User Manual

Julienne LaChance
Daniel Cohen
Mechanical and Aerospace Engineering, Princeton University

Contents.

1. Introduction

This manual is intended to guide cell biologists through the code we have provided at:

https://github.com/CohenLabPrinceton/Fluorescence-Reconstruction

Our goal is to clarify the steps involved with training and testing a U-Net neural network, with code for every step provided in its entirety. Users should be able to collect their own raw data and utilize this code with minimal modifications. Ideally, this will give new users the flexibility to quickly train and utilize a model with their own cell types, equipment, and labeled features.

A sample dataset (of 256x256 pixel^2 images) for testing the code we provide in this manual can be found at the following Princeton DataSpace link:

http://arks.princeton.edu/ark:/88435/dsp013r074x85b

This repository contains our complete test set for every experimental condition, along with the corresponding reconstructed images produced by our model.

We additionally provide two smaller datasets (subsets of our complete test set) for rapid testing:

http://arks.princeton.edu/ark:/88435/dsp019w032593v

The DataSpace content enables users to work through the entire training workflow using a portion of our keratinocyte dataset (1,000 sub-images), imaged at 10x magnification. The input images are phase

contrast images (“PHASE”) and the output images contain Hoechst 33342-stained nuclei, imaged using a standard DAPI fluorescent filter set (“DAPI”).

If users would like to test out a sample DIC dataset instead, we provide another data subset at:

http://arks.princeton.edu/ark:/88435/dsp019880vt87x

1.1 Scope

This manual will describe each module of the code in detail. Most of the provided code is written in Python [1], due to its readability and widespread use within the machine learning community. We assume basic familiarity with FIJI/ImageJ [2,3], an open-source image processing toolbox. For neural network training, we use the libraries TensorFlow/Keras [4,5], and we use Jupyter notebooks [6] to assist in demonstrating how to use the model to process large images. We assume only basic familiarity with Python, as most modifications to the code will involve basic changes, such as changing paths to the user’s local folders.

1.2 Hardware Configuration

Our experimental tests were run using a single NVIDIA Tesla P100 GPU within a research computing cluster. Each GPU processor core has 16 GB of memory, and cluster nodes are interconnected by an Intel Omnipath fabric. Each cluster node has 2.9TB of NVMe connected scratch and 256G of RAM. The CPUs are Intel Broadwell e5-2680v4 with 28 cores per node.

Additionally, the code was tested on a standalone desktop, equipped with a single NVIDIA GeForce GTX 1070 Ti GPU. The desktop additionally runs with an ASUS Z97-A Intel motherboard, an Intel Core i5-4690 Haswell Quad-Core processor, 16 GB of RAM, and a 250 GB Samsung 850 EVO SSD.

Typical training times spanned from several hours on the cluster, to more than a day on the desktop. Users should be aware that access to a cluster or powerful GPU will save substantial time in training.

2. Process Overview and Setup

The code workflow is provided in a series of folders which indicate the order of the processes which comprise the workflow. Users may choose to use the code in these folders (named “0_Helper_Functions”, “1_Splitting_Images_Into_Patches”, etc.), or may find those same scripts grouped together in the folder entitled “All_Files”. Regardless of whether the user wishes to utilize individual scripts or utilize all the code together from “All_Files”, the user needs to ensure that the user input variables in each script are modified correctly, and that paths are set correctly to the helper functions. We describe these steps in more detail in Section 4.

2.1 Python and Jupyter Notebooks

In order to use the Python scripts, users will need to install Python 3, plus the versions of TensorFlow/Keras which are compatible with their GPU drivers. Additional libraries to be installed are indicated in each Python script and their corresponding helper scripts (see Section 2.2 for a complete list). Additional requirements will be noted in each subsection of the Workflow below.

The Python language is well-documented and supported by a strong open-source community. Python can be downloaded at the following link: https://www.python.org/downloads/

We use TensorFlow, a popular deep learning library, and Keras, which is an API that can be used with TensorFlow, and provides high-level programming tools to make coding with TensorFlow easier. Keras and TensorFlow installation instructions can be found at: https://keras.io/

Additionally, we provide our code in the form of Jupyter notebooks when appropriate. Jupyter is a handy interface for writing code in a number of languages, including Python, and enables users to display output in a "notebook" format in real-time. Jupyter is popular today for quick prototyping in the machine learning community, and we use it for easily displaying images for the user. Jupyter installation resources can be found at: https://jupyter.org/

2.2 Python Libraries

All the required Python libraries for the provided scripts are listed here:

cv2, jupyter, keras (tensorflow), libtiff, os, math, numpy, Pillow, random, scipy, shutil, sklearn, sys, time, tqdm, warnings

Users may choose to install them in advance, or install only those libraries associated with portions of the code they are interested in running.

3. Data Collection

In order to run this code, the user must collect a dataset of matched image pairs. We will not describe the data collection process in detail here, but we assume users have transmitted-light input images paired with high-quality images of labeled fluorescent features, such as cell nuclei or junctions. We recommend using image registration so that fluorescent features align spatially with transmitted-light features, and for ease of processing downstream in the workflow, we recommend cropping matched pairs into height- and width- values which are multiples of 256 (the height and width of one patch, and therefore the input/output size of images to/from the model). While this cropping step is not necessary, the failure to crop matched pair images will require the user to re-write the Splitting_Macro.ijm FIJI/ImageJ script in Step 1. Here we assume the image pairs are stored as image sequences of 16-bit TIFF files in two separate folders, and that all images are of the same size.

Sample matched image pairs (of size 512x512 pixels^2) in their respective folders are provided in the folder "1_Splitting_Images_Into_Patches" within "Sample_Images". (We also provide a complete, pre-cropped dataset of 1,000 matched sub-image pairs at the DataSpace link.) In this example, we wish the "phase" image to become inputs to the network, and for the "DAPI" nuclei images to become the ground truth for the model output.

4. Workflow

Introduction.
The workflow consists of the following steps:
Step 1: Chop Images Into Patches.
Step 2: Perform Test/Train Splits.

Step 3: Get Data Statistics.
Step 4: Train the Network.
Step 5: Test the Model.
Step 6: Get Additional Prediction Statistics.
Step 7: Process New Images.

We now will discuss each step of the workflow in more detail. Initially, the raw images must be chopped into many patches of size 256x256 pixel^2. Then, the dataset must be split into a training set and a test set, and dataset statistics are collected from the training set for normalization of the data. Next, the network is trained, and evaluated on the test set, with relevant statistics such as test accuracy reported to a CSV file. Finally, we demonstrate how to use the network to process a new large input image.

Some of the scripts in this workflow rely on helper scripts: namely, "**data_loader.py**", "**models.py**", and "**utils.py**". These can be found within the folder "0_Helper_Functions" (or "All_Files"). These scripts serve the following functions:

"**data_loader.py**": This Python file contains helper functions which load in the user's dataset. This is necessary, for example, to read in the input/output images for training or testing the model.
"**models.py**": This contains the U-Net architecture implementation. If the user wishes to test different models, this Python file may be modified.
"**utils.py**": This contains helper functions, including the implementation of the Pearson correlation coefficient loss function, and also stores information about the user's dataset, including normalization statistics and paths to folders containing the user's dataset. This script should be modified as dataset statistics are collected and experimental conditions and datasets are added.

The Python scripts in the following sections of the workflow expect these helper files to be in the same folder, or for paths to the files to be set appropriately in the import statement. The user should be especially wary of this if attempts are made to utilize individual scripts, rather than modifying code from within the "All_Files" folder.

4.1 Step 1: Chop Images Into Patches.

The main script to perform this step is entitled "**Splitting_Macro.ijm**", and can be found in either folder: "1_Splitting_Images_Into_Patches" or "All_Files". This portion of the code is provided as a FIJI/ImageJ script. Users can test the functionality of this script on the test images provided in the subfolder, "Sample_Images".

The sole purpose of this macro is to chop a sequence of images within a specified input directory into a number of 256x256 pixel^2 patches, and save these patches into a specified output directory.

To use this macro, four variables must be set by the user, which are detailed in the macro comments. These variables are paths to the input and output directories, an additional string to set the name of the output patch files, and an integer *n*, which indicates how many patches to produce from each input image. Although *n* can be automatically determined, we chose to manually set this value to ensure that the input image size would be verified by the user.

The user should apply this script to each of the image sequences comprising their dataset: for example, on both the “phase” and “DAPI” image sequences. For standard applications, this results in the transmitted light image patches saved in one folder, with the fluorescent label image patches saved in a separate folder.

If the user applies this script to the test images in the GitHub folder “Sample_Images”, four sub-images should be produced of size 256x256 pixel^2 for each test image (“phase” and “dapi”) corresponding to the four quadrants of each test image.

This script has no dependencies.

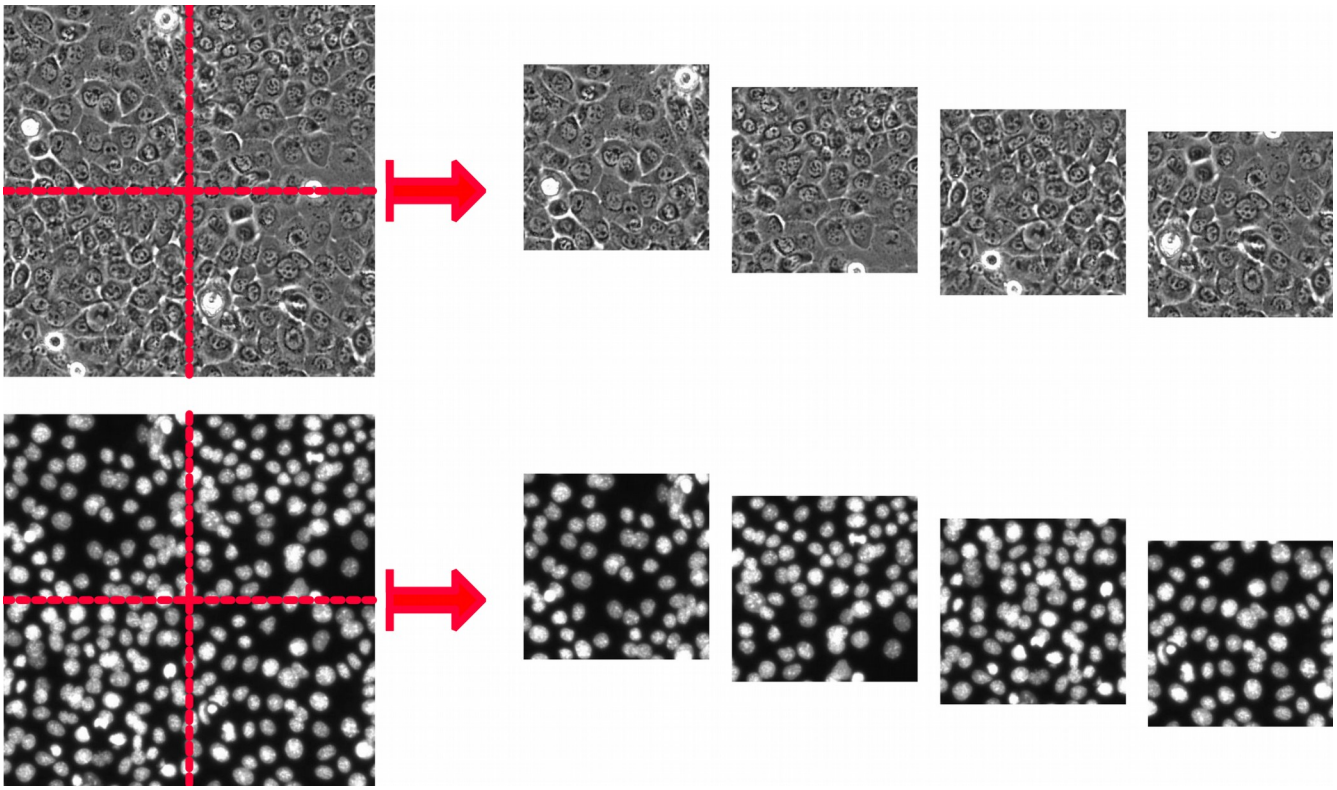

Figure 1. Chopping sample images into patches. We provide a 512x512 pixel^2 image, and demonstrate how to split this image into four 256x256 pixel^2 sub-images. These sub-images together form the dataset with which the user will train the neural network: transmitted-light sub-images are the input to the network, while fluorescence sub-images are the ground truth for the output.

4.2 Step 2: Perform Test/Train Splits.

The main script to perform this step is entitled “**run_test_train_split.py**”, and can be found in either folder: “2_Perform_Test_Train_Splits”or “All_Files”. This portion of the code is provided as a Python 3 (.py) file, as will be most of the subsequent scripts. Sample images from the previous section may be used to test code from any of these sections; however, the model will not be very accurate when trained with so few images! Therefore, we have provided a larger, pre-cropped datasets for your convenience.

For example, begin with the dataset consisting of 1,000 matched image pairs at the following Princeton DataSpace link:

http://arks.princeton.edu/ark:/88435/dsp019w032593v

The .zip file provided here contains a subset of our experimental dataset using keratinocyte type cells, imaged at 10x magnification. The input images are phase contrast images which can be found in the folder “PHASE”, and corresponding ground truth (output) images are fluorescent images of the cell nuclei which can be found in the folder “DAPI”. The nuclei were stained using Hoechst 33342 and imaged using a standard DAPI filter set.

Once the complete dataset has been pre-processed (including chopping into patches), it must be split into a training set and a test set. The training set will consist of the data that is used to train the network, while the test set will consist of images which are “held out” (not seen by the network during training), and are only used to validate the accuracy of the model later on. Matched pairs of images must be randomly selected to be sorted into these two categories.

To use this code, four inputs must be specified by the user. These are the path to the folder containing the entire dataset (“base_path”), the names of the sub-folders containing the input and output images, respectively, and a float named “split”, which determines the percentage of the dataset to use in the training set. The script creates new sub-folders within the base directory to store the training and test subsets, respectively. Input/output images are stored in separate folders within these training and test subset folders.

If the user wishes to test this script on sample data, please download content from the DataSpace link provided above, and ensure your path to the folder (“base_path”) is set correctly to the folder which contains both “PHASE” and “DAPI” sub-folders. The newly split training and test datasets can be used in all future steps as well.

This script has the following library dependencies: os, random, math, shutil.

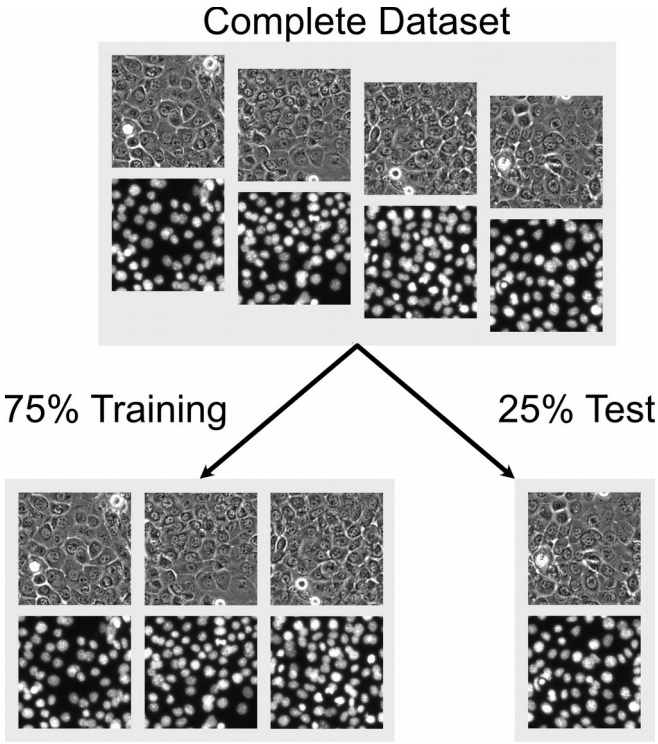

Figure 2. Test/train split. Prior to training the network, we recommend reserving some data for testing. This figure demonstrates a scenario in which 75% of the complete dataset is used for training the neural network, and 25% of the complete dataset is used to make the test set (and not used during training).

4.3 Step 3: Get Data Statistics.

The main script to perform this step is entitled "**run_get_mean_stdev.py**", and can be found in either folder: "3_Get_Data_Statistics" or "All_Files". Users must also modify "**utils.py**", which can be found in either "0_Helper_Functions" or "All_Files".

Once the data is collected, pre-processed, and split into training and test sets, the last step before training is to collect the mean and standard deviation of the input and output data training sets. These values are used to normalize the data which is input into the neural network. In many computer vision applications, normalization/scaling is essential to ensure that the neural network converges in fewer epochs. So, the user will collect the relevant data statistics prior to training the network.

To use this code, the user will need to modify "run_get_mean_stdev.py" and "utils.py" to configure a new experiment and set the path to the training data. One example is provided in "run_get_mean_stdev.py" involving an experiment with keratinocyte-type cells, at 10 magnification, with imaging in the DAPI channel. The function "get_normalization_factors()" in "utils.py" takes in these parameters and determines the path to the image data as set by the user. Then, data is loaded and the relevant statistics are printed out. Users should think carefully how to organize their data, especially if planning future experiments.

After running this script, the users should add the data statistics to the file "utils.py". This file contains all the information about each dataset (from independent data collections), so that statistics and paths can be referenced from one location. The files "data_loader.py" and "models.py" do not require modification.

This script has the following library dependencies: time, numpy. It also utilizes our custom Python files: data_loader.py and utils.py.

The file "utils.py" has the additional dependency: keras.
The file "data_loader.py" has the additional dependencies: os, sys, random, warnings, tqdm, cv2.

4.4 Step 4: Train the Network.

The main script to perform this step is entitled "**run_train.py**", and can be found in either folder: "4_Train_the_Network"or "All_Files".

Having gathered the data statistics, the user may proceed to train the network. This portion of the code compiles the model the user wishes to train, loads in and normalizes the data, and proceeds to train the network. This step can take a long time (several hours on an NVIDIA P100 GPU), so it's helpful to use the most powerful GPU available.

This script requires the user to specify the experimental conditions (as in the previous step, for referencing the correct dataset and normalization factors), the name of a weights file to be saved, the loss function to be used (for example, mean-squared error), and the model type to use (for example, the standard 1-stack U-Net). Please refer to the comments in the script for additional details on setting these parameters.

According to these inputs, the U-Net neural network will be trained using the ADADelta optimization scheme and with early stopping (such that if the validation loss does not decrease in 100 epochs, the training process will end). The script additionally logs the training and validation loss to a CSV file, and saved the finalized model as a YAML file.

Upon completion of this step, users should add the new weight (.h5) file to “utils.py”, so that the weight file can be referenced for each experiment in future steps.
This script has the following library dependencies: time, numpy, and keras. It also utilizes our custom Python files: data_loader.py, models.py, and utils.py.
The file “data_loader.py” has the additional dependencies: os, sys, random, warnings, tqdm, cv2.

4.5 Step 5: Test the Model.

The main script to perform this step is entitled “**run_test.py**”, and can be found in either folder: “5_Test_the_Model”or “All_Files”.

Once the model is trained, the images in the test set may be used to quantify its accuracy. This script loads in the model and weight (.h5) file from the training step, and processes the test set, ultimately printing out the test statistics and saving prediction images into a new folder.

This script requires the user to specify the experimental conditions (as in the previous steps, to normalize the test images appropriately), the path to the output folder into which prediction images will be saved, the loss function to be used (for example, mean-squared error), and the model type to use (for example, the standard 1-stack U-Net). Please refer to the comments in the script for additional details on setting these parameters.

According to these input parameters, the test data will be loaded in, normalized, and passed through the trained model to produce output predictions for each input image. These predictions will be compared to the ground truth images to determine the final accuracy of the model.

This script has the following library dependencies: os, time, numpy, and keras. It also utilizes our custom Python files: data_loader.py, models.py, and utils.py.
The file “data_loader.py” has the additional dependencies: sys, random, warnings, tqdm, cv2.

4.6 Step 6 (optional): Get Additional Prediction Statistics.

The main script to perform this step is entitled “**run_get_signal_stats.py**”, and can be found in either folder: “6_Get_Prediction_Statistics”or “All_Files”.

In previous steps, the user trained and tested the model. In addition to the test loss/accuracy, the user may utilize “run_get_signal_stats.py” to collect additional statistics on the test set, and print them out.

Here, we demonstrate how to determine the Pearson correlation coefficient and mean-squared error statistics on the entire test set, and also on a subset of the test set. This subset is determined by specifying an intensity cutoff value, stored in “utils.py” for the corresponding dataset. By utilizing this value, called “signal_thres”, only image pairs are considered if the ground truth image contains pixel intensity values above the threshold, thereby eliminating images which contain only background. Users will manually determine the appropriate intensity cutoff value for each dataset and modify “utils.py” prior to running this script.

This script requires the user to specify the experimental conditions (as in the previous steps, to determine the correct path to the ground truth data), the path to the folder containing the predicted images, the loss function used (for example, mean-squared error), and the model type used (for example, the standard 1-stack U-Net). Additionally, a path called “save_path” is defined to determine where to write out CSV files, which contain the final results. Please refer to the comments in the script for additional details on setting these parameters.

This script has the following library dependencies: numpy, scipy, sklearn. It also utilizes our custom Python files: data_loader.py, models.py, and utils.py.
The file “data_loader.py” has the additional dependencies: os, sys, random, warnings, tqdm, cv2.
The file “models.py” has the additional dependency: keras.

4.7 Step 7 (optional): Process New Images.

The main code to perform this step is entitled “**Process_Directory.ipynb**”, and can be found in either folder: “7_Process_New_Image”or “All_Files”.

Finally, the trained model may be used to process new, larger input images. This Jupyter notebook demonstrates how to take in an image sequence (again, assuming width/height are multiples of 256 pixels), and produce a sequence of predicted images, patched together from network outputs. In this demo, we leave boundaries of the predicted images unprocessed to emphasize the fact that edge predictions are assumed to be less accurate. However, users may modify

This notebook requires the user to specify the experimental conditions (as in the previous steps), paths to the input images to be processed and the output directory where predictions are saved, a path to the location of the weights (.h5) file, plus information about the neural network (loss function and model type).

The notebook demonstrates how to read in and normalize the input images, apply the trained model to the data in a sliding-window fashion, and compose and save the predicted images. Central regions of each output patch are averaged in this demo; users may consider modifying to code to utilize the data median instead, as is done in [7].

This notebook has the following library dependencies: os, numpy, Pillow, libtiff, sklearn, keras. It also utilizes our custom Python files: models.py and utils.py.

6. References.

[1] Python Software Foundation. Python Language Reference, version 3.5. Available at http://www.python.org

[2] Rueden, C. T.; Schindelin, J. & Hiner, M. C. et al. (2017), "ImageJ2: ImageJ for the next generation of scientific image data", BMC Bioinformatics **18:529**, PMID 29187165, doi:10.1186/s12859-017-1934-z (on Google Scholar).

[3] Schindelin, J.; Arganda-Carreras, I. & Frise, E. et al. (2012), "Fiji: an open-source platform for biological-image analysis", Nature methods **9(7)**: 676-682, PMID 22743772, doi:10.1038/nmeth.2019 (on Google Scholar).

[4] Abadi, Martín, et al. "Tensorflow: A system for large-scale machine learning." *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016.

[5] Chollet, François. "Keras." (2015).

[6] Kluyver, Thomas, et al. "Jupyter notebooks." *a publishing format for reproducible computational workflows* 850 (2016): 87-90.

[7] Christiansen, Eric M., et al. "In silico labeling: predicting fluorescent labels in unlabeled images." *Cell* 173.3 (2018): 792-803.