Table of Contents

Demo 1 Information *2*

Local Environment 2

HPC Singularity Container 2

Local Singularity Container 2

Local Docker Image 3

Inference Quick Start *3*

Relevant Package File Structure for Inference 3

Input directory required structure for inference 4

Basic Function Call - Inference 4

Outputs 4

Inference Argument Information *5*

Basic Options 5

Corrections Options 5

Image Preprocessing Options – Mode 1 – Not Recommended 6

Image Preprocessing Options – Mode 2 7

Quality Check Options 8

Training Quick Start *8*

Relevant Package File Structure for Training 9

Input Dataset Directory Required for Training 9

Basic Function Call – Training 9

Outputs 10

Training Argument Information *10*

Basic Options 10

Talos Options 11

Universal Image Preprocessing Options 13

Image Preprocessing Options – Mode 1 – Not Recommended 14

Image Preprocessing Options – Mode 2 15

Demo 1 Information

As of December 22, 2021 there are four ways to run a demo of the software:

1. Local environment
2. HPC singularity container
3. Local singularity container
4. Local docker image

The primary goal is to run inference on MRI scans from three mice: two from Jax, one from UF. There are a total of 11 scans. The dataset includes masks output from the default set of parameters, to provide an idea of what the final output should look like. Running clean_test_dataset.sh will reset test_dataset/ to its initial state. It is necessary to do this before each run. The following sections will go into greater detail about each of the four methods to run the demo.

Local Environment

1. Clone repo at https://github.com/TheJacksonLaboratory/seg-for-4modalities
2. Create a virtual environment, install packages as specified in requirements.txt
3. Activate the virtual environment and navigate to seg-for-4modalities/
4. Take a look at pre-generated masks in demo/test_dataset/
5. Clean dataset via ./demo/clean_test_dataset.sh to reset the dataset
6. Run inference with the following command:
   python3 msUNET/segment_brain.py -i demo/test_dataset/
7. View the results in demo/test_dataset/
8. If desired, alter options in command line – refer to 'inference argument information'
9. Clean dataset via ./demo/clean_test_dataset.sh to reset the dataset before each inference run

HPC Singularity Container

1. Login to Winter
2. cd projects/compsci/neural_imaging_ssif_data/aim-2-demo-1/msUNET_release_pep8
3. Take a look at pre-generated masks in test_dataset/
4. Clean dataset via ./clean_test_dataset.sh to reset the dataset
5. Run inference with the following command:
   sbatch segment_brain_hpc.sh
6. View the results in test_dataset/
7. If desired, alter options by way of segment_brain_hpc.sh, line 13
8. Clean dataset via ./clean_test_dataset.sh to reset the dataset before each inference run

Local Singularity Container

Running inference using singularity locally will only work if using a Linux machine or VM.

1. Clone repo at https://github.com/TheJacksonLaboratory/seg-for-4modalities
2. Download singularity image from Sumner/Winter:
   /projects/compsci/neural_imaging_ssif_data/containers/tensorflow2003.sif\
3. Place the singularity image 'tensorflow2003.sif in seg-for-4modalities/msUNET/

4. Take a look at pre-generated masks in demo/test_dataset/
5. Clean dataset via ./demo/clean_test_dataset.sh to reset the dataset
6. Run inference with the following command:
   python3 msUNET/segment_brain.py -i demo/test_dataset/
7. View the results in demo/test_dataset/
8. If desired, alter options in command line – refer to ‘inference argument information’
9. Clean dataset via ./demo/clean_test_dataset.sh to reset the dataset before each inference run

Local Docker Image

1. Download the docker image ‘seg-for-4modalities-demo1.tar’ from Sumner/Winter: /projects/compsci/neural_imaging_ssif_data/containers/seg-for-4modalities-demo1.tar
2. Load the image via ‘docker load {path to seg-for-4modalities-demo1.tar}’
3. Run a container from the image
4. Run a command line interface in the container
5. cd msUNET_release_pep8/
6. Take a look at pre-generated masks in test_dataset/
7. Clean dataset via ./clean_test_dataset.sh to reset the dataset
8. Run inference with the following command, from msUNET_release_pep8/:
   python3 msUNET/segment_brain.py -i test_dataset/
9. View the results in test_dataset/
10. If desired, alter options in command line – refer to ‘inference argument information’
11. Clean dataset via . /clean_test_dataset.sh to reset the dataset before each inference run

Inference Quick Start

Relevant Package File Structure for Inference

```
msUNET-RAM ->  train
             -> predict -> core
                        scripts -> rbm.py
                                   156mice_noregwarp_norotatedaffine_inmemory.hdf5
               segment_brain.py
```

For inference, the relevant handler function is contained in segment_brain.py. It requires the contents of predict/ to function. The only user specified file here is the model. The model defaults to 156mice_noregwarp_norotatedaffine_inmemory.hdf5, the current best stable model. If the user would like to specify a different model, there are a few things to take into consideration. First, ensure that the model file is located within predict/scripts. Second, pass the name of the model to the ‘-m’ argument (detailed later). Third, ensure that the input patch dimensions are correct for the new model. If they are incorrect, inference will fail, printing out a message letting you know what the current dimensions are, and what they should be.

Input directory required structure for inference

```
*Dataset_name* -> *mouse1_name* -> *modality_1* -> *mouse_1_modality_1*.nii
                                -> *modality_2* -> *mouse_1_modality_2*.nii
                                …
               -> *mouse_2_name* -> *modality_2* -> *mouse_2_modality_1*.nii
               …
```

To summarize, the input dataset must be structured in the following way. Inside the dataset directory, there should exist one directory for each mouse. Inside each mouse directory should be one directory per modality. Inside each modality directory should be exactly one file, the raw data file in .nii format, corresponding to the mouse and modality specified in the above folders. The file itself should have no '.' In its name, with the exception of the .nii suffix. The .nii file can be named whatever is most convenient for the user – that name will be used as a base for creating all other output files.

Basic Function Call - Inference

1. cd *install_directory*
2. python3 segment_brain.py -i *input_folder_name*

Example:

0. Assuming there exists the following file structure…
1. cd Documents/msUNET-RAM/
2. python3 segment_brain.py -i test_dataset

The basic function call requires only one input, the location of the dataset you would like to run inference on. All other options are not required and will remain at their default values unless otherwise specified. Ensure that the '/' after the dataset name has been removed. It is possible that leaving it in will cause an error. It is recommended that the dataset directory be backed up before running inference. Although the program will create a backup of the raw data file before it runs, thus limiting data loss, the files left behind in case of a crash will not be conveniently placed to run again.

Outputs

All outputs are contained to the input dataset folder. At the top level, you will find a file called quality_check.txt, if slice quality checks are enabled. This file contains information about all slices which have been flagged for manual review for all mice/modality combinations in the dataset directory. The output files for each mouse/modality combination can be found in the corresponding mouse/modality directory, right next to the input file. The files always generated, no matter the options, are the predicted mask and a likelihood map. Both are in '.nii' format. Other files will be generated depending on the options selected at runtime. For example, if Z-Axis correction is performed, the Z-Axis corrected data will be saved for reference.

Inference Argument Information

Basic Options

-i, --input: Dataset directory containing the above specified file structure. Must contain either the full path to the dataset directory, or the name of a dataset directory located in msUNET/.
EX: -i test_dataset/

-m, --model: Filename of the model to be used for inference. Must be located in msUNET/predict/core. In testing, all models were contained in '.hdf5' files. It is possible other Keras saved model file types could work, but their use is not recommended. The model chosen determines the required value of one other input parameter, -ip –image_patch.
EX: -m 156mice_noregwarp_norotatedaffine_inmemory.hdf5

-th, --threshold: Set the threshold above which the model output is categorized as brain. Float [0,1]. After running inference on many overlapping patches, the model creates a score map of the same dimensions as the input image. Associated with each pixel is a score on the interval [0,1]. The threshold value determines the score breakpoint, below which is not brain, above which is brain. The model is validated with a threshold of 0.5. It is unlikely it will need to be changed.
EX: -th 0.5

-cl, --channel_location: Whether input channels are the first or last dimension. String, either 'channels_first' or 'channels_last'. This option pertains to the structure of the image data. In the case of Jax and UF data, the dimension corresponding to channel appears last. It is possible that other data could have this first by default. It is unlikely that users at one institution will have to change this value once it has been set, assuming their data conforms to a single standard. If it is set incorrectly, the inference will fail.
EX: -cl channels_first

Corrections Options

-zc, --z_axis_correction: Whether to perform z-axis correction on input images before inference. String, either True or False. If true, z-axis correction will be performed on raw images before inference. Z-Axis correction roughly determines brain region in all slices, then normalized intensity by slice such that the mean intensity in the preliminarily selected brain region is a constant. If Z-Axis corrections are to be applied, inference will take around twice as long, as inference is being run twice: once to determine a rough brain region for z-axis correction, once to create the final mask. If Z-Axis corrections are applied, additional output files will be created. The additional output files are the preliminary mask created to determine rough brain region for Z-Axis correction and its corresponding likelihood map, a plot of the intensity by slice before and after Z-Axis correction, and the Z-Axis corrected data file. Use Z-Axis correction if there is a large difference in intensity between slices in a single image.
EX: -zc True

-yc, --y_axis_correction: Whether to apply Y-Axis correction to raw images before inference or not. String, True or False. If True, Y-axis corrections will be applied to raw images before inference. Y-Axis correction normalizes image intensity within a single slice along the vertical axis. Applying Z-axis corrections should not radically increase computation time. If Y-Axis corrections are applied, additional output files will be created. Those additional output files are the mask used to select pixel to apply Y-axis correction to (if applicable), and the Y-Axis corrected data file. Use Y-Axis correction if there is a dip in intensity across individual slices along the vertical axis.
EX: -yc True

-ym, --y_axis_mask: Whether to use a mask to determine approximate foreground areas before applying Y-Axis corrections. Boolean. This option is only relevant if Y-Axis corrections have been enabled. If True, Y-Axis corrections will only be applied to the foreground of an image, estimated by Otsu binarization. It is possible that this could reduce the increase of noise due to applying Y-Axis corrections to low signal regions outside the brain.
EX: -ym True

-nt, --normalization_mode: Determines which normalization mode raw data should be subjected to before being input into inference mechanism. String, either ‘by_slice’ or ‘by_image’. When image data is input into neural networks, it is normalized to the range [0,1]. Since we are working with 2D slices of 3D data, there are two schemes by which that normalization can occur, either over the entire 3D image or over each 2D slice. If ‘by_image’ is set, the model will normalize an entire 3D image. If ‘by_slice’ is selected normalization will occur over each 2D slice. In general, models are trained using ‘by_image’ normalization, so that mode is recommended. It is possible to use ‘by_slice’ normalization to limit the effect of outlier pixels. In some cases, images have a small number of pixels with an intensity many standard deviations above the mean. When normalized, those large intensity pixels will squish the rest of the data very close to zero. Normalizing by slice in this case will limit this damaging effect to a single slice. By slice normalization can also serve as an alternative to Z-Axis correction. If data has vastly different intensity by slice, consider trying both.
EX: -nt by_image

Universal Image Preprocessing Options
-ip, --image_patch: Dimensions of the image patches into which an image is broken for inference. Integer, [1,min(Slice_Dimension-1)). This value is defined by the model to be used for inference. In general, the model filename will have some information about the image patch size used during training. The training value must be replicated here. Common values taken by image patch are 256, 128, and 64.
EX: -ip 128

Image Preprocessing Options – Mode 1 – Not Recommended

-ns, --new_spacing: A multiplicative factor by which images are divided before patching and inference are performed. Three floats on the interval (0,1]. Understanding this option requires

some background on the inference method used by this program. Images are not passed through the model whole. Instead, they are chopped up into many smaller, overlapping patches. Those patches are fed into the model, then reassembled into something the same dimension as the input image. Due to the structure of our model, the size of the patches into which the images are divided must be constant for a given model. Since the dimensions of MRI images can vary significantly by modality, it is possible that a patch size that leads to a reasonable number of patches in a high-resolution anatomical MRI image would be larger than the entire input image for the corresponding fMRI image. Since this would not work, it is essential to increase the size of small images such that the constant patch size works well will all relevant modalities. Setting the value of 'new_spacing' is one such way to accomplish that task. The input dimensions of a given MRI image are divided by one of the three values passed to this option. Consider an fMRI image with dimensions [64, 64, 17]. Say the original spacing was [1, 1, 1]. If we were to pass new spacing options of [0.25, 0.25, 1], the image sent to patching would be of dimensions [256, 256, 17]. This expansion is accomplished by linear interpolation. The new image size would now allow for reasonable use of a 128 by 128 image patch, which is larger than the original image size.
NOTE: This option is not recommended. It is the primary mechanism by which the original CAMRI at UNC model adjusted image dimensions, but more clear and consistent options have since been developed. We leave this option here for completeness, but it is not recommended. Consider using 'target_size' instead!
EX: -ns 0.08 0.08 0.76 (typical values for 17 slice Jax images)

-is, --image_stride: Distance along either axis image patches translate. Integer [1,min(Slice_Dimension-1)). This value defines the amount of overlap neighboring patches will have. On average, we have seen that increasing overlap leads to increasing performance. Correspondingly, increasing overlap will also increase inference time. Common stride values are 0.5*image_patch and 0.25*image_patch. For an image patch 128 pixels by 128 pixels, the most common stride value is 32.
EX: -is 32

Image Preprocessing Options – Mode 2

-ts, --target_size: Whether to set all input images to have a single constant input dimension before inference. String, True or False. This option serves the same purpose as new_spacing but allows for more transparent behavior and consistent operation. If True, the algorithm will use target-size based resampling instead of new-spacing based resampling. In target-size based resampling, one number is provided to a function. That function sets one dimension of an input image to that value. The second dimension is set to preserve the aspect ratio of the input image. The number of slices is not adjusted. This is the recommended mode of image resampling currently.
EX: -ts True

-cs, --constant_size: Size to which all input images should be adjusted before they are sent to inference. Three integers, [1,inf). An alternative method of increasing the size of images to ensure successful patching. Determines the dimensions to which images will be resampled for

inference. The second input dimension is adjusted such that aspect ratio will be preserved. It is generally recommended to set this value to roughly twice the image patch size defined by the model of choice. It is also recommended that the value not be below the largest dimension of any input image.
EX: -cs 256 256 17

-ufp, --use_frac_patch: Whether to define patch size as a fraction of input image size. Boolean. If true, enables use of fractional patch sizes. Fractional patch sizing is an alternative to the traditional user-defined patch size. If the model was trained using fractional patch size, then this value should be set to true.
EX: -ufp True

-fp, --frac_patch: Dimension of images patches on which inference is run, as a fraction of resampled input image dimensions. Float, (0,1). This value is defined by the model used for inference. This value is only checked if use_frac_patch is true. Currently, this value is not essential, and input dimension information is input via image_patch -ip.

-fs, --frac_stride: Distance neighboring image patches translate within a slice as a fraction of resampled input image dimensions. Float, (0,1). This value defines the amount of overlap neighboring patches will have. On average, we have seen that increasing overlap leads to increasing performance. Correspondingly, increasing overlap will also increase inference time. Common fractional stride values are 0.75, 0.5, and 0.25.

Quality Check Options

-qc, --quality_checks: Whether to perform by-slice quality checks. Boolean. If true, the program will save quality_checks.txt inside the input dataset directory. It contains a list of slices that have been algorithmically determined to be worthy of manual review. It contains the filename, the slice number, and what caused the flag to be raised. Currently, this raises flags based on an unvalidated, unsupervised method of outlier detection. It is not to be relied upon until it is expert validated.
EX: -qc True

-st, --low_snr_threshold: Threshold below which a low signal to noise ratio flag will be added to quality_checks.txt for a given slice. Float, [1,inf). While most features associated with raising flags for individual slice review are based on properties of the predicted mask, if the signal to noise ratio of the input image is low it is more likely that the predicted mask will be poor. If quality checks are enabled, low signal to noise ratio flags will be thrown when
$Image\ Center\ Intensity < Image\ Edge\ Intensity * Low\ SNR\ Threshold$
EX --st 3.5

Training Quick Start

Relevant Package File Structure for Training

```
msUNET-RAM -> predict
              train -> augmentation.py
                       image.py
                       metrics.py
                       model.py
                       setup.py
                       train.py
                       util.py
                       start_models -> rat_brain-2d_unet.hdf5
```

For training, the relevant handler function is contained in train.py. It requires the contents of train/ listed above alongside it in order to function. The only user specified input file for training is the initial model from which transfer learning is to be based. Currently, the default expects that the CAMRI at UNC 2D rbm model, named rat_brain-2d_unet.hdf5, will be contained in train/start_models, as indicated above. This location and file name can be changed via the input argument -path, as detailed in the options section below.

Input Dataset Directory Required for Training

```
*Dataset_name* -> *training_data* -> *mouse_1_modality_1_data*.nii
                                     *mouse_1_modality_2_data*.nii
                                     …
                                     *mouse_2_modality_1_data*.nii
                                     *mouse_2_modality_1_data*.nii
                                     …
                  *mask_data*   ->   *mouse_1_modality_1_mask*.nii
                                     *mouse_1_modality_2_mask*.nii
                                     …
                                     *mouse_2_modality_1_mask*.nii
                                     *mouse_2_modality_1_mask*.nii
                                     …
```

For training, the input data structure is different than for inference. A dataset directory contains two subdirectories, containing to training data and manually annotated masks respectively. It is critical that the data be in the same order (alphanumerically) in both folders, i.e, the first file in the training_data directory must correspond to the first file in the mask_data directory. It is recommended that a clean backup of training data is kept elsewhere, as the dataset directory is manipulated before training. It is possible that this manipulation, if interrupted, could cause issues with individual samples.

Basic Function Call – Training

```
cd *install_directory*/train
python3 train.py -tpath *training_data_path* -mpath *mask_data_path*
```

EX:

1. cd Documents/msUNET-RAM

2. python3 train.py -tpath test_dataset/data -mpath test_dataset/masks

The basic functional call requires two inputs, corresponding to the directory containing training and corresponding mask files. Ensure that there is no '/' after either of the dataset directory names. All other options have reasonable defaults and are thus not required. Those inputs are specified below.

Outputs

Training outputs come in the form of an experiment directory. These experiment directories will be output in msUNET/train. By default, they will be named experiment*start_timestamp*. It is possible to change the experiment directory name via a command line argument. Upon successful completion, the experiment directory should contain four or more items. First, *complete_timestamp*.csv. This file contains summary results from each of the models trained during the Talos scan. If only one permutation of variables is requested, this file will contain only one row of data. Second is modality_results_*complete_timestamp*.csv. This file (or files) contains information on the performance of the model on each of the four relevant modalities for each of the models trained during the Talos scan: anatomical, DTI, fMRI, and NODDI, broken down by training and validation set. Each Talos hyperparameter permutation requested will create another of these files. Third: model_deployexperiment*write_timestamp*.zip, which is a Talos.Deploy object. Information about this object type can be found at the Talos package GitHub, located here: https://github.com/autonomio/talos/blob/master/docs/Deploy.md . In summary, the archive contains eight files. The following will describe them briefly, but more details can be found at *details.txt contains information about the hyperparameter search method used for the Talos scan. *model.h5 and *model.json contain the best model's weights and architecture, respectively. *params.npy contains the best value of the Talos hyperparameters selected for use in the final model. *results.csv contains very similar contents to the first ouput mentioned in this section, plus more detailed information about scan runtime. *x.csv and *y.csv contain randomized dummy data, as Talos is not equipped to handle output of image data. README.txt contains information about how to restore the file. Since the recommended model input format for inference is .hdf5, it is recommended that the Talos Deploy object be used to create a single Keras model file via Talos.Restore. Documentation for that process can be found here:
https://github.com/autonomio/talos/blob/master/docs/Restore.md. The fourth and final output file in the experiment directory is talos_scan_history_*timestamp*.csv. One such file is created per hyperparameter permutation requested of Talos. Each provides by-epoch information about a given training instance, including various relevant metrics.

Training Argument Information

Basic Options

-path, --initial_model_path: Path to the model from which initial weights are to be pulled. String. Points to location, including filename, of start model. Default location is msUNET-RAM/train/start_models/rat_brain-2d_unet.hdf5.

-tpath, --training_data_path: Path to directory containing training data. String. Points to location of training data, does not include filenames. Ensure there is no '/' after the final directory. The directory should contain both training and validation data. All training data should come before all validation data alphanumerically. Data files should be in .nii format. It is critical that the alphanumeric order of data files matches that of mask files. It is recommended that this directory be backed up before training, as it is possible that existing files will be manipulated, or additional files will be written. Must contain the same number of files as the mask directory.

-mpath, --mask_data_path: Path to directory containing annotated mask data. String. Points to location of mask data, does not include filenames. Ensure that there is no '/' after the final directory. The directory should contain both training and validation masks. All training masks should come before all validation masks alphanumerically. Mask files should be in .nii format. It is critical that the alphanumeric order of mask files matches that of data files. It is recommended that this directory be backed up before training, as it is possible that existing files will be manipulated, or additional files will be written. Must contain the same number of files as the training directory.

-v, --validation_split: Fraction of samples to be used as validation. Float [0,1). Which samples are selected to be used as validation is determined as follows: the number of samples is determined from number of files in the dataset directories; we take the floor of that value multiplied by the validation split value. It is assumed that all validation samples come after all training samples alphanumerically.

-n, --experiment_name: Name to be used for the output experiment directory. String.

-mw, --modality_weight: Name of one of the four modalities that should be weighted more heavily in the loss function. String, choice of ['anatomical', 'dti', 'fmri', 'noddi']. If selected, the given modality will be weighted some factor more heavily than the three others. The factor by which it is weighted more heavily is defined by -wf, --weight_factor. If -wf is set to 1, no matter the value for -mw, all samples will be weighted identically.

-wf, --weight_factor: Factor by which the samples corresponding to the chosen modality should be weighted more heavily than the remaining samples. Float [0,inf). If set to 1, all modalities will have identical sample weights. If set greater than 1, errors in samples corresponding to the modality selected by -mw will be penalized more heavily than errors in samples corresponding to other modalities. If set less than one, errors in samples corresponding to the modality selected by -mw will be penalized less heavily than errors in samples corresponding to other modalities.

Talos Options ****INCOMPLETE– CURRENTLY CHANGE DEFAULTS FOR EACH RUN**

-talos, --talos_params: Dictionary containing one list each for a selection of hyperparameters from which permutations will be created for Talos scans. The dictionary takes the following form:

```
'epochs':[10, 20],
'losses':[dice_coef_loss],
'optimizer':[Adam],
'lr':[.001,  0.005],
'batch_size':[128, 256],
'finalActivation':['sigmoid'],
'dropout':[0.5, 0.25],
'which_layer':[19, 5],
'patch_dimensions':[128],
'patch_stride':[32],
'scoreTrainable':[True, False],
'expandingBatchNormTrainable':[True, False],
'contractingBatchNormTrainable':[True, False],
'upsamplingTrainable':[True, False],
'upsamplingBatchNormTrainable':[True, False],
'pca_only':[False]
```

In the following section, a brief interlude to describe each of these hyperparameters.

1. epochs – Int, number of epochs for each training instance in the scan
2. losses - name of a default keras loss function or dice_coef_loss, a custom loss based on dice coefficient. It is highly recommended to use dice_coef_loss
3. optimizer – name of a default keras optimizer function. It is highly recommended to use Adam
4. lr – float, learning rate
5. batch_size – int, batch size. In this case, batch size does not refer to the number of scans to be processed at the same time. Instead, a sample is considered a single slice within a scan. In the case of JAX data, each scan will have 17 slices. Thus, each mouse/modality combination will be cast as 17 samples for the purposes of batch size determination
6. finalActivation – string, name of a default Keras activation function. It is highly recommended to use sigmoid
7. droupout – float [0,1), value used across all dropout layers
8. which_layer – int [0,19]. Determination of which layers of start model should be trainable during transfer learning. There are a total of 19 convolutional layers in the model, and it is possible to specify the layer below which no additional layers are trainable. For example, a which_layer value of 19 allows all convolutional layers to train. A which_layer value of 0 would allow no convolutional layers to train.
9. patch_dimensions – int, divisible by 32. Dimensions into which individual slices are split into for training if use_frac_patch argument is set to False. It is not recommended to use this option. See inference image preprocessing options – mode 1 for discussion. If not using frac_patch, is it highly recommended to supply only one value for patch_dimensions. For each value in patch dimensions, the dataset must be recreated, as the input data itself will change. As such, two datasets will both live in memory

throughout the training process for both. While this should not be an issue for small datasets on high performance computing clusters, memory usage can become prohibitive quickly

10. patch_stride – int, [1,max(min_image_dimensions)-1]. Amount, in pixels, by which image patches translate between neighboring samples if use_frac_patch is set to False. It is not recommended to use this option. See inference image preprocessing options – mode 1 for discussion. If not using frac_patch, is it highly recommended to supply only one value for patch_stride. For each value in patch stride, the dataset must be recreated, as the input data itself will change. As such, two datasets will both live in memory throughout the training process for both. While this should not be an issue for small datasets on high performance computing clusters, memory usage can become prohibitive quickly
11. scoreTrainable – Bool. Whether the final convolutional layer that computes score is trainable
12. expandingBatchNormTrainable – Bool. Whether the batch normalization layers on the expanding side of the U-Net model are trainable
13. contractingBatchNormTrainable – Bool. Whether the batch normalization layers on the contracting side of the U-Net model are trainable
14. upsamplingTrainable – Bool. Whether the convolutional layers associated with up sampling in the expanding arm of the U-Net model are trainable
15. upsamplingBatchNormTrainable – Bool. Whether the batch normalization layers associated with up sampling convolutional layers are trainable
16. pca_only – Bool. If True, performs PCA using intermediate representation instead of training as normal.

Universal Image Preprocessing Options

-er, --enable_augmentation: Whether image augmentation is to be used. Bool. If true, dataset will be augmented before training if there are no files in either the training or mask directory containing the substring ‘augmented’. There are four types of augmentation possible; rotation, affine, noise, and contrast. The following five arguments control how augmentation is done.

-ia, --in_place_augmentation: Input argument used to switch between two augmentation methods. Boolean. If False, will use a basic augmentation scheme in which every image has rotation and affine transformations applied, assuming that the corresponding option is selected (--enable_rotation and –enable_affine below). If both are selected in this scheme, then three augmented images will be created for each raw image, one with only rotations applied, one with only affine transforms applied, and one with both applied. It is not recommended to use this augmentation scheme. Thus, it is recommended that this argument remain True. If true, all four augmentation methods detailed below are available, as is the single controlling parameter augmentation threshold.

-at, --augmentation_threshold: Determines the extent to which the dataset will be augmented. Float [0,1). This value serves two purposes. It first serves as the chance that an individual image will be selected for augmentation. If 0, no images will be augmented. Second, it serves to control the number of manipulations that are applied to each image. Lower values correspond to a higher chance of having a larger number of manipulations applied, while higher values correspond to a smaller number of manipulations. To summarize, high values lead to fewer images augmented, but with more manipulations applied per augmented image. Low values lead to many augmented images, but with a smaller number of manipulations applied per image.

-er, --enable_rotation: Use rotation augmentations. Bool. If true, there is a chance rotation will be among the augmentations included in the dataset

-ea, --enable_affine: Use affine augmentations. Bool. If true, there is a chance affine transformations will be applied to images in the dataset. The possible affine transformations include shear and scaling

-en, --enable_noise: Use noise augmentations. Bool. If true, there is a chance additive gaussian noise will be applied to some images in the dataset

-ec, --enable_contrast_change: Use contrast augmentations. Bool. If true, there is a chance images will have adaptive histogram equalization as a method to suppress contrast applied for augmentation purposes

Image Preprocessing Options – Mode 1 – Not Recommended

-hspace, --horizontal_interpolation_spacing: A multiplicative factor by which images are divided before patching and inference are performed. Float (0,1]. Understanding this option requires some background on the inference method used by this program. Images are not passed through the model whole. Instead, they are chopped up into many smaller, overlapping patches. Those patches are fed into the model, then reassembled into something the same dimension as the input image. Due to the structure of our model, the size of the patches into which the images are divided must be constant for a given model. Since the dimensions of MRI images can vary significantly by modality, it is possible that a patch size that leads to a reasonable number of patches in a high resolution anatomical MRI image would be larger than the entire input image for the corresponding fMRI image. Since this would not work, it is essential to increase the size of small images such that the constant patch size works well will all relevant modalities. Setting the value of 'new_spacing' is one such way to accomplish that task. The input dimensions of a given MRI image are divided by one of the three values passed to this option. Consider an fMRI image with dimensions [64, 64, 17]. Say the original spacing was [1, 1, 1]. If we were to pass new spacing options of [0.25, 0.25, 1], the image sent to patching would be of dimensions [256, 256, 17]. This expansion is accomplished by linear interpolation. The new image size would now allow for reasonable use of a 128 by 128 image patch, which is larger than the original image size.
This option corresponds to altering the dimensions of a given slice.
NOTE: This option is not recommended. It is the primary mechanism by which the original CAMRI at UNC model adjusted image dimensions, but more clear and consistent options have

since been developed. We leave this option here for completeness, but it is not recommended. Consider using 'target_size' instead!

-vspace, --interlayer_interpolation_spacing: A multiplicative factor by which images are divided before patching and inference are performed. Float (0,1]. See above -hspace for information. Not recommended.

Image Preprocessing Options – Mode 2

-ts, --target_size: Size to which all input images should be adjusted before they are sent to inference. Int, [1,inf). An alternative method of increasing the size of images to ensure successful patching. Determines the single value to which one dimension of input images will be set. Other slice dimension set to preserve aspect ratio of input image. It is generally recommended to set this value to roughly twice the image patch size defined by the model of choice. It is also recommended that the value not be below the largest dimension of any input image.
EX: -cs 256

-ufp, --use_frac_patch: Whether to define patch size as a fraction of input image size. String, True or False. If true, enables use of fractional patch sizes. Fractional patch sizing is an alternative to the traditional user-defined patch size. If the model was trained using fractional patch size, then this value should be set to true.
EX: -ufp True

-fp, --frac_patch: Dimension of images patches on which inference is run, as a fraction of resampled input image dimensions. Float, (0,1). This value is defined by the model used for inference. This value is only checked if use_frac_patch is true. If the value is incorrect, inference will fail.

-fs, --frac_strice: Distance neighboring image patches translate within a slice as a fraction of resampled input image dimensions. Float, (0,1). This value defines the amount of overlap neighboring patches will have. On average, we have seen that increasing overlap leads to increasing performance. Correspondingly, increasing overlap will also increase inference time. Common fractional stride values are 0.75, 0.5, and 0.25.