MATERIALS SEGMENTATION (MatSeg)

USER GUIDE

Tiberiu Stan and Joshua Pritz

tiberiu.stan@northwestern.edu

joshuapritz2020@u.northwestern.edu

Northwestern University

Document Updated on 05/13/2020

Contents

Introduction

Image segmentation is the act of partitioning an image into multiple segments. In materials science, segmentation is often used to identify areas of interest such as precipitates, multi-phase regions, and cracks. Machine learning approaches have shown promise as next-generation tools for image segmentation. In this user guide we showcase “MatSeg”, an open-source Python-based convolutional neural network approach to image segmentation.

Everything needed to understand, install, and run MatSeg is available in the “TS_MatSeg_Share” folder here: https://northwestern.box.com/s/o42lk7z262e3sxhr45rc2bsw20pmvetr.

It is highly recommend that users first read the paper by Tiberiu Stan et al. (reference [1]) which further discusses many of the segmentation, machine learning, and image processing concepts used in this guide. A pdf of the paper is included in the “0UserGuide” folder and can also be downloaded here: https://doi.org/10.1016/j.matchar.2020.110119. Furthermore, an x-ray tomography dataset has been included in the “data” folder and will be used as a tutorial.

We currently don’t have an official way to reference our MatSeg package so we hope you will cite the paper (reference [1]) and the x-ray tomography dataset (references [2,3]). Thank you for using MatSeg!

Environment Configuration

While MatSeg should run on any operating system, we recommend using Mac or Linux if possible. You may encounter hurdles with other operating systems.

MatSeg is a python-based series of codes, so we recommend installing Anaconda (www.anaconda.com) which includes the latest version of python (3.8) as well as other useful packages. We also recommend installing pip (https://pip.pypa.io/en/stable/installing/) which is used to download packages.

Third-party Python packages are required to run many of the scripts needed for network training. In particular, the present training algorithm is based in PyTorch, which is not built into any standard Python distribution. For this reason, we recommend creating a virtual environment in which to install required packages. This can be done by running the following directly into the command line:

```
$ conda create -n <your_venv_name> python=3.8
```

Example for an environment named EnvCNN:

```
$ conda create -n EnvCNN python=3.8
```

This will create a virtual environment with all standard packages from Python 3.8 in your home directory. OPTIONAL: You can also create a virtual environment using the command:

```
$ virtualenv -p python3.8 <your_venv_name>
```

Next, activate your new virtual environment using:

```
$ conda activate <your_venv_name>
```

Next, use the command line to navigate to the “TS_MatSeg_Share” directory.

The remaining packages that are required, as well as some standard packages, are listed in the requirements.txt file. These packages can be installed directly from this file using the pip installer and the following command:

```
$ pip install -r requirements.txt
```

You may get an error regarding the torch package. If so, try installing the PyTorch libraries from the website: https://pytorch.org/

To check whether all required packages in the "requirements.txt" file were successfully installed, run the following command in your virtual environment:

```
$ pip list
```

Dataset Setup

File Structure

The foundation of training a neural network lies in training data. Convolutional neural network-based image segmentation requires sets of raw images and their associated ground truth labels. An example of an image and a label are shown in Figure 1 and are also found in the folder: TS_MatSeg_Share –> data –> tomography –> train.

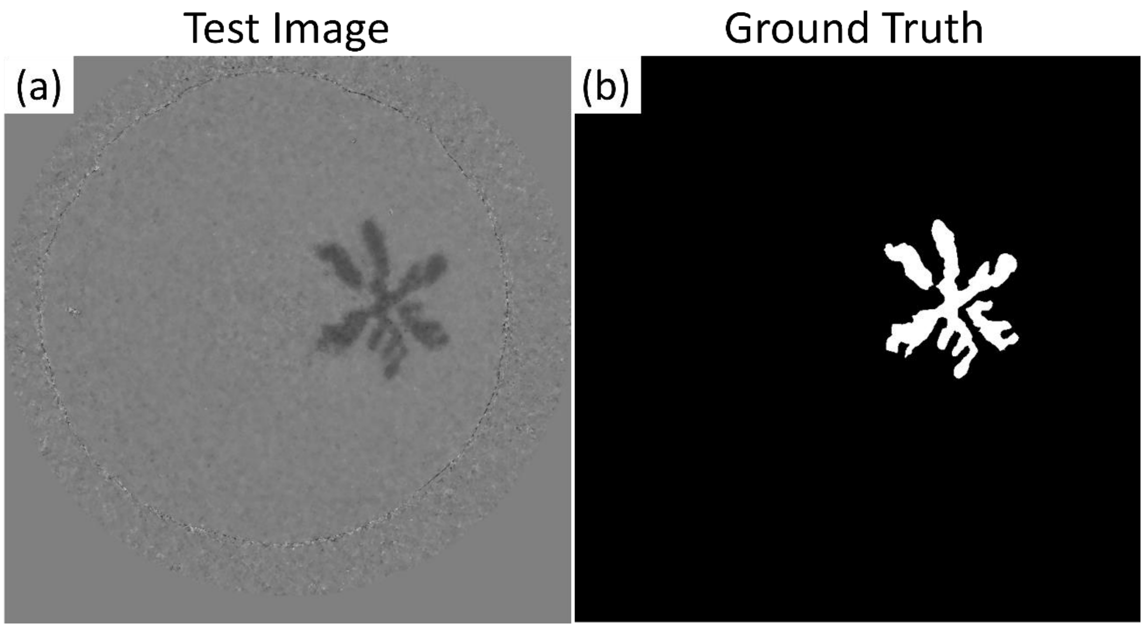

Figure 1. (a) X-ray tomography image showing a dark-grey dendrite cross-section in a light-gray liquid; and (b) the associated two-class ground truth image where dendrite pixels are white, and all other pixels are black.

Generally, datasets must be split into three categories: training data used by the network to adjust weights and biases, validation data used by the network during training to measure loss, and test data used by users to compare the success of different networks. Usually, 60-80% of the available data is used for training, 20-40% is used for validation, and a few image-label pairs are reserved for testing. For example, in the 42-image x-ray tomography dataset available in the "data" folder, 30 images are used for training, 10 images are used for validation, and 2 are saved for testing.

In each folder, there must be further subdirectories corresponding to images and ground truth labels. The file tree for a single dataset should match the following. Note that labels_npy directories do not need to be created at this time and will follow from the creation of numpy arrays.

```
Main_Dataset_Directory
    |--test
        |--images
        |--labels
        |--labels_npy
    |--train
        |--images
            |--a.png
        |--labels
            |--a_L.png
        |--labels_npy
            |--a_L.npy
    |--validate
        |--images
        |--labels
        |--labels_npy
```

Numpy Array Generation

While most image file formats can be handled, it is preferable that files in the image and label directories are of PNG or TIF format. Numpy array labels are arrays whose entries range from 0 to one less than the number of classes in the label, and correspond to each ground truth label. These are necessary for network training. Numpy arrays are generated using the image2npy.py script, which will convert the lables of PNG or TIF format to arrays. It will also create the labels_npy folder in each subdirectory if it does not already exist. If this folder exists and is not empty, the user will be asked whether or not to continue. New arrays will be appended to the existing folder in this case. This script requires specification of the dataset, as well as the subdirectory whose labels are to be converted. It is run with the following code.

```
$ python image2npy.py <dataset> <subdirectory>
```

Example:

```
$ python image2npy.py tomography train
```

The second argument may be train, validate, or test. Immediately following this command, the user will be prompted to input the number of classes in their labels. If any labels have more unique pixel values than the number of classes given, the script will alert the user and identify these images along with their unique pixel values. For the tomography dataset, there are two classes, thus the users will input “2” when prompted.

This script outputs, at its conclusion, whether the numpy arrays where properly normalized, as well as whether or not they have the correct number of dimensions. If either of these are False, troubleshooting is required. This can be aided by examining the script’s output for each array; namely, its unique values and number of dimensions. If the former is not a sequence of integers ranging from 0 to one less than the number of classes, the arrays were not properly normalized. If the latter is not equal to two, the arrays were not flattened.

Optional: Creation of Smaller Datasets

While most neural networks are trained for image segmentation using full-sized images, networks may perform better when training on a larger number of smaller images. The paper by Tiberiu Stan (reference [1]) details this optional image sampling approach. If your original training images are large (greater than 1000 x 1000 pixels), it may be helpful to create training datasets of smaller images which were sampled from your large images. The cropping.py script creates these datasets by randomly cropping squares of a given size from images in the original full-sized set.

This script can be run entirely from the command line. The user must provide the desired dataset from which to crop, the desired size of the resulting images, the number of these images in the training set, and the number of these images in the validation set. The user also has the option to rename the dataset from its default name, which given by

```
<full_sized_dataset>_<image_size>pix<number_of_images>num.
```

The optional rename flag given can be either --rename or simply -r. The image cropping script is run by the following:

```
$ python cropping.py <full_sized_dataset> <image_size> <number_of_training_images>
<number_of_validation_images> --rename
```

This script will admit properly structed subdirectories for both training and validation, cropping images, labels, and numpy arrays. The name of each resultant image, label, and array will append its source image with the x and y coordinates that designate its upper-left corner within this image. Test images, however, are neither cropped, nor transfered to the new dataset through this script. Prior to evaluating this network, test images must be moved manually from the full-sized dataset. The user may also run the command python image2npy.py --help for more information about running the script in the command line.

Neural Network Training

Hyperparameter Configuration

Prior to training a neural network, the user must define several hyperparameters that govern the network's performance. Different choices of hyperparameters can yield distinctly performing networks, as these values are often dataset specific and must be tuned to admit the optimal network. To limit the amount of coding that must be done by the user, as well as exhibit all hyperparameters concisely, all relevant hyperparameters are provided by the user within the config.py module.

It is highly suggested to open the config.py script now to familiarize yourself with it's structure. You will have to edit this script.

This module contains a multi-level dictionary with entries corresponding to each dataset available for network training. Within these entries are further sub-dictionaries with keys corresponding to each hyperparameter version. These versions assign custom values to hyperparameters whose default value the user wishes to overwrite. A short description of each hyperparameter as well as its default value is given below.

- **Model**: This defines the neural network architecture. The scripts are currently equipped to handle three choices of deep convolutional neural networks: SegNet [4], U-Net [5], and PixelNet [6]. SegNet is a smaller network and usually takes the least time to train, while U-Net is a newer architecture that generally produces better results but is more computationally intensive. Unlike all other hyperparameters, the model must be specified in each version since it does not have a default value. We suggest U-Net if this is your first time.

  Default: None.

- **Root**: This defines the primary directory containing all data needed for network training and evaluation. It is required for directing the network to the correct data. To properly define the root, it must be entered as the first argument of the gen-default function within the dataset sub-dictionary, as well as match the name of the key corresponding to that dataset in the master dictionary.

  Default: Must be Specified in Generating Default.

- **Class Number**: The number of classes found in the ground truth labels for the corresponding dataset. The network will learn to partition images into the number of classes specified here. Like the root, this must also be specified in the gen-default function, namely as the second argument.

  Default: Must be Specified in Generating Default.

- **Image Size**: The size of the images on which the network is training must be specified for the network to perform correctly. Note that this does not resize incoming images when evaluating the network and is only necessary in training. Like root and n-class, it must also be specified as the third argument of the gen-default function as a tuple.

  Default: Must be Specified in Generating Default.

- **Batch Size**: This defines the number of images fed to the network for training in a single iteration. Increasing the batch size can decrease the time taken for a single iteration, but concurrently increase the memory needed for a single iteration. For large images, we typically overwrite the default value to be one.

  Default: 4.

- **Optimizer**: The optimizer is responsible for adjusting the internal weights and biases of the network in response to the gradient of the loss function, so as to minimize loss throughout training. The scripts herein support two optimizer choices: Adam and SGD. The right optimizer may yield faster convergence in training, yet this choice is often dataset specific.

  Default: Adam.

- **Learning Rate**: The learning rate scales the gradient of the loss function to determine how much the networks internal parameters are changed through each training iteration. The relative scale of the learning rate depends on the network architecture used. A higher learning rate ($\sim 10^{-2}$) can lead to faster convergence in training, especially for larger images, but can also cause the network to diverge to a poor result.

  Default: 10^{-4}*.*

- **Epochs**: This defines the number of times the training data is passed through the network. Doing so many times is crucial for the network achieving its best result, yet even a single epoch can be very time consuming. Enough epochs must be supplied for the network to converge in training, which is highly dataset dependent.

Default: 40.

- **Data Augmentation**: If True, augmentation applies random transformations to the training data after each epoch. Transformed data activates new weights and biases in the network during training, so that the process of augmentation creates a much more robust set of training data. This can create a more transferable network but can cause it to take longer to converge in training.

  Default: False.

- **Shuffling**: If True, shuffling changes the order in which the training data is passed through the network after each epoch. Like augmentation, this may increase the robustness of the training set, but result in more epochs being needed to reach convergence.

  Default: False.

- **Patience**: This defines the number of epochs after which to scale the learning rate by 0.1 if the loss does not continue to decrease. Such plateaus in training are often indicative of a network beginning to converge. It may be advantageous to reduce the learning rate here so that its internal parameters do not continue to change by such a large margin. To instate this, we often set the patience to 3. By default, it is equal to the number of epochs which implies the learning rate will never be reduced.

  Default: Equal to Number of Epochs.

- **Class Balancing**: If True, the loss function is scaled by the reciprocal of the proportion that each class makes up in the training labels dataset. This process can take up to a minute for large datasets, but aids significantly in pushing network to identify underrepresented classes. This can be very helpful for imbalanced datasets.

  Default: False.

To demonstrate, an example of the "config.py" master dictionary with a single entry corresponding to the tomography dataset is shown below. In the sub-dictionaries corresponding to different versions, herein named v1, v2, v3, and so on, the defined parameters will overwrite their default value given above, while those not present will be assigned their default value.

```
config = {
    'tomography': {
        'default': gen_default('tomography', n_class=2, size=(852, 852)),
        'v1': {'model': 'unet'},
        'v2': {'model': 'unet', 'epoch’: 1},
        'v3': {'model': 'segnet', 'optimizer': 'SGD', 'patience': 3},
        'v4': {'model': 'pixelnet', 'epoch': 30, 'aug': True, 'shuffle': True},
        'v5': {'model': 'unet', 'epoch': 40, 'balance': True},
        'v6': {'model': 'segnet', 'aug': True, 'optimizer': 'SGD', 'lr': 0.01},
        'v7': {'model': 'unet', 'epoch': 4, 'lr': 0.001, 'aug': True}
    }
}
```

For example, v2 assigns the network architecture to be UNet, changes the batch size to 1, and leaves all remaining default values intact. Again, it is crucial that the master configuration dictionary contains an entry corresponding to each dataset on which a network will be trained in the format given above. After doing so, the user is ready to train a network.

Training a Model

The training algorithm is executed by the main.py script, which utilizes each of the hyperparameters identified above. It is run through the command line and requires the following arguments:

- **Dataset**: The dataset on which the network will train. This must be structured as given above, as well as correspond to a correctly formatted entry in the configuration dictionary.
- **Version**: The version defines the hyperparameters to be used in training and must match one of those defined in the configuration sub-dictionary corresponding to the dataset specified. For example, if we were training a network on the tomography dataset, we may specify v2 as its version, as given in the configuration example above.

There are also two optional flags that may be included in the network training command. The --save command directs the algorithm to save the highest performing model in the saved directory named by its dataset and version. The --gpu flag will direct the network's action to a CUDA-enabled GPU device should this be available.

Prior to training, ensure that the virtual environment is activated.

The full command to begin training a network is:

```
$ python main.py <dataset> train <version> --save --gpu
```

Example:

```
$ python main.py tomography train v2 --save --gpu
```

Calling "python main.py –help" provides information as to the usage of this script within the command line.

During training, a number of metrics such as global accuracy, loss, and intersection over union (IoU) scores, are recorded from the networks performance on the training and validation data. If training completes successfully, these metrics are plotted with respect to epoch and saved according to its dataset and version in the "plots" directory. The dictionary responsible for recording these metrics is saved as a .log file in the log directory.

The trained neural network, namely its weights and biases, are explicitly saved as a .pth file in the "saved" directory. It is named according to the dataset on which it was trained, in addition to the hyperparameter version used. While this information is enough to uniquely specify a trained model, the user should be wary of overwriting existing networks by retraining with the same dataset and version, despite changes made to either.

Evaluating a Trained Network

After a neural network is trained, it is often useful to gauge its performance by having it segment images whose ground truths are still readily available, such as the test images mentioned in the Dataset Setup section above. It is important that these images were not included in neither the training nor validation sets so that the user may measure how well the model performs on novel images. Typically, it is also important that the test images be full-sized, regardless of whether the network was trained using full-sized or cropped images.

Evaluating a network also uses the main.py script. Unlike training, the script does nothing to improve the model's internal parameters during evaluation, such as performing dropout or batch normalization. In segmenting the specified images, the network compares its predictions to the corresponding ground truths to compute a more verbose set of metrics. This includes intersection over union (IoU) scores, precision, recall, and F-1 scores for each class, in addition to global accuracy, loss, and mean IoU. To run the script in evaluation mode, the following arguments are required:

- **Dataset**: This identifies the dataset on which the model was trained, as well as the directory where the test images and their labels can be found.
- **Version**: This specifies the version of hyperparameters used to train the network to be evaluated. Not only must this correspond to an entry in the subdictionary named for the dataset, but the combination of the dataset and version must correspond to a trained network found in the saved directory as well.
- **Test Set**: The directory containing the images to be segmented, as well as their labels, must be specified after the test folder flag found in the full command below. Typically, a trained model will be evaluated on the test dataset, but are often evaluated on the validation set as well. If no set is specified, the test folder flag will apply the network to the test dataset by default.

  Default: test.

There is one required flag for the evaluation command: --test-folder. This will instruct the script to store the following argument as the dataset whose images are to be segmented. This dataset must be correctly formatted as given in Dataset Setup above. If no subdirectory is specified, the flag will search for the test directory by default.

Optional flags may also be given for this command. Like the training process, evaluation may be directed to a CUDA-enabled GPU if available. This is indicated by the --gpu flag. To create a visual aid in gauging network performance, there is also the option to create an overlay of the network's prediction with the corresponding ground truth. In the case of two segmentation classes, this overlay colors false positive pixels in green and false negative pixels in pink. Correctly identified pixels retain the same color as in their label image. For more than two classes, misidentified pixels are simply colored pink, while correctly segmented regions retain the same color as in their label. To create and save these overlays, the --overlay flag must be provided.

The full command to evaluate a trained model is given by the following:

```
$ python main.py <dataset> evaluate <version> --test-folder <folder> --overlay --gpu
```

Example:

```
$ python main.py tomography evaluate v7 --test-folder test --overlay --gpu
```

As in training, the second positional argument, evaluate, given above is necessary for this process.

Note here we create overlays but run evaluation on our system's GPU. The network's segmentations are saved in the test folder specified within a directory titled predictions. Moreover, if overlays are created, these are also saved in the test folder in a directory titled overlays. Both predictions and overlays can be found in a further subdirectory named for the hyperparameter version and network architecture used in their respective directories.

Evaluating a trained neural network in no way changes the model's internal parameters, but merely provides the user an indicator as to the network's performance on novel and/or full-sized images. The network's predictions and overlays, as well, provide a visual benchmark for the model's quality. If you are satisfied with the network's performance, you may next apply the network to a full dataset of raw images for which ground truth labels do not exist.

Applying a Trained Neural Network

The work of the last four sections pays off here, in applying your trained neural network to a dataset of raw images. These images, of course, do not have corresponding ground truth labels or numpy arrays, and must therefore be structed differently than the datasets used for training. In particular, the raw images must be contained in a directory located in the data folder. Any subdirectories or non-image type files in this dataset will cause segmentation to crash. Hence, the set of raw images must be structed by the following:

```
|--data
   |--<image_folder>
      |--a.png
      |--b.png
      |--c.png
          .
          .
          .
```

The segmentation of these images using a previous trained model is performed by the apply.py script. While PNG or TIF format images are preferred, this script can handle any common image format. It requires the following three arguments.

- **Dataset**: This specifies the dataset on which the neural network to be applied was trained. It must correspond to a correctly formatted set contained in the data directory.
- **Version**: This gives the hyperparameter version with which the network to be applies was trained. Along with the dataset indicated above, it must correspond to a saved model as well as an entry in the sub-dictionary for that dataset found in the configuration script.
- **Raw Image Dataset**: This specifies the set of raw images to be segmented by the network identified by the dataset and version above. It must be formatted as given above, contain only image-type files, and contain no subdirectories.

There is one optional flag that may be included in the image segmentation command. To direct the segmentation process to an available CUDA-enabled GPU, the user may include --gpu. The full command is:

```
$ python apply.py <dataset> <version> <image_folder> --gpu
```

As in using all scripts discussed above, ensure that your correctly configured virtual environment is active for running the application program. For example, if we were to segment a set of images named tomography_full with a neural network trained on the tomography dataset using v7 hyperparameters, the command would be:

```
$ python apply.py tomography v7 tomography_full --gpu
```

This script will save the model's segmentations in a subdirectory of the data folder titled: <image_folder>_predictions. In this subdirectory, a folder named for the hyperparameter version and training dataset will contain the segmented images. Segmenting images using a trained network will take roughly ten seconds per image, but is highly dependent on computational resources.

List of Useful Commands

Below is a summary of the commonly used commands for MatSeg.

Command	Purpose
`$ conda activate <your_venv_name>`	Activate your virtual environment. It is good practice to do this before running any other commands.
`$ python image2npy.py <dataset> <subdirectory>`	Create numpy arrays for new ground truths.
`$ python main.py <dataset> train <version> --save --gpu`	Train a network on the images and associated ground truths in the <dataset> folder, using the <version> specified in the config.py file.
`$ python main.py <dataset> evaluate <version> --test-folder <folder> --overlay --gpu`	Evaluate the <version> of the network trained on <dataset> by applying it to the images in the <folder>. Use this if <folder> has images and ground truths.
`$ python apply.py <dataset> <version> <image_folder> --gpu`	Apply the <version> of the network trained on <dataset> to the images in the <image_folder>. Use this if you don't have ground truths for the images in <image_folder>.

Acknowledgements

Tiberiu Stan and Joshua Pritz acknowledge the important contributions to this work by Bo Lei and Elizabeth Holm (Carnegie Melon University), Marta Garcia-Martinez (Argonne National Laboratory), Michael D'Mello (Intel), and Peter Voorhees (Northwestern University). The x-ray tomography dataset was collected at the Argonne National Laboratory Advanced Photon Source, Beamline 2-BM with financial support by the U.S. Department of Energy's Office of Science under grant number DE-FG02-99ER45782. The data analysis and machine learning codes were developed with support by the U.S. Department of Commerce, National Institute of Standards and Technology as part of the Center for Hierarchical Materials Design (CHiMaD) under award 70NANB14H012. This work was also partly supported by the Northwestern-Argonne Institute of Science and Engineering (NAISE) Summer Internship Program.

References

[1] T. Stan, Z.T. Thompson, P.W. Voorhees, Optimizing Convolutional Neural Networks to perform Semantic Segmentation on Large Materials Imaging Datasets: X-ray Tomography and Serial Sectioning, Mater. Charact. 160 (2020). https://doi.org/doi.org/10.1016/j.matchar.2020.110119.

[2] T. Stan, Z.T. Thompson, P.W. Voorhees, Raw Images for Semantic Segmentation of Dendrites via Machine Learning, (2019). https://doi.org/10.18126/M2RM08.

[3] T. Stan, Z.T. Thompson, P.W. Voorhees, Ground Truths for Semantic Segmentation of Dendrites via Machine Learning, (2019). https://doi.org/10.18126/M2W93J.

[4] V. Badrinarayanan, A. Kendall, R. Cipolla, SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation, IEEE Trans. Pattern Anal. Mach. Intell. (2017). https://doi.org/10.1109/TPAMI.2016.2644615.

[5] O. Ronneberger, P. Fischer, T. Brox, U-net: Convolutional networks for biomedical image segmentation, Lect. Notes Comput. Sci. (Including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics). 9351 (2015) 234–241. https://doi.org/10.1007/978-3-319-24574-4_28.

[6] A. Bansal, X. Chen, B. Russell, A. Gupta, D. Ramanan, PixelNet: Representation of the pixels, by the pixels, and for the pixels, (2017).