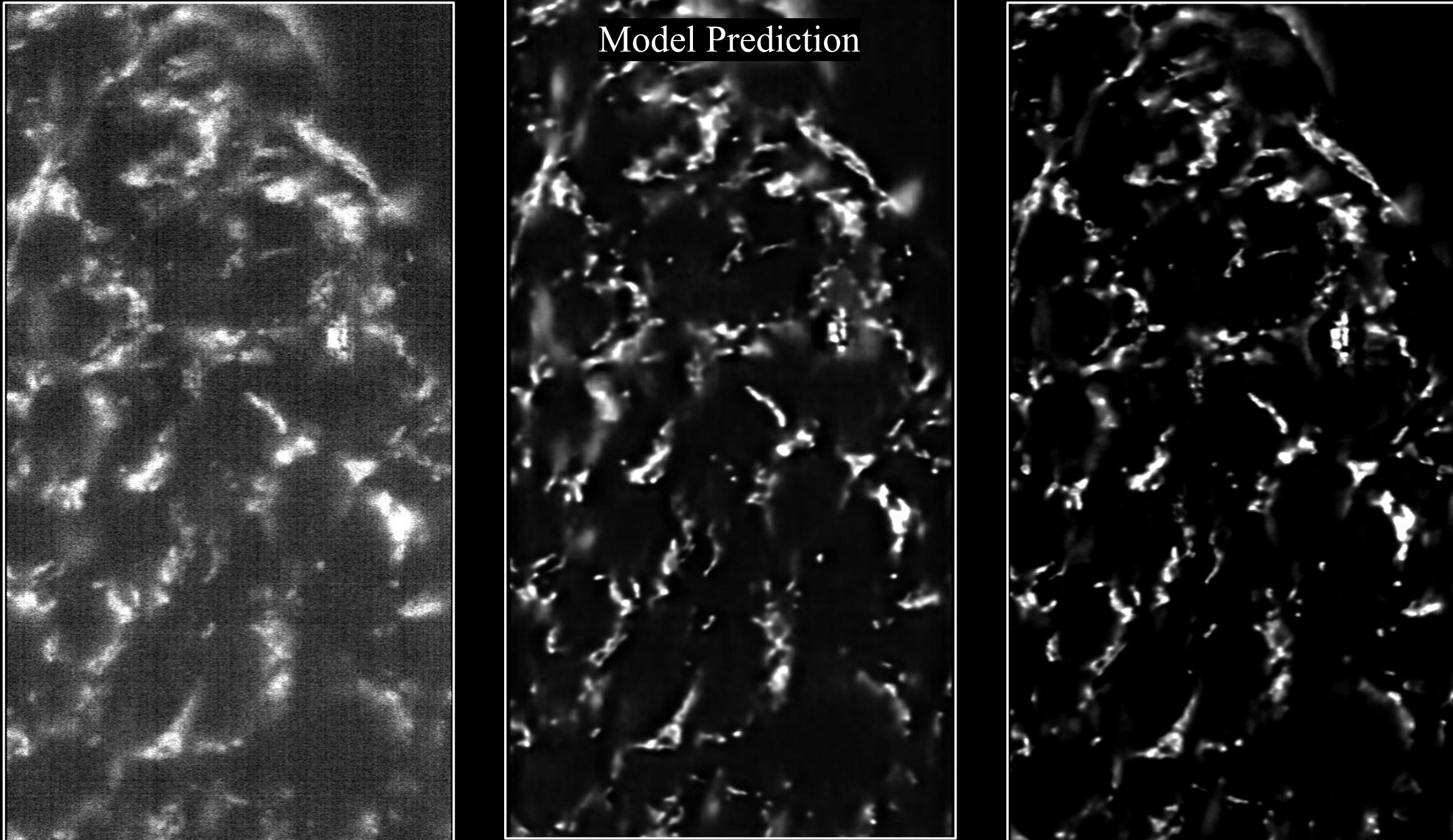


Prediction

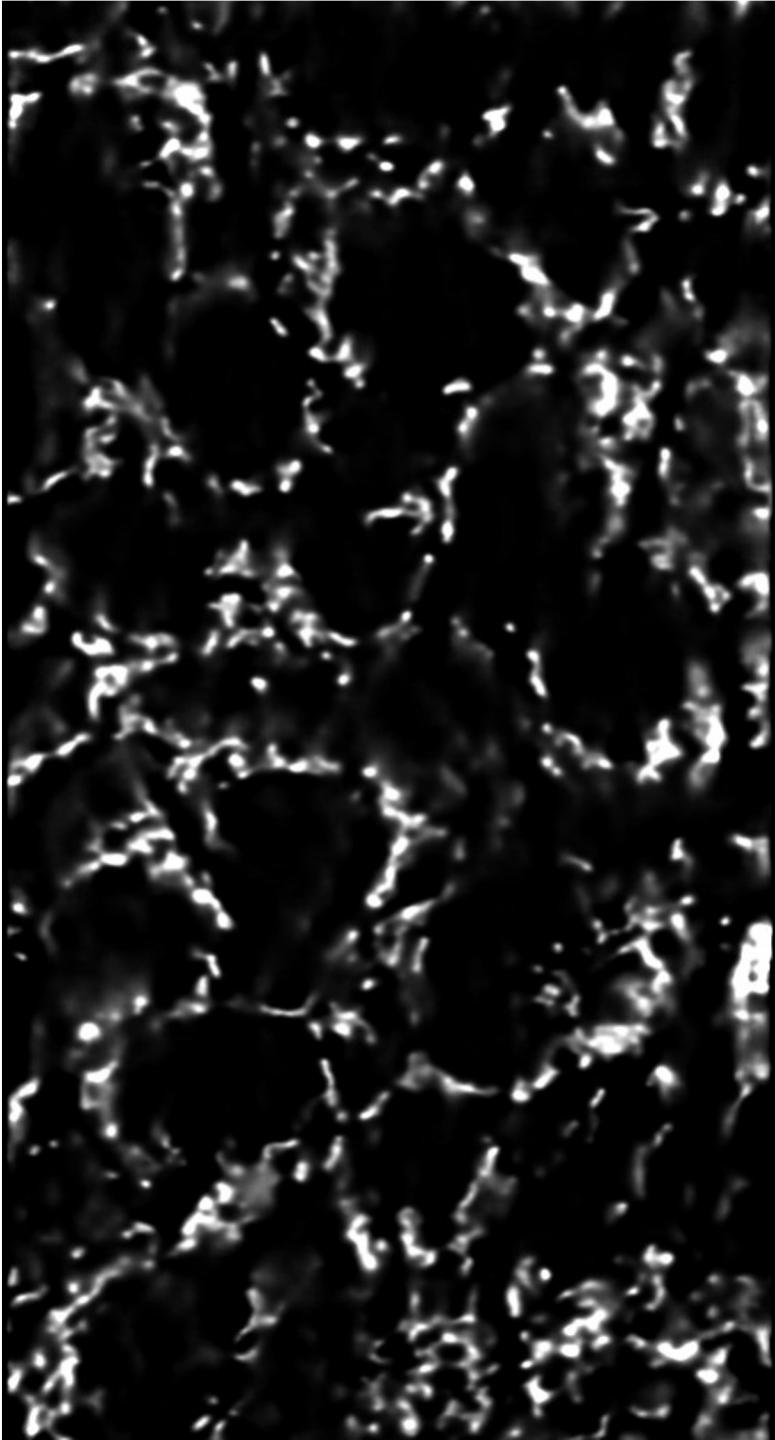


GT – Laser Power (20% | 10%)

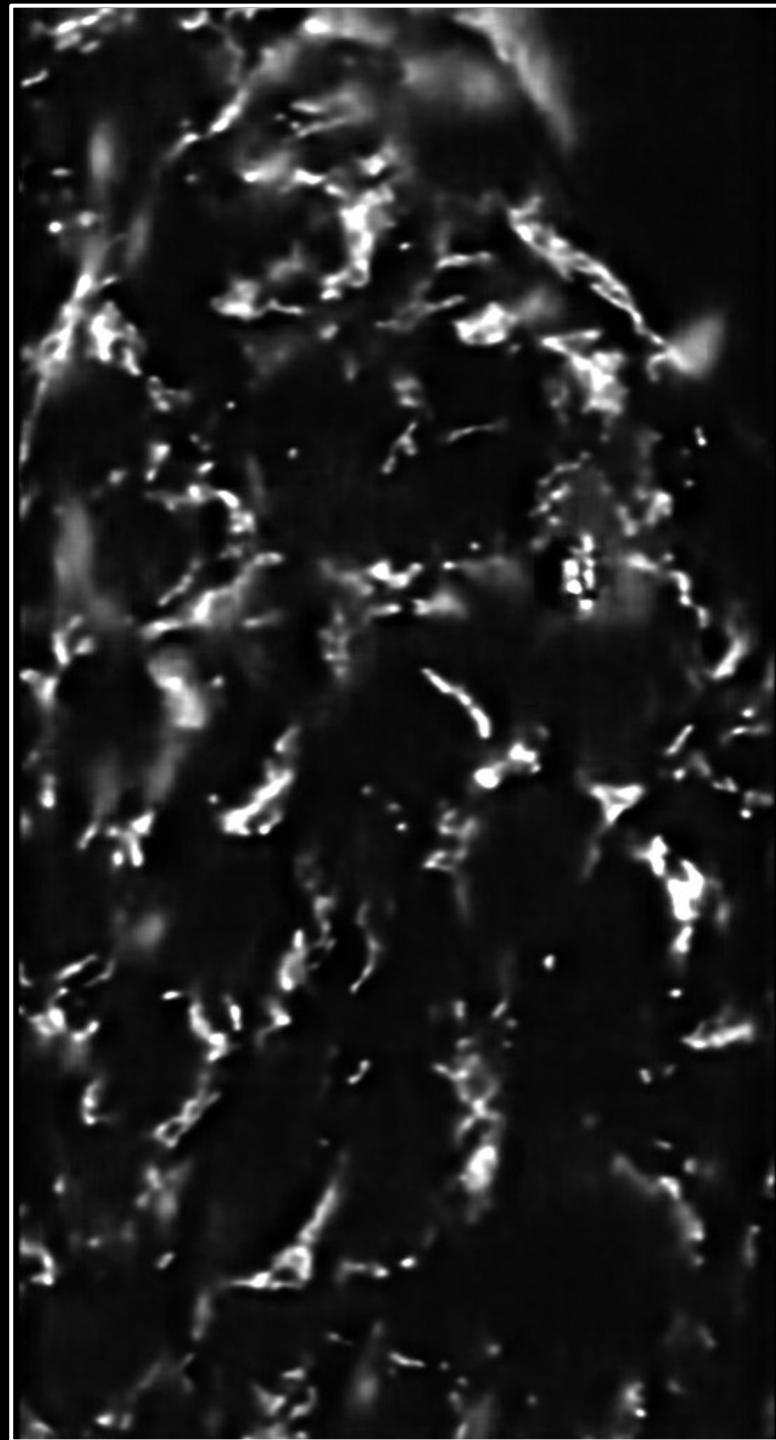
Model Prediction



Model 1.0



Model 1.1



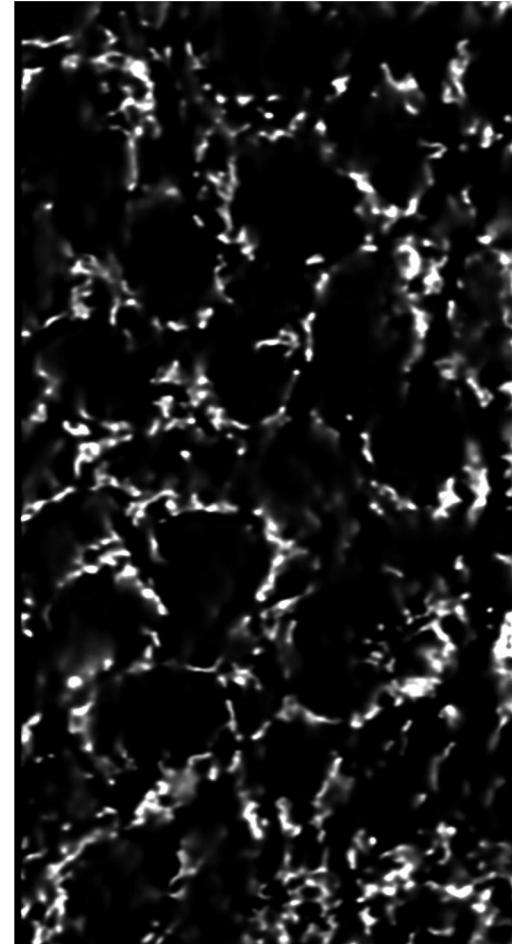
# Model\_Mito Raw Data – Denoised Data

sample 6

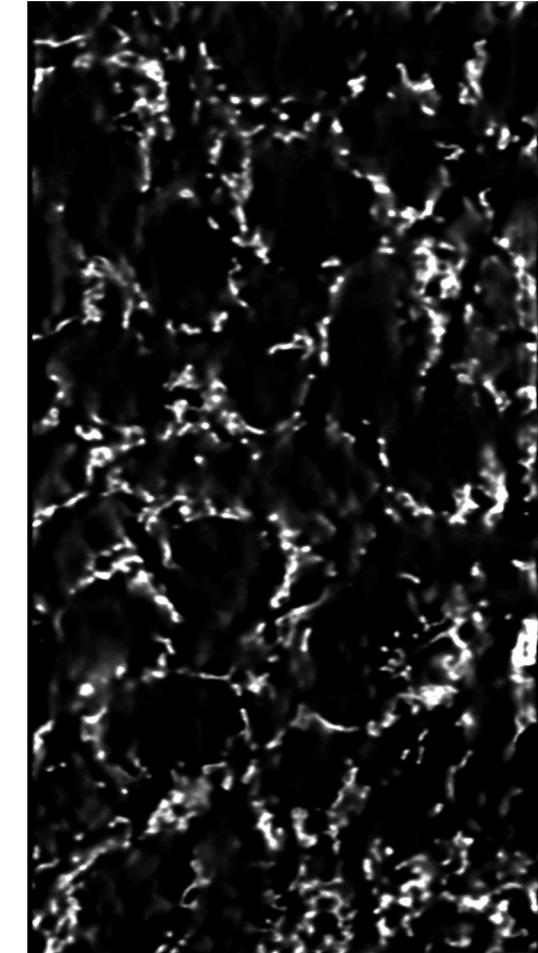
Mitochondria



Low – Laser Power (0.5 | 0.2)

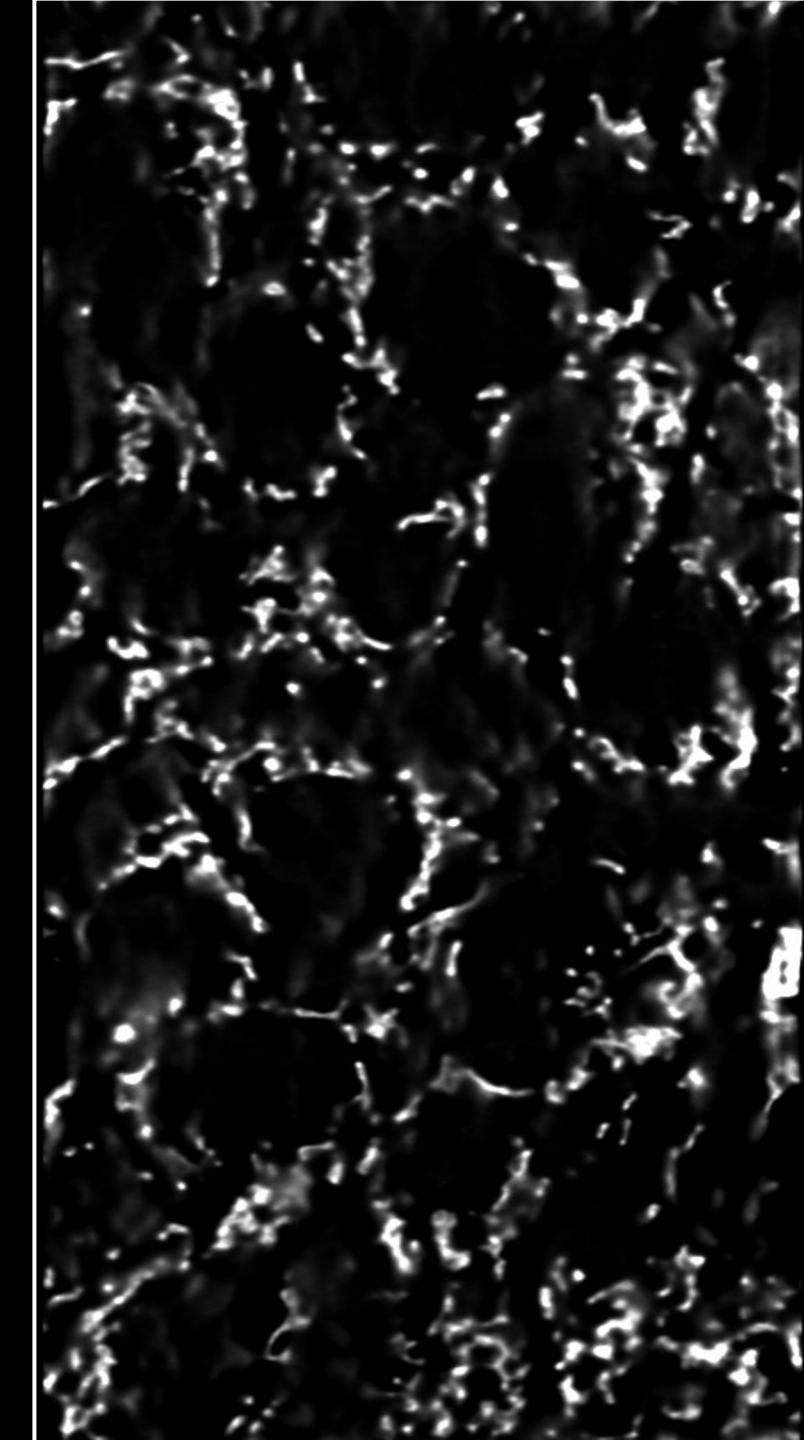
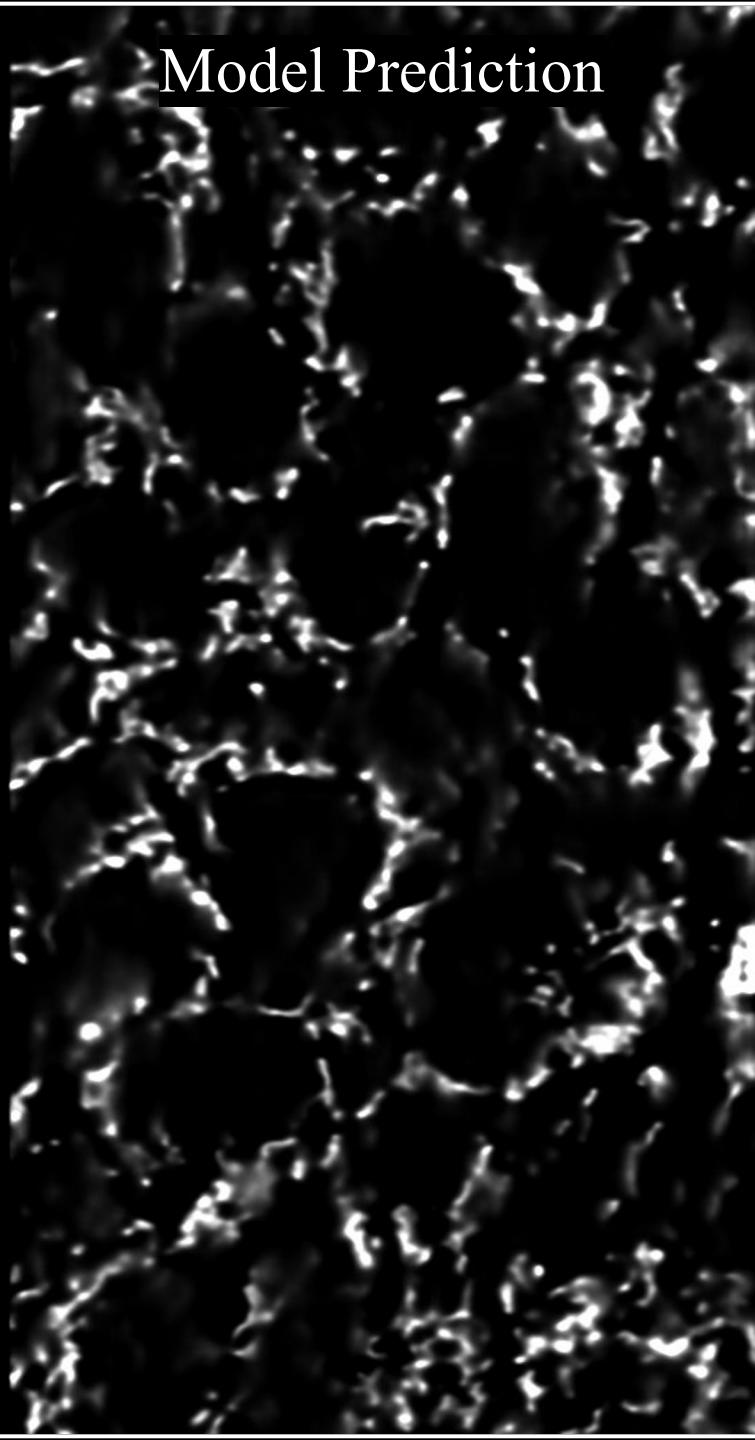


Prediction

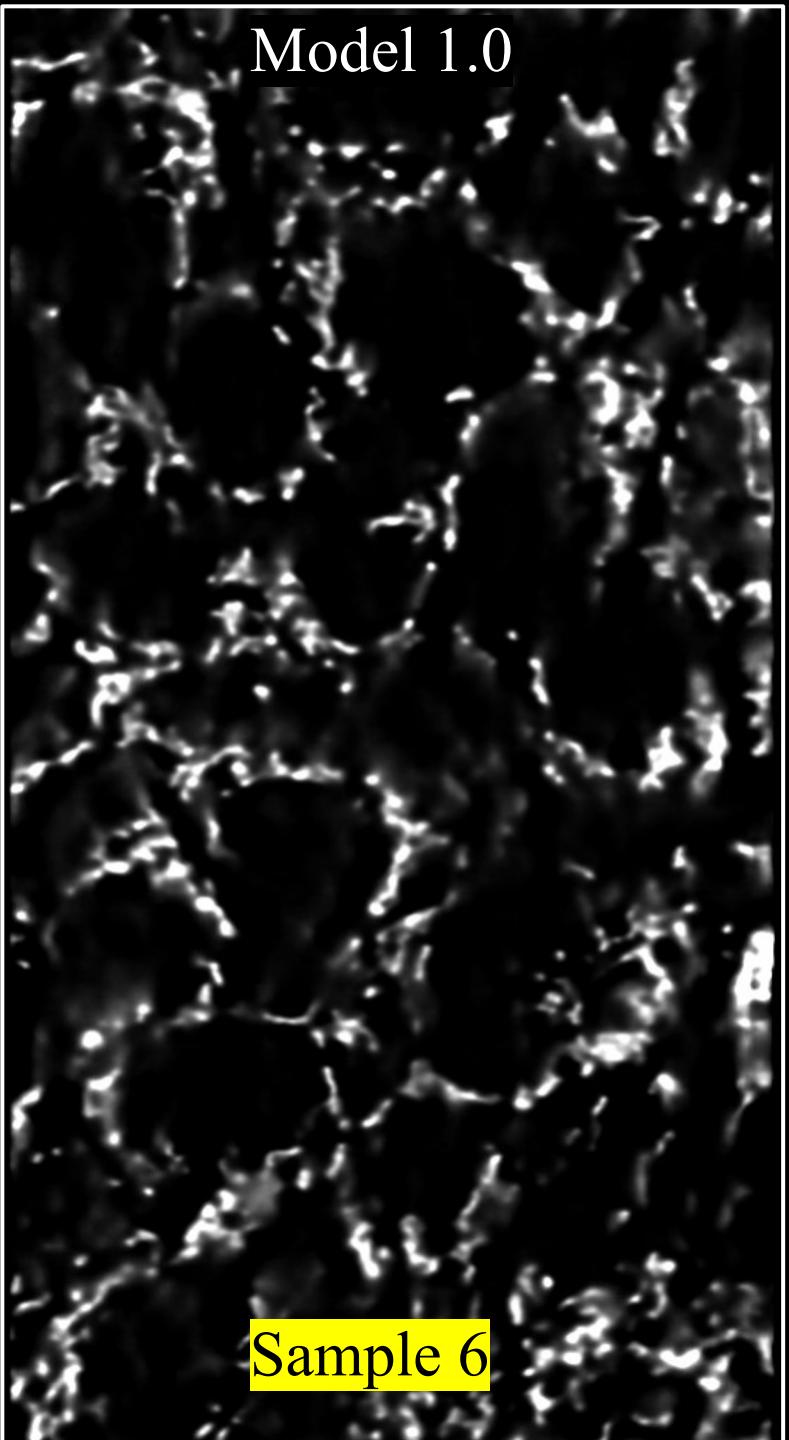


GT – Laser Power (20% | 10%)

Model Prediction

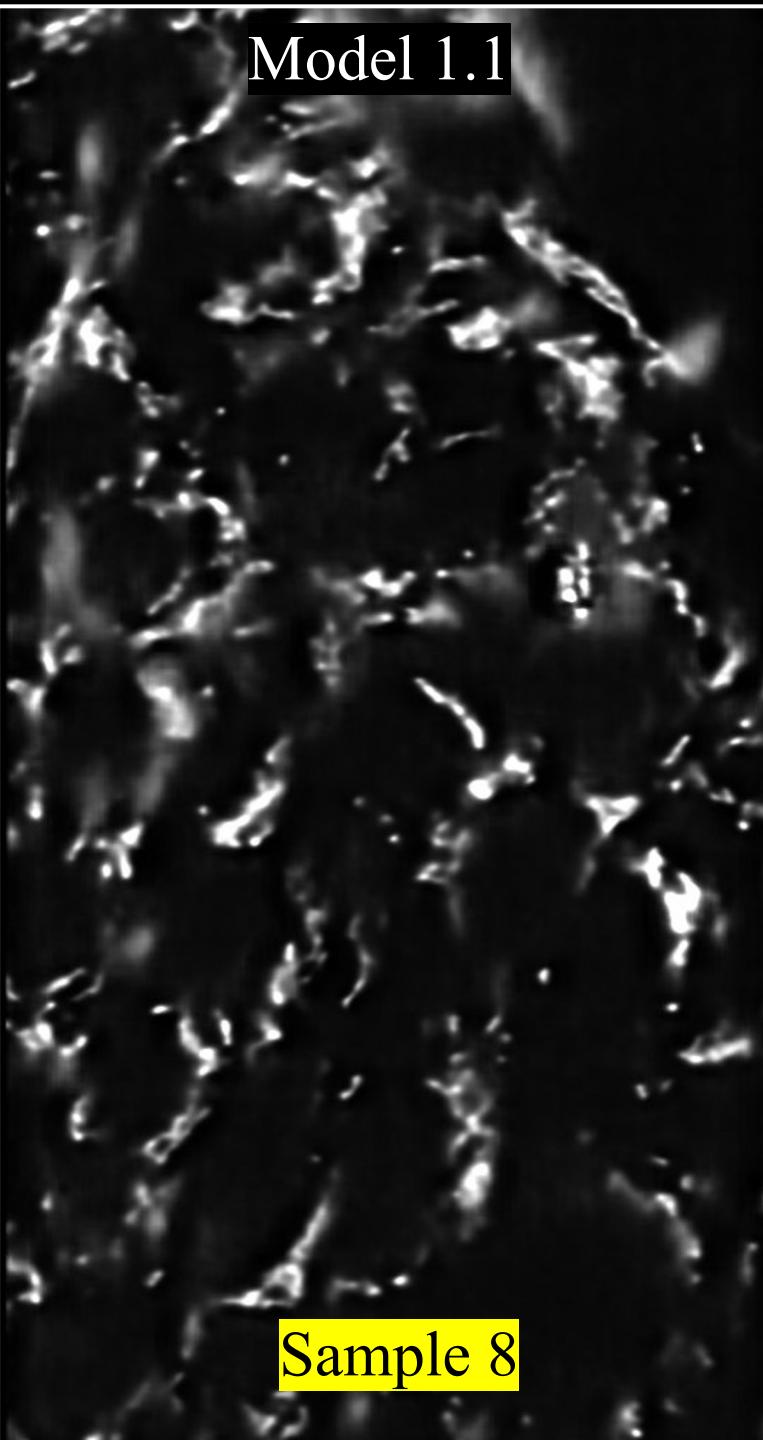


Model 1.0



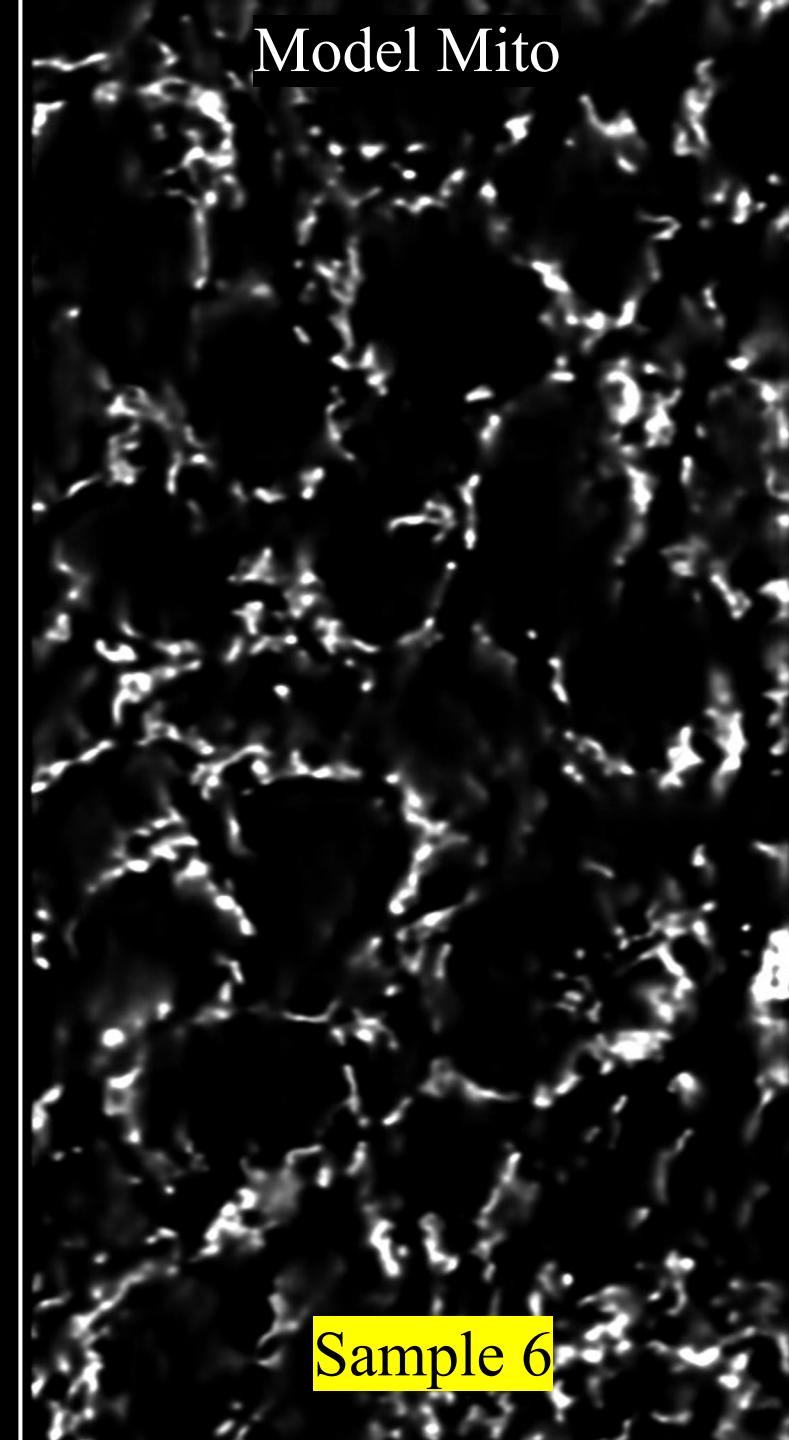
Sample 6

Model 1.1



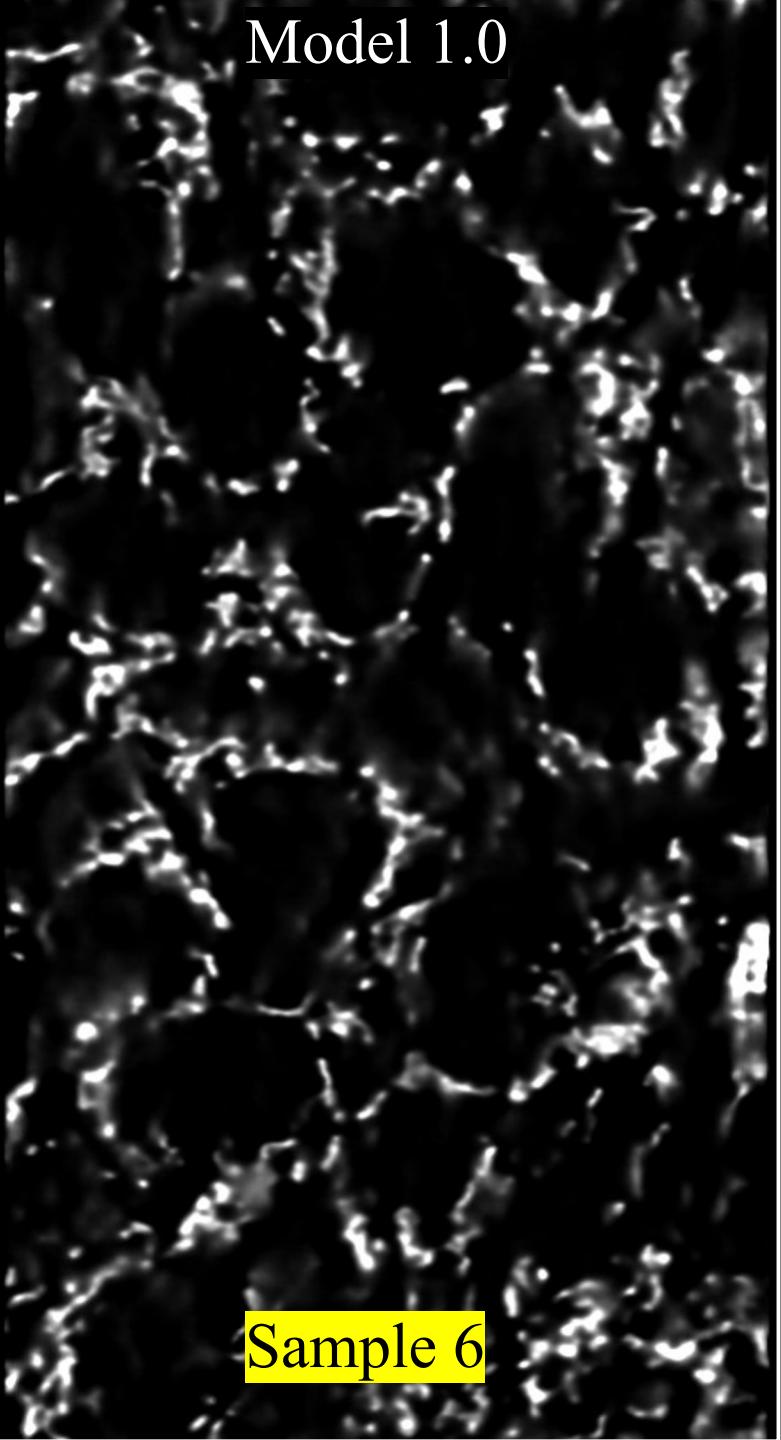
Sample 8

Model Mito



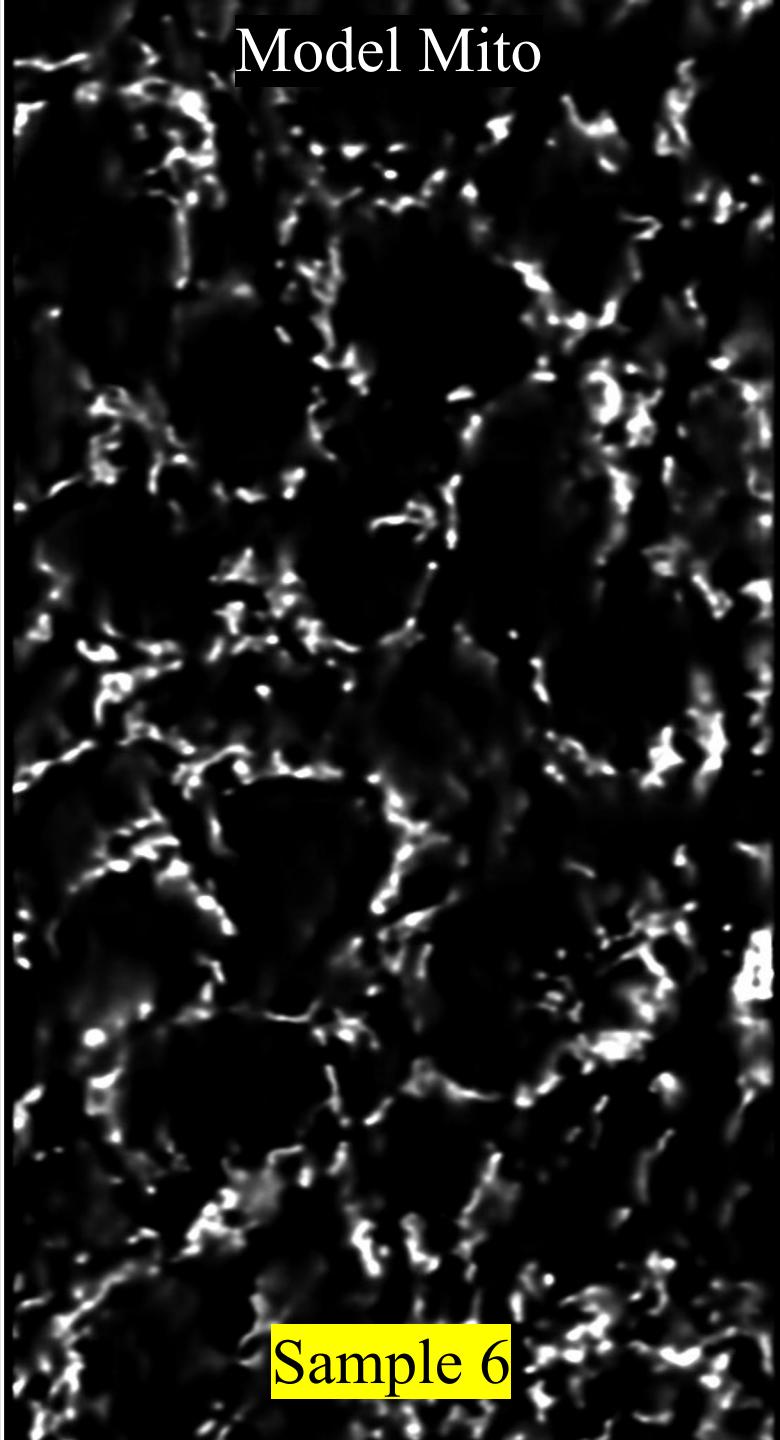
Sample 6

Model 1.0



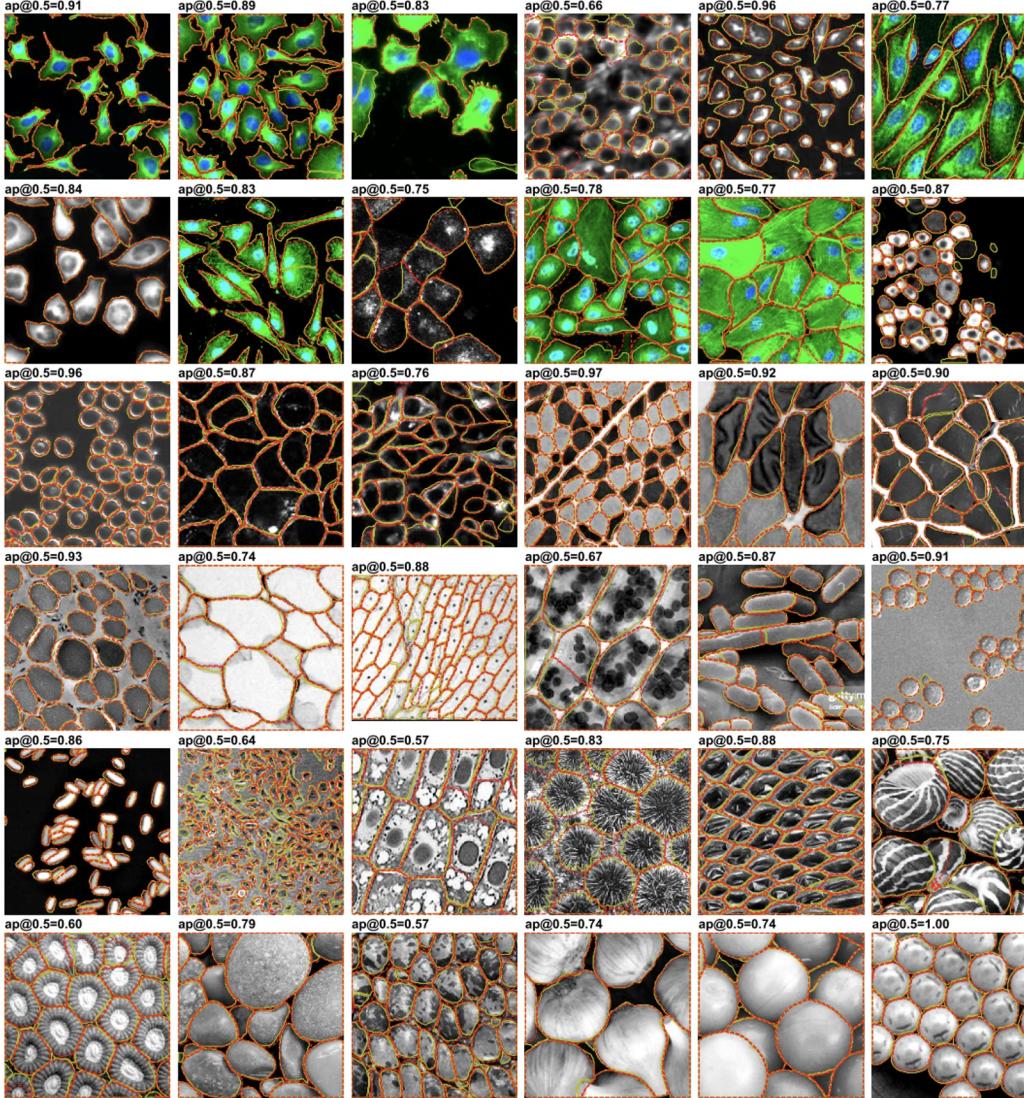
Sample 6

Model Mito



Sample 6

# CARE | CellPose



bioRxiv preprint doi: <https://doi.org/10.1101/2020.02.02.931238>; this version posted February 3, 2020. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

## Cellpose: a generalist algorithm for cellular segmentation

Carsen Stringer<sup>1\*</sup>, Michalis Michaelos<sup>1</sup>, Marius Pachitariu<sup>1\*</sup>

<sup>1</sup>HHMI Janelia Research Campus, Ashburn, VA, USA

\* correspondence to (stringerc, pachitariu) @ janelia.hhmi.org

Many biological applications require the segmentation of cell bodies, membranes and nuclei from microscopy images. Deep learning has enabled great progress on this problem, but current methods are specialized for images that have large training datasets. Here we introduce a generalist, deep learning-based segmentation algorithm called Cellpose, which can very precisely segment a wide range of image types out-of-the-box and does not require model retraining or parameter adjustments. We trained Cellpose on a new dataset of highly-varied images of cells, containing over 70,000 segmented objects. To support community contributions to the training data, we developed software for manual labelling and for curation of the automated results, with optional direct upload to our data repository. Periodically retraining the model on the community-contributed data will ensure that Cellpose improves constantly.

### Introduction

Quantitative cell biology requires simultaneous measurements of different cellular properties, such as shape, position, RNA expression and protein expression [1]. A first step towards assigning these properties to single cells is the segmentation of an imaged volume into cell bodies, usually based on a cytoplasmic or membrane marker [2–8]. This step can be straightforward when cells are sufficiently separated from each other, for example when a fluorescent marker is expressed sparsely in a subset of cells, or in cultures where cells are dissociated from tissue. However, cells are commonly imaged in tissue, where they are tightly packed and difficult to separate from one another.

Methods to achieve cell body segmentation typically trade off flexibility for automation. In order of increasing automation and decreased flexibility, these methods range from fully-manual labelling [9], to user-customized pipelines involving a sequence of image transformations with user-defined parameters [2, 8, 10, 11], to fully automated methods based on deep neural networks with parameters estimated on large training datasets [4, 5, 7, 12, 13]. Fully automated methods have many advantages, such as reduced human effort, increased reproducibility and better scalability to big datasets from large screens. However, these methods are typically trained on specialized datasets, and do not generalize well to other types of data, requiring new human-labelled images to achieve best performance on any one image type.

Recognizing this problem in the context of nuclear segmentation, a recent Data Science Bowl challenge amassed a dataset of varied images of nuclei from many different laboratories [14]. Methods trained on this dataset can generalize much more widely than those trained on data from a single lab. The winning algorithms from the challenge used established com-

puter vision algorithms like Mask R-CNN [15, 16] and adapted the algorithms for the biological problem. Following the competition, this dataset generated further progress, with other methods like Stardist and nucle-Alzer being developed specifically for this dataset [17, 18].

In this work, we followed the approach of the Data Science Bowl team to collect and segment a large dataset of cell images from a variety of microscopy modalities and fluorescent markers. We had no advance guarantee that we could replicate the success of the Data Science Bowl competition, because images of cells are much more varied and diverse than images of nuclei. Thus, it was possible that no single model would segment all possible images of cells, and indeed we found that previous approaches performed poorly on this task. We hypothesized that their failure was due to a loss of representational power for the complex cell shapes and styles in our dataset, and developed a new model with better expressive power called Cellpose. We first describe the architecture of Cellpose and the new training dataset, and then proceed to show performance benchmarks on test data. The Cellpose package with a graphical user interface (Figure S1) can be installed from [www.github.com/mouseland/cellpose](https://github.com/mouseland/cellpose) or tested online at [www.cellpose.org](http://www.cellpose.org).

### Results

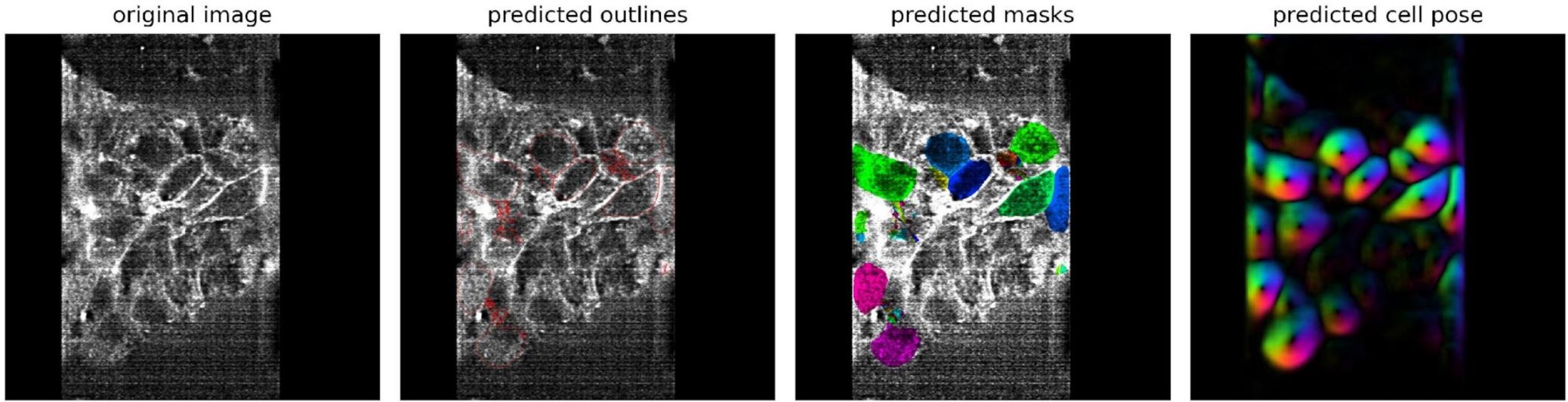
#### Model architecture

In developing an architecture for instance segmentation of objects, we first noted that a neural network cannot directly predict the cell masks in their standard representation as an image of pixel labels, because the labels are assigned arbitrary, inter-changeable numbers. Instead, we must first convert the masks of all cells in the training set to a representation that can

# CARE | CellPose

Slice 98

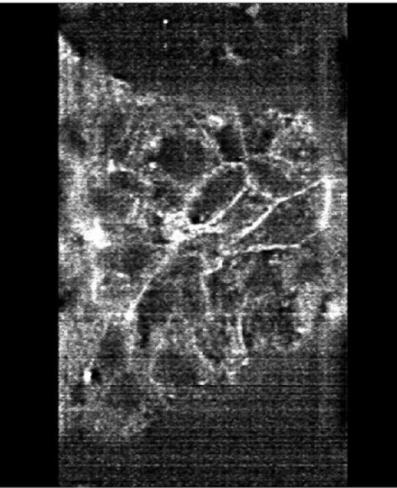
Raw



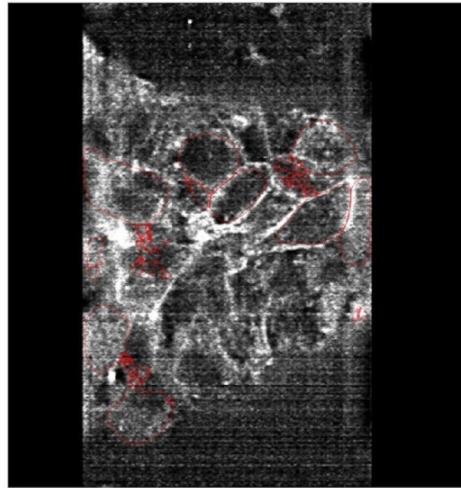
# CARE | CellPose

Slice 98

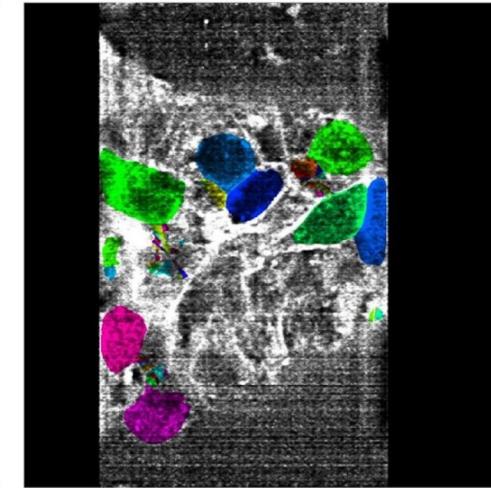
original image



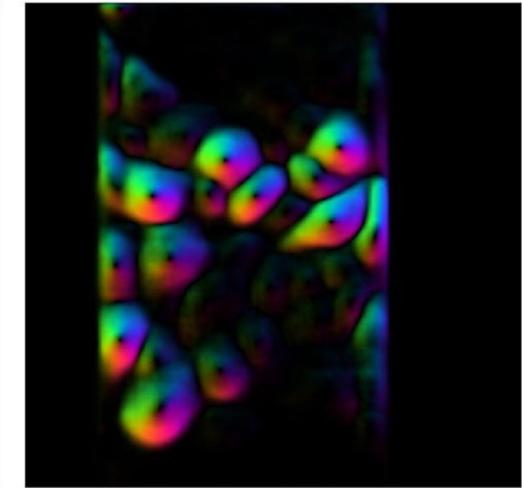
predicted outlines



predicted masks

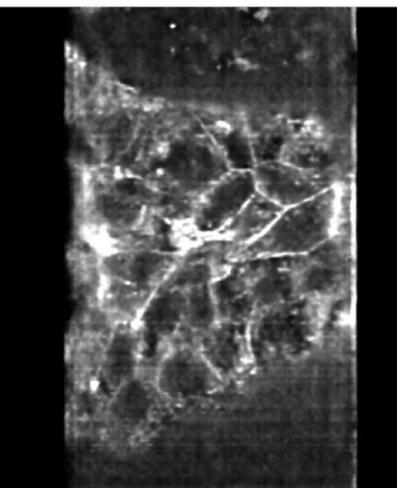


predicted cell pose

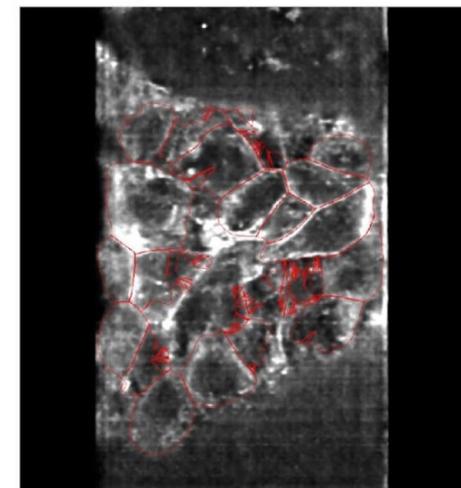


Raw

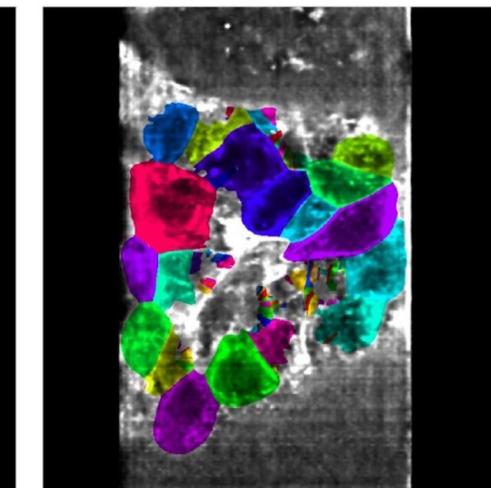
original image



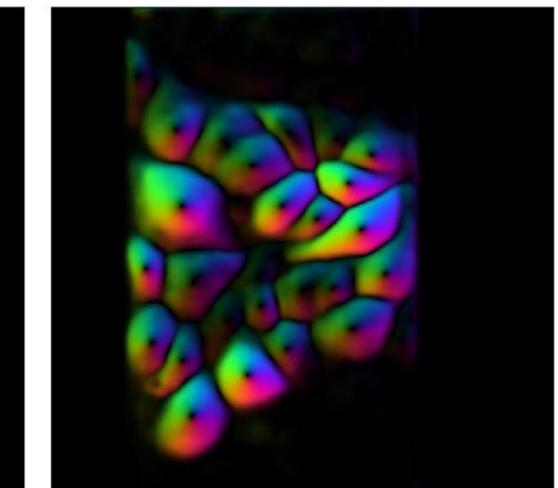
predicted outlines



predicted masks



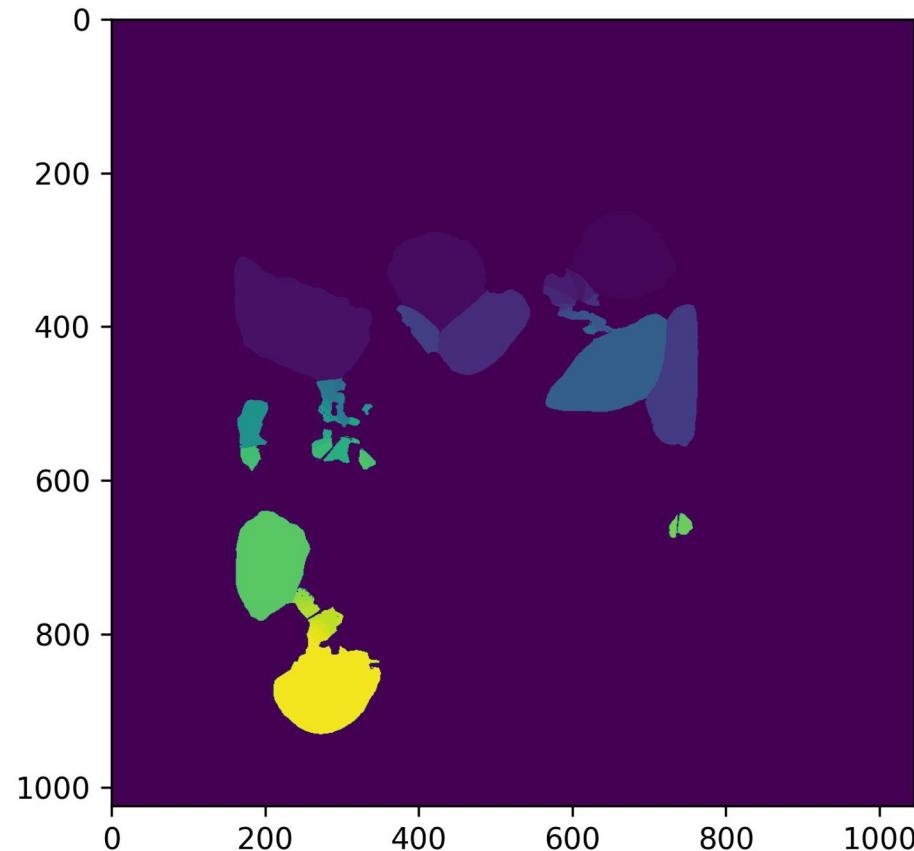
predicted cell pose



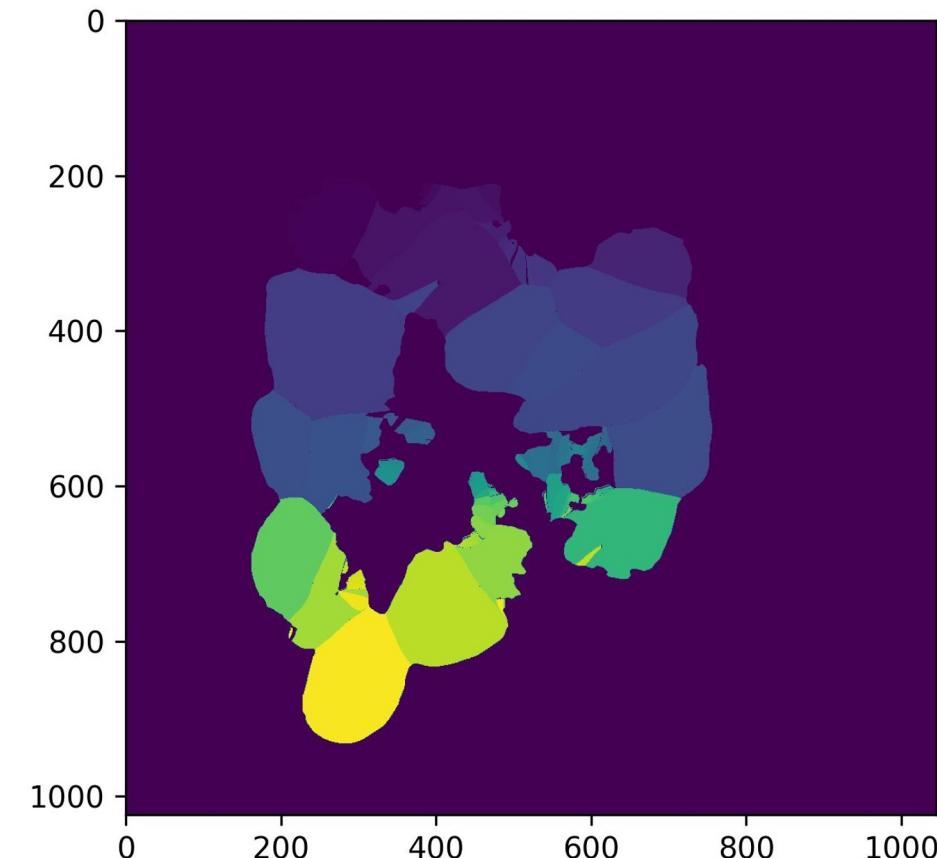
CARE

# CARE | CellPose

Slice 98



Raw

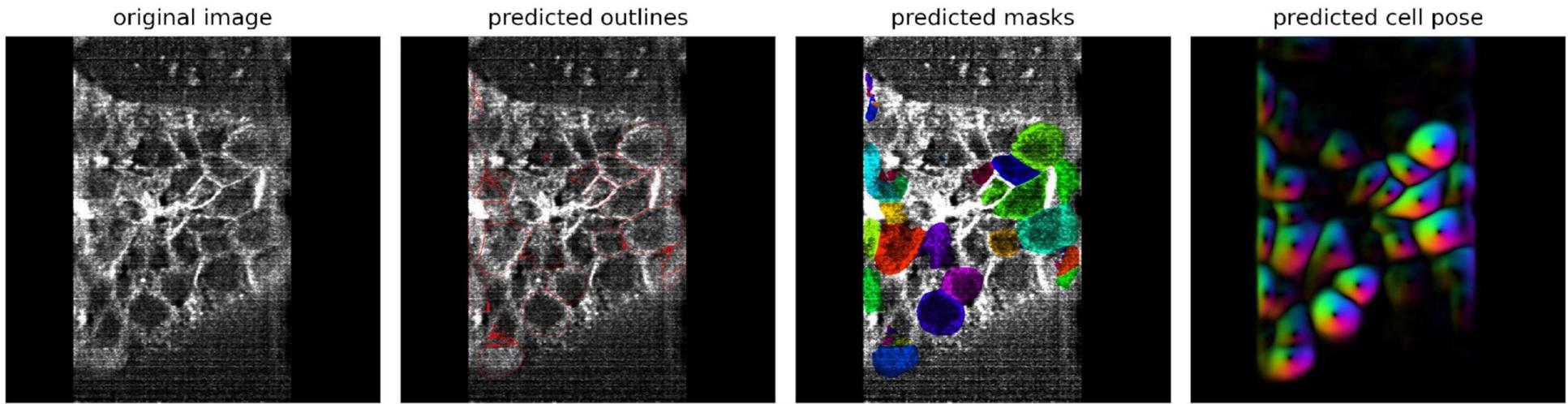


CARE

# CARE | CellPose

Slice 117

Raw

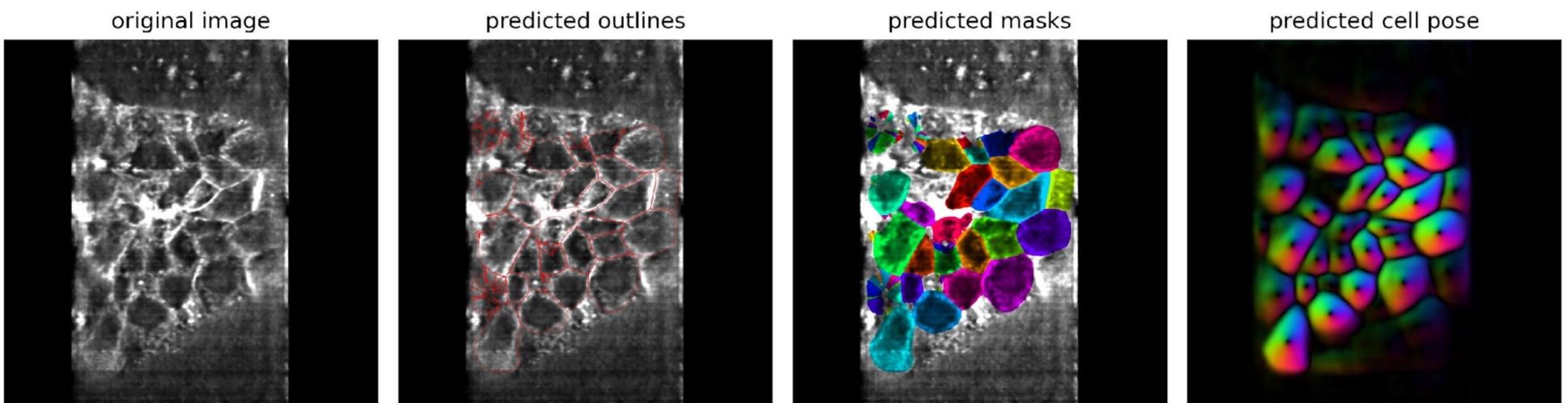
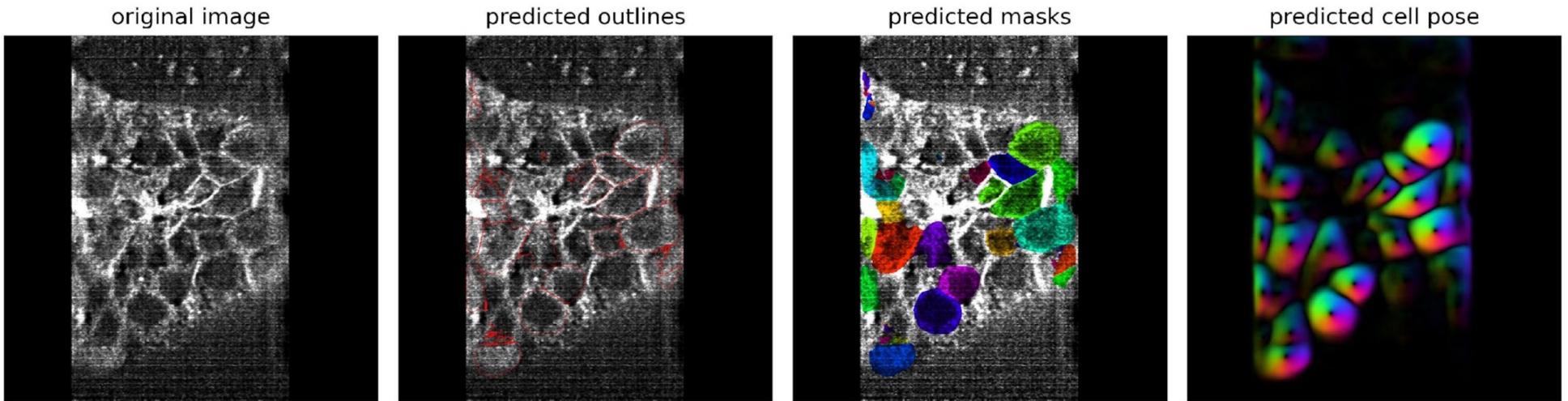


# CARE | CellPose

Slice 117

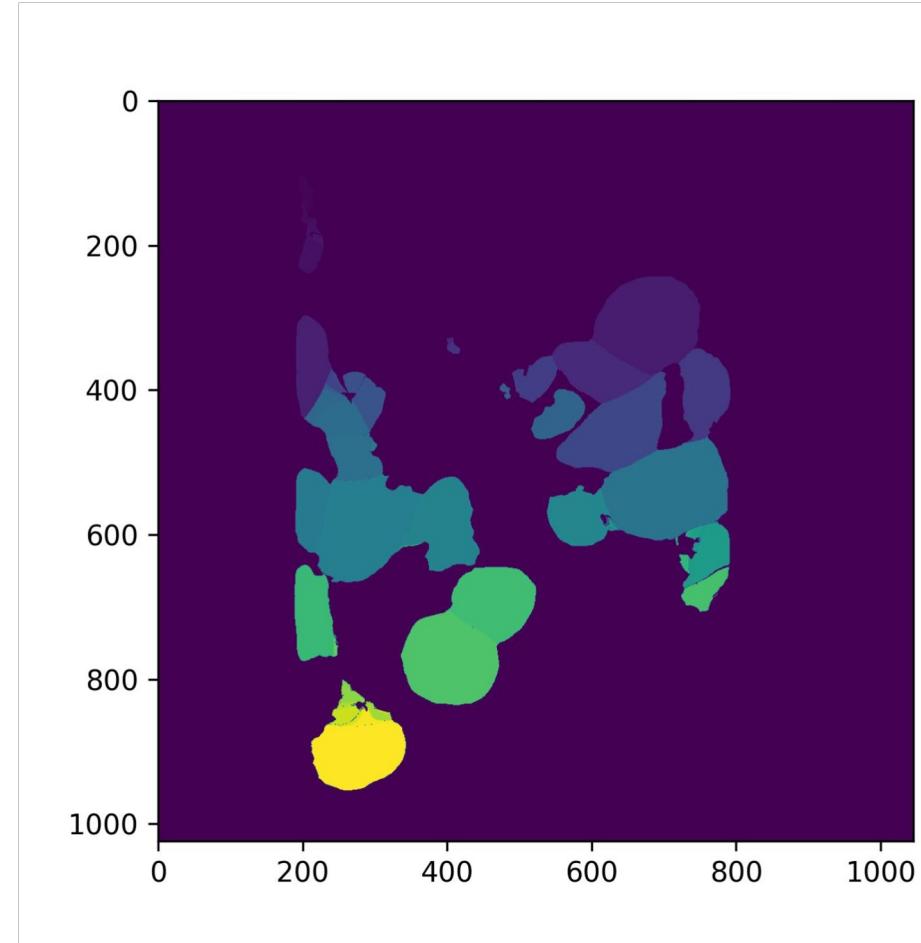
Raw

CARE

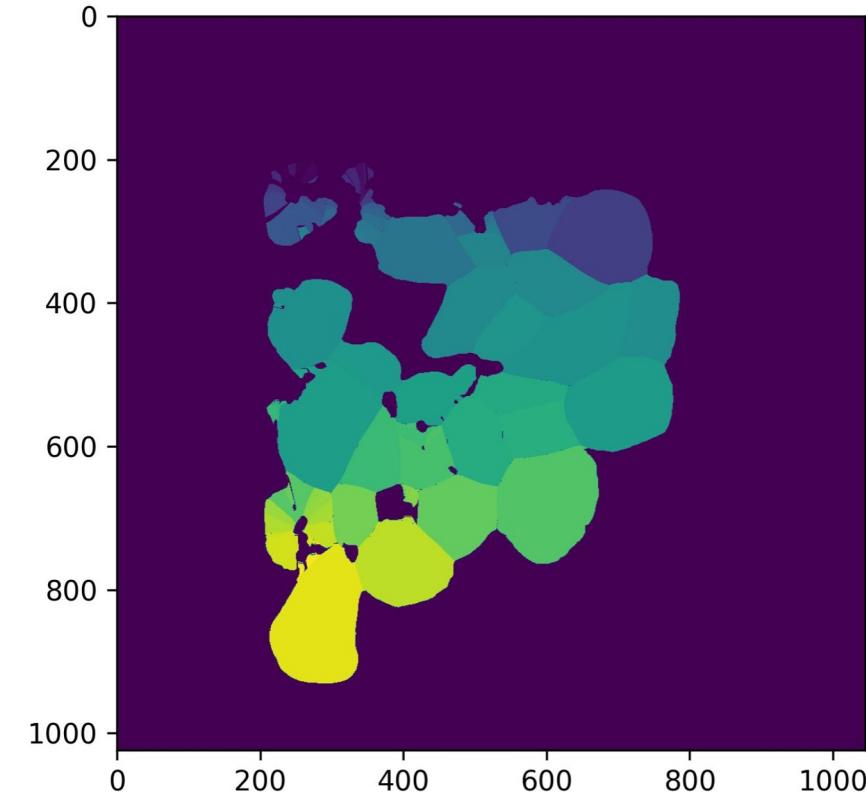


# CARE | CellPose

Slice  
117



Raw

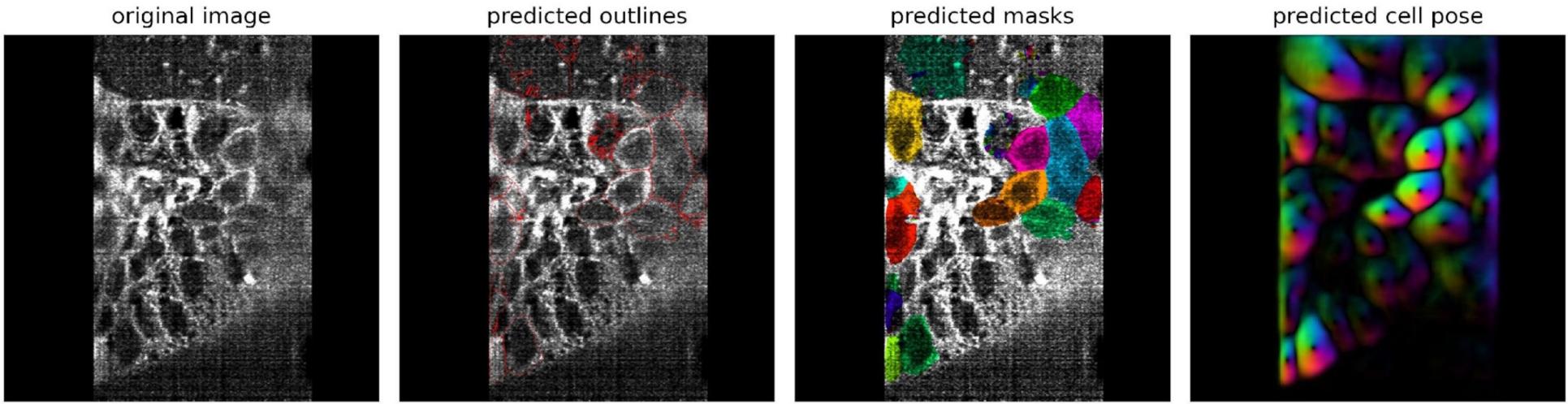


CARE

# CARE | CellPose

Slice 150

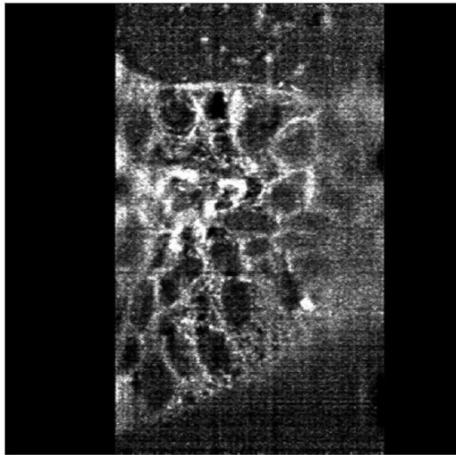
Raw



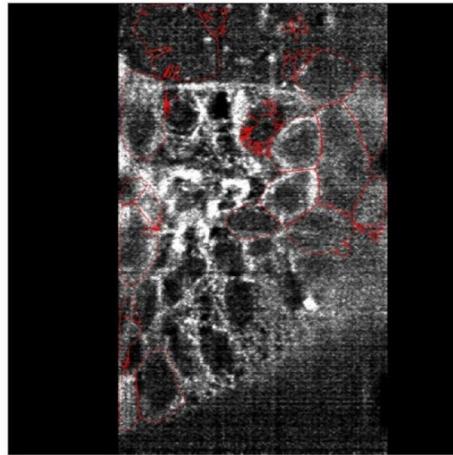
# CARE | CellPose

Slice 150

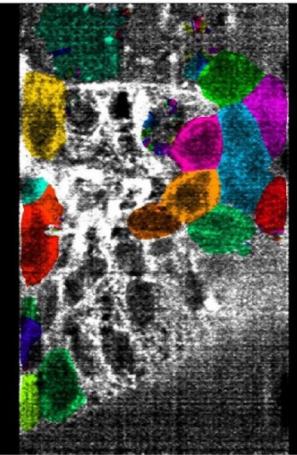
original image



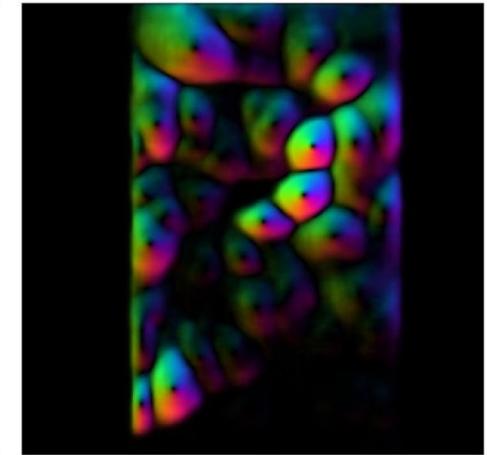
predicted outlines



predicted masks

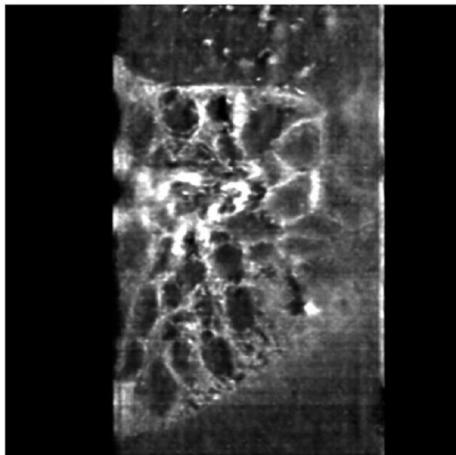


predicted cell pose

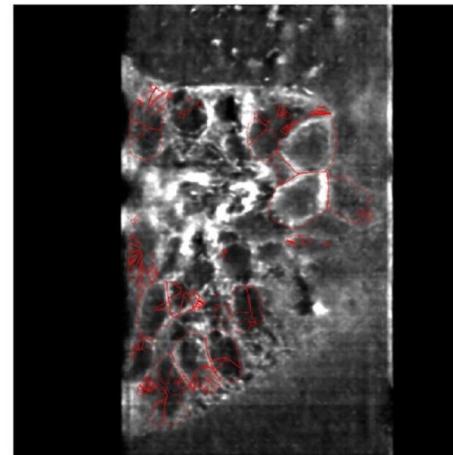


Raw

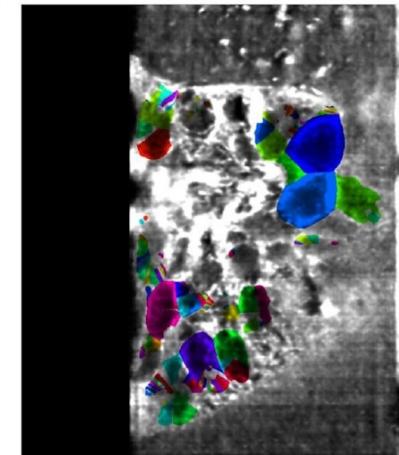
original image



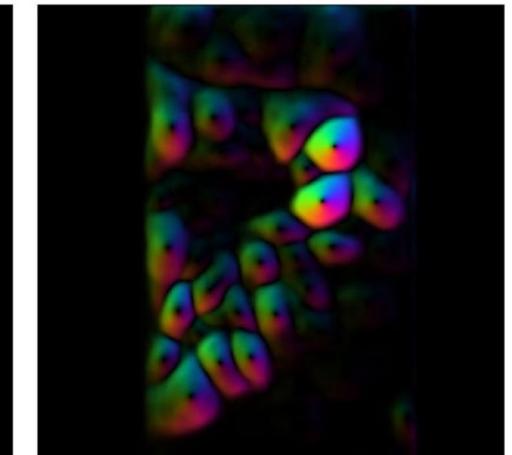
predicted outlines



predicted masks



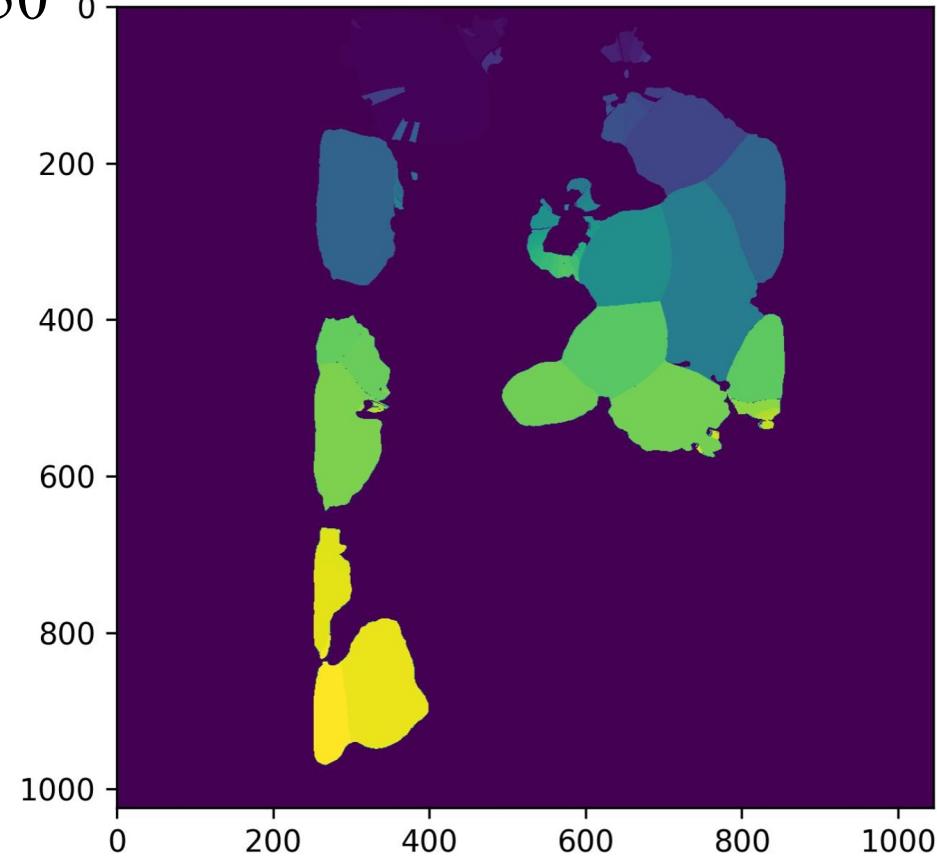
predicted cell pose



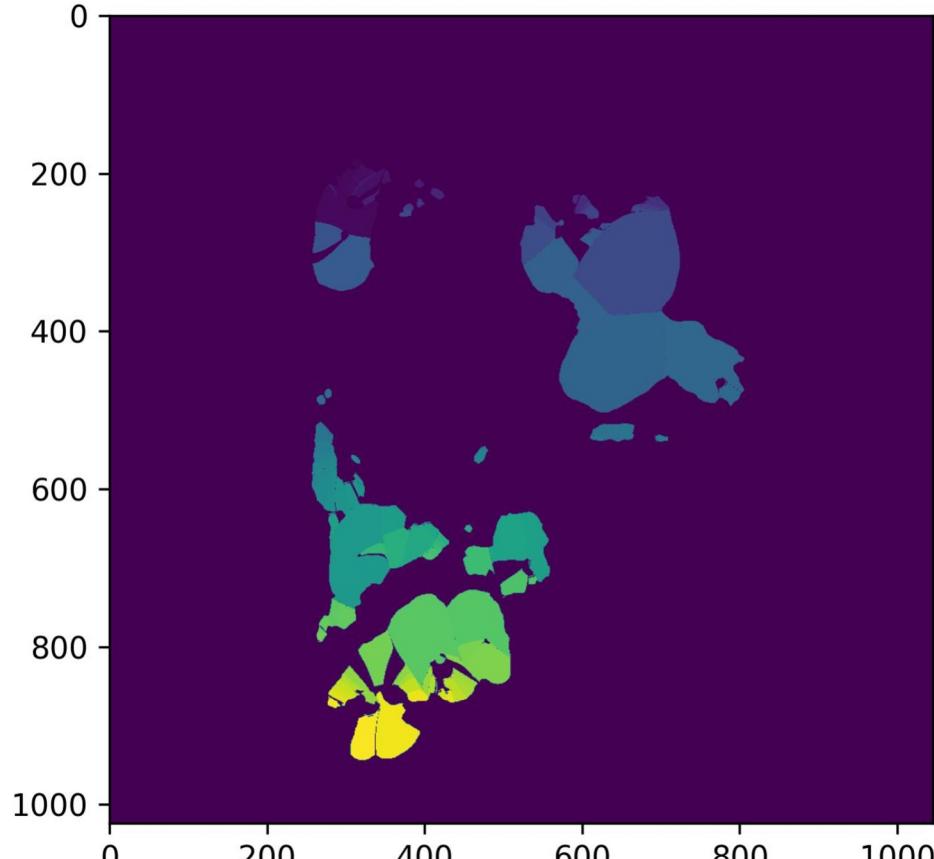
CARE

# CARE | CellPose

Slice 150



Raw



CARE

Thank you! You are an amazing team!!!



4D CELL Biology Lab

4D Computer Science, Engineering, Lattice Light-Sheet, Biology Laboratory

# Supplementary

Deep Learning methods are based on:

## Mathematical Models

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (\text{Mean Error Squared}) (Y_i - \hat{Y}_i)^2$$

Average Error Across the Model

$$\text{MAE} = \frac{\sum_{i=1}^n |y_i - x_i|}{n} = \frac{\sum_{i=1}^n |e_i|}{n}.$$

$$\text{MAE} = \frac{\sum |\text{actual} - \text{prediction}|}{\text{number of observations}}$$

['loss', 'lr', 'mae', 'mse', 'val\_loss', 'val\_mae', 'val\_mse']

