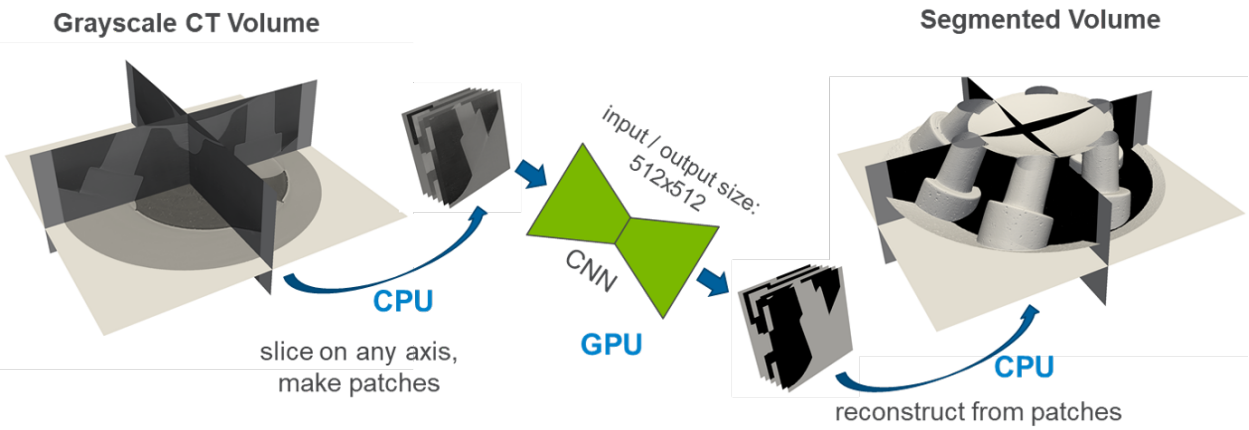
CTSegNet Documentation

Release 1.225

Argonne National Laboratory

May 16, 2021

CONTENTS



CTSegNet is a package for end-to-end 3D segmentation workflow for large X-ray tomographic datasets using 2D fully convolutional neural networks (fCNN).

CHAPTER

ONE

FEATURES

- Command-line interface for deploying 3D segmentation workflow using 2D fCNNs as well training your own Unet-like models with data-augmentation.
- Read/write utilities support tiff series and hdf5 format. The hdf5 implementation exploits chunking to minimize RAM usage in very large (>50 GB) datasets.
- Write your own scripts to test models and visualize segmentation results using the API.

CHAPTER

TWO

CONTRIBUTE

- Documentation: https://github.com/aniketkt/CTSegNet/tree/master/doc
- Issue Tracker: https://github.com/aniketkt/CTSegNet/docs/issues
- Source Code: https://github.com/aniketkt/CTSegNet/

CHAPTER

THREE

CONTENT

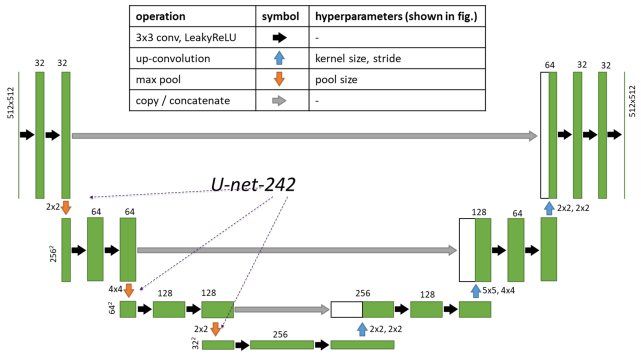
3.1 About

3.1.1 The Algorithm

fCNN architecture

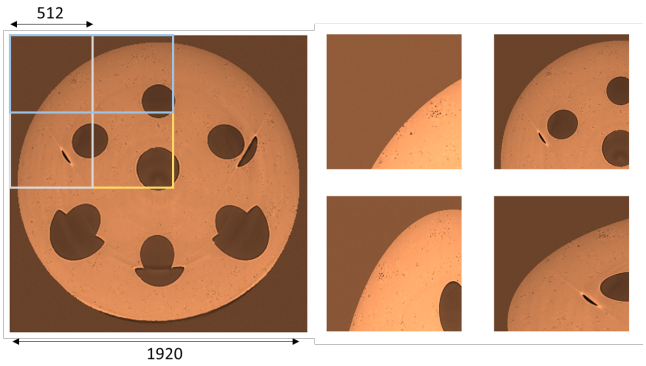
CTSegNet deploys unique Unet-like models trained with focal loss to provide accuracy with reduced number of convolutional layers. The methodology and performance metrics are discussed in [].

Here is an example architecture that you can build using the model_utils sub-module in CTSegNet. We will refer to it as Unet-242 because of the 2-4-2 implementation of pooling layers.

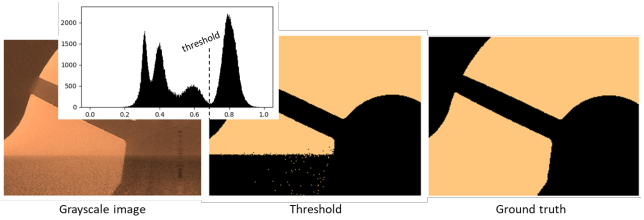


What is unique about CTSegNet?

While Unet-based segmentation is now commonplace, it is primarily limited to 2D data since 3D convolutional layers require prohibitively large GPU memory during training. Our approach efficiently exploits 2D fCNNs for 3D segmentation. You can generate multiple 3D masks by slicing along any axis, and choose a patching strategy based on the resolution-to-context trade-off in your CT data. For an fCNN with input/output images sized 512x512, you can make patches in several ways. This a slice drawn from a scan of a gasoline injector along the transverse plane.



An ensemble vote from several 3D segmentations maps yields near voxel accuracy in many cases, where thresholding just won't work. Here's an example of a band-like artifact from restricted field-of-view in a CT scan (sagittal plane is shown).



The data_utils.data_io module contains the DataFile class, which enables fast and memory-efficient slicing using hdf5 format so you can visualize and segment 100GB+ datasets from your workstation. With this, you can segment only parts of your data or test models on slices of your data, with a few lines of code. Tiff format is also supported but with limited functionality.

Tell me more

Read our paper at [] or contact me at: atekawade [at] anl [dot] gov

3.2 Install

3.2.1 ct_segnet with CLI

To download the entire package with executables, sample model files and config files, clone the master branch and install locally. To download the .h5 model files when cloning, you will need Git LFS installed.

```
git clone https://github.com/aniketkt/CTSegNet.git
pip install CTSegNet/.
```

3.2.2 ct_segnet only

To install only ct_segnet modules into your python 3 environment, use pip. For compatibility with tensorflow 1.14, please install ct_segnet 1.16 from the tf-1 branch.

```
$ pip install git+https://github.com/aniketkt/CTSegNet.git@master#egg=ct_segnet
```

in a prepared virtualenv or as root for system-wide installation.

3.3 Usage

3.3.1 Command-line interface

In addition to the API, CTSegNet provides a command-line interface to perform segmentation and training of new models.Data formats supported are .tiff sequence and hdf5. Example config files are provided in cfg_files/.

TRAIN/TEST: Extract training data from arbitrarily sized CT data and ground-truth pairs:

```
python bin/make_training_dataset.py -c cfg_files/setup_train.cfg
```

Build and train several Unet-like fCNN architectures for an input image size of your choice:

```
python bin/train_fCNN.py -t cfg_files/train.cfg -m cfg_files/models/Unet242.cfg
```

SEGMENT: An end-to-end 3D segmentation workflow that binarizes 2D images extracted from 3D CT data using the fCNN model, then rebuilds the corresponding 3D segmentation map. The hdf5 version is optimized for low RAM usage in very large (>50 GB) datasets.:

```
python bin/run_segmenter.py -c cfg_files/setup_seg.cfg
```

USE HDF5 FORMAT: Re-package your CT data into hdf5 format, with methods to determine optimal chunk size. Although optional, using hdf5 format accelerates read/write time while slicing through your datasets. Set -c as chunk size in MB or chunk shape z,y,x.:

```
python bin/rw_utils/convert_to_hdf5.py -f my_tiff_folder -o output_file.hdf5 -c 20.0
```

3.4 API reference

CTSegNet Modules:

3.4.1 seg_utils

CTSegNet is more than a 2D CNN model - it's a 3D Segmenter that uses 2D CNNs. The set_utils.py defines the Segmenter class that wraps over a keras U-net-like model (defined by models.py), integrating 3D slicing and 2D patching functions to enable the 3D-2D-3D conversations in the segmentation workflow. To slice a 3D volume, manipulations such as 45 deg rotations, orthogonal slicing, patch extraction and stitching are performed.

ct_segnet.seg_utils.**process_data**(*p*, *segmenter*, *preprocess_func=None*, *max_patches=None*, *overlap=None*, *nprocs=None*, *rot_angle=0.0*, *slice_axis=0*, *crops=None*, *arr_split=1*, *arr_split_infer=1*)

Segment a volume of shape (nz, ny, nx). The 2D keras model passes along either axis (0,1,2), segmenting images with a patch size defined by input to the model in the segmenter class.

Parameters

- **max_patches** (*tuple*) – (my, mx) are # of patches along Y, X in image (ny, nx)
- **overlap** (*tuple or int*) – number of overlapping pixels between patches
- **nprocs** (*int*) – number of CPU processors for multiprocessing Pool
- **arr_split** (*int*) – breakdown chunk into arr_split number of smaller chunks
- **slice_axis** (*int*) – (0,1,2); axis along which to draw slices

crops [list] list of three tuples; each tuple (start, stop) will define a python slice for the respective axis

rot_angle [float] (degrees) rotate volume around Z axis before slicing along any given axis. Note this is redundant if slice_axis = 0

nprocs [int] number of CPU processors for multiprocessing Pool

arr_split [int] breakdown chunk into arr_split number of smaller chunks

preprocess_fun [func] pass a preprocessing function that applies a 2D filter on an image

class ct_segnet.seg_utils.**Segmenter**(*model_filename=None*, *model=None*, *model_name='unknown'*, *weight_file_name=None*, *GPU_mem_limit=16.0*)

The Segmenter class wraps over a keras model, integrating 3D slicing and 2D patching functions to enable the 3D-2D-3D conversations in the segmentation workflow.

model: tf.keras.model keras model with input shape = out shape = (ny, nx, 1)

model_filename [str] path to keras model file (e.g. "model_1.h5")

model_name [str] (optional) just a name for the model

GPU_mem_limit [float] max limit of GPU memory to use

seg_chunk(*p*, *max_patches=None*, *overlap=None*, *nprocs=None*, *arr_split=1*, *arr_split_infer=1*)

Segment a volume of shape (nslices, ny, nx). The 2D keras model passes along nslices, segmenting images (ny, nx) with a patch size defined by input to the model

max_patches: tuple (my, mx) are # of patches along Y, X in image (ny, nx)

overlap [tuple or int] number of overlapping pixels between patches

nprocs [int] number of CPU processors for multiprocessing Pool

arr_split [int] breakdown chunk into arr_split number of smaller chunks

seg_image(*s*, *max_patches=None*, *overlap=None*)

Test the segmenter on arbitrary sized 2D image. This method extracts patches of shape same as the input shape of 2D CNN, segments them and stitches them back to form the original image.

max_patches [tuple] (my, mx) are # of patches along Y, X in image

s [numpy.array] greyscale image slice of shape (ny, nx)

overlap [tuple or int] number of overlapping pixels between patches

class ct_segnet.seg_utils.**FeatureExtraction2D**(*max_patches=None*, *overlap=None*, *model_filename=None*)

This class converts a 2D image into an n-dimensional vector z

Parameters **max_patches** (*tuple*) – (my, mx) are # of patches along Y, X in image

overlap [tuple or int] number of overlapping pixels between patches

model: tf.keras.model keras model with input shape = out shape = (ny, nx, 1)

model_filename [str] path to keras model file (e.g. "model_1.h5")

model_name [str] (optional) just a name for the model

`extract_code()`
not implemented

to do: consider patches are created. How should the code vectors of each patch be converted to singe vector? (mean, median, std?)

`extract_measurement`(*img*, *measurement*, ***kwargs*)

Returns **measured_features** (*np.array*) – shape (ndims,)

Parameters

- **img** (*np.array*) – A 2D numpy array (ny,nx). Could be a tomo slice or projection.
- **measurement** (*func*) – function to extract a measurement, e.g. radius, particle centroid, etc.

`vis_feature`(*s*, *measurement*, ***kwargs*)
This method extracts patches of shape same as the input shape of 2D CNN, measures a feature for each patch's segmentation map and stitches them back to form a checkered image.

s [numpy.array] greyscale image slice of shape (ny, nx)

3.4.2 data_io

A memory-efficient interface to slice, read and write CT data. Tiff series and hdf5 data formats are currently supported.

`class ct_segnet.data_utils.data_io.DataFile`(*fname*, *data_tag=None*, *tiff=False*, *chunk_shape=None*, *chunk_size=None*, *chunked_slice_size=None*, *d_shape=None*, *d_type=None*, *VERBOSITY=1*)
An instance of a DataFile class points to a 3D dataset in a tiff sequence or hdf5 file. The interface includes read/write methods to retrieve the data in several ways (slices, chunks, down-sampled data, etc.)

For setting chunk size in hdf5, either chunk_shape > chunk_size > chunked_slice_size can be input. If two or more are provided, this order is used to select one.

Parameters

- **fname** (*str*) – path to hdf5 filename or folder containing tiff sequence
- **tiff** (*bool*) – True if fname is path to tiff sequence, else False
- **data_tag** (*str*) – dataset name / path in hdf5 file. None if tiff sequence
- **VERBOSITY** (*int*) – 0 - print nothing, 1 - important stuff, 2 - print everything
- **d_shape** (*tuple*) – shape of dataset; required for non-existent dataset only
- **d_type** (*numpy.dtype*) – data type for voxel data; required for non-existent dataset only
- **chunk_size** (*float*) – in GB - size of a hyperslab of shape proportional to data shape
- **chunked_slice_size** (*float*) – in GB - size of a chunk of some slices along an axis
- **chunk_shape** (*tuple*) – shape of hyperslab for hdf5 chunking
- **Example**

- **.. highlight:: python**
- **.. code-block:: python** – from ct_segnet.data_io import DataFile # If fname points to existing hdf5 file dfile = DataFile(fname, tiff = False, data_tag = "dataset_name")

  # read a slice img = dfile.read_slice(axis = 1, slice_idx = 100)

  # read a chunk of size 2.0 GB starting at slice_start = 0 vol, s = dfile.read_chunk(axis = 1, slice_start = 0, max_GB = 2.0)

  # read a chunk between indices [10, 100], [20, 200], [30, 300] along the respective axes vol = dfile.read_data(slice_3D = [slice(10, 100), slice(20, 200), slice(30,300)])

  # or just read all the data vol = dfile.read_full()

`create_new`(*overwrite=False*)
For hdf5 - creates an empty dataset in hdf5 and assigns shape, chunk_shape, etc. For tiff folder - checks if there is existing data in folder.

Parameters `overwrite` (*bool*) – if True, remove existing data in the path (fname).

`est_chunking`()

`get_slice_sizes`()

`get_stats`(*return_output=False*)
Print some stats about the DataFile (shape, slice size, chunking, etc.)

`read_chunk`(*axis=None*, *slice_start=None*, *chunk_shape=None*, *max_GB=10.0*, *slice_end=''*, *skip_fac=None*)
Read a chunk of data along a given axis.

Parameters

- **axis** (*int*) – axis > 0 is not supported for tiff series
- **slice_start** (*int*) – start index along axis
- **chunk_shape** (*tuple*) – (optional) used if hdf5 has no attribute chunk_shape
- **max_GB** (*float*) – maximum size of chunk that's read. slice_end will be calculated from this.
- **slice_end** (*int*) – (optional) used if max_GB is not provided.
- **skip_fac** (*int*) – (optional) "step" value as in slice(start, stop, step)

Returns *tuple* – (data, slice) where data is a 3D numpy array

`read_data`(*slice_3D=(slice(None, None, None), slice(None, None, None), slice(None, None, None))*)
Read a block of data. Only supported for hdf5 datasets.

Parameters `slice_3D` (*list*) – list of three python slices e.g. [slice(start,stop,step)]*3

`read_full`(*skip_fac=None*)
Read the full dataset

`read_sequence`(*idxs*)
Read a list of indices idxs along axis 0.

Parameters

- `axis` (*int*) – axis 0, 1 or 2
- `idxs` (*list*) – list of indices

read_slice(*axis=None*, *slice_idx=None*)
Read a slice.

Parameters

- **axis** (*int*) – axis 0, 1 or 2
- **slice_idx** (*int*) – index of slice along given axis

set_verbosity(*VERBOSITY*)

show_stats(*return_output=False*)
print dataset shape and slice-wise size

write_chunk(*ch*, *axis=None*, *s=None*)
Write a chunk of data along a given axis.

Parameters

- **axis** (*int*) – axis > 0 is not supported for tiff series
- **s** (*slice*) – python slice(start, stop, step) - step must be None for tiff series

write_data(*ch*, *slice_3D=None*)
Write a block of data. Only supported for hdf5 datasets.

Parameters

- **ch** – 3D numpy array to be saved
- **slice_3D** (*list*) – list of three python slices e.g. [slice(start,stop,step)]*3 - must match shape of ch

write_full(*ch*)
Write the full dataset to filepath.

Parameters **ch** – 3D numpy array to be saved

ct_segnet.data_utils.data_io.**Parallelize**(*ListIn*, *f*, *procs=- 1*, ***kwargs*)
This function packages the "starmap" function in multiprocessing, to allow multiple iterable inputs for the parallelized function.

Parameters

- **ListIn** (*list*) – list, each item in the list is a tuple of non-keyworded arguments for f.
- **f** (*func*) – function to be parallelized. Signature must not contain any other non-keyworded arguments other than those passed as iterables.

Example:

```
def multiply(x, y, factor = 1.0):
    return factor*x*y

X = np.linspace(0,1,1000)
Y = np.linspace(1,2,1000)
XY = [ (x, Y[i]) for i, x in enumerate(X)] # List of tuples
Z = Parallelize_MultiIn(XY, multiply, factor = 3.0, procs = 8)
```

Create as many positional arguments as required, but remember all must be packed into a list of tuples.

3.4.3 train_utils

3.4.4 viewer

3.4.5 stats

A module for estimating

1. signal-to-noise ratio (SNR) for binarizable datasets.
2. accuracy metrics for segmentation maps.

ct_segnet.stats.**ROC**(*thresh*, *true_img=None*, *seg_img=None*)
Receiver Operating Characteristics (ROC) curve

Parameters

- **thresh** (*float*) – threshold value
- **true_img** (*numpy.array*) – ground truth segmentation map (ny, nx)
- **seg_img** (*numpy.array*) – predicted segmentation map (ny, nx)

Returns *tuple* – FPR, TPR

ct_segnet.stats.**calc_SNR**(*img*, *seg_img*, *labels=(0, 1)*, *mask_ratio=None*)
SNR = 1 / s*sqrt(std0^^2 + std1^^2) where s = 1 / (mu1 - mu0) mu1, std1 and mu0, std0 are the mean / std values for each of the segmented regions respectively (pix value = 1) and (pix value = 0). seg_img is used as mask to determine stats in each region.

Parameters

- **img** (*np.array*) – raw input image (2D or 3D)
- **seg_img** (*np.array*) – segmentation map (2D or 3D)
- **labels** (*tuple*) – an ordered list of two label values in the image. The high value is interpreted as the signal and low value is the background.
- **mask_ratio** (*float or None*) – If not None, a float in (0,1). The data are cropped such that the voxels / pixels outside the circular mask are ignored.

Returns *float* – SNR of img w.r.t seg_img

ct_segnet.stats.**calc_dice_coeff**(*true_img*, *seg_img*)

Parameters

- **true_img** (*np.array*) – ground truth segmentation map (ny, nx)
- **seg_img** (*np.array*) – predicted segmentation map (ny, nx)

Returns *float* – Dice coefficient

ct_segnet.stats.**calc_jac_acc**(*true_img*, *seg_img*)

Parameters

- **true_img** (*np.array*) – ground truth segmentation map (ny, nx)
- **seg_img** (*np.array*) – predicted segmentation map (ny, nx)

Returns *float* – Jaccard accuracy or Intersection over Union

ct_segnet.stats.**fidelity**(*true_imgs*, *seg_imgs*, *tolerance=0.95*)

Fidelity is number of images with IoU > tolerance

Parameters

- **tolerance** (*float*) – tolerance (default = 0.95)
- **true_imgs** (*numpy.array*) – list of ground truth segmentation maps (nimgs, ny, nx)
- **seg_imgs** (*numpy.array*) – list of predicted segmentation maps (nimgs, ny, nx)

Returns *float* – Fidelity

3.4.6 models

Easily define U-net-like architectures using Keras layers

ct_segnet.model_utils.models.**build_Unet_flex**(*img_shape*, *n_depth=1*, *n_pools=4*, *activation='lrelu'*, *batch_norm=True*, *kern_size=(3, 3)*, *kern_size_upconv=(2, 2)*, *pool_size=(2, 2)*, *dropout_level=1.0*, *loss='binary_crossentropy'*, *stdinput=True*)

Define your own Unet-like architecture, based on the arguments provided. Checks that the architecture complies with the converging-diverging paths and ensures the output image size is the same as input image size.

Returns a keras model for a U-net-like architecture

Return type tf.Keras.model

Parameters

- **img_shape** (*tuple*) – input image shape (ny,nx,1)
- **n_depth** (*int or list*) – Option 1: a list of the number of filters in each convolutional layer upstream of each pooling layer. Length must equal number of max pooling layers.

  Option 2: an integer that multiplies the values in this list: [16, 32, ...]. E.g. n_depth = 2 creates [32, 64, ...]
- **n_pools** (*int*) – Number of max pooling layers
- **activation** (*str or tf.Keras.layers.Activation*) – name of custom activation or Keras activation layer
- **batch_norm** (*bool*) – True to insert BN layer after the convolutional layers
- **kern_size** (*list or tuple*) – kernel size, e.g. (3,3). Provide a list (length = n_pools) of tuples to apply a different kernel size to each block.
- **kern_size_upconv** (*list or tuple*) – kernel size for upconv, e.g. (2,2). Provide a list (length = n_pools) of tuples to apply a different kernel size to each block
- **pool_size** (*list or tuple*) – max pool size, e.g. (2,2). Provide a list (length = n_pools) of tuples to apply a different size to each block
- **dropout_level** (*float or list*) – Option 1: a list (length = n_pools) of dropout values to apply separately in each block

  Option 2: a float (0..1) that multiples the values in this list: [0.1, 0.1, 0.2, 0.2, 0.3]
- **loss** (*str*) – The loss function of your choice. The following are implemented: 'weighted_crossentropy', 'focal_loss', 'binary_crossentropy'
- **stdinput** (*bool*) – If True, the input image will be normalized into [0,1]

ct_segnet.model_utils.models.**conv_layer**(*tensor_in*, *n_filters*, *kern_size=None*, *activation=None*, *kern_init='he_normal'*, *padding='same'*, *dropout=0.1*, *batch_norm=False*)

Define a block of two convolutional layers

Returns tensor of rank 4 (batch_size, n_rows, n_cols, n_channels)

Parameters

- **tensor_in** (*tensor*) – input tensor
- **n_filters** (*int*) – number of filters in each convolutional layer
- **kern_size** (*tuple*) – kernel size, e.g. (3,3)
- **activation** (*str or tf.Keras.layers.Activation*) – name of custom activation or Keras activation layer
- **kern_init** (*str*) – kernel initialization method
- **padding** (*str*) – type of padding
- **dropout** (*float*) – dropout fraction
- **batch_norm** (*bool*) – True to insert a BN layer

ct_segnet.model_utils.models.**insert_activation**(*tensor_in*, *activation*)

Returns tensor of rank 4 (batch_size, n_rows, n_cols, n_channels)

Parameters

- **tensor_in** (*tensor*) – input tensor
- **activation** (*str or tf.Keras.layers.Activation*) – name of custom activation or Keras activation layer

ct_segnet.model_utils.models.**pool_layer**(*tensor_in*, *n_filters*, *pool_size*, *dropout=None*, *activation=None*, *batch_norm=False*, *kern_size=None*)

Define a block of 2 convolutional layer followed by a pooling layer

Returns tensor of rank 4 (batch_size, n_rows, n_cols, n_channels)

Parameters

- **tensor_in** (*tensor*) – input tensor
- **n_filters** (*int*) – number of filters in each convolutional layer
- **pool_size** (*tuple*) – max pooling (2,2)
- **dropout** (*float*) – fraction of dropout
- **activation** (*str or tf.Keras.layers.Activation*) – name of custom activation or Keras activation layer
- **batch_norm** (*bool*) – True to insert a BN layer
- **kern_size** (*tuple*) – kernel size for conv layer, e.g. (3,3)

ct_segnet.model_utils.models.**upconv_layer**(*tensor_in*, *concat_tensor*, *n_filters=None*, *activation=None*, *kern_size=None*, *strides=None*, *padding='same'*, *batch_norm=False*)

Define an upconvolutional layer and concatenate the output with a conv layer from the contracting path

Returns tensor of rank 4 (batch_size, n_rows, n_cols, n_channels)

Parameters

- **tensor_in** (*tensor*) – input tensor
- **concat_tensor** (*tensor*) – this will be concatenated to the output of the upconvolutional layer
- **n_filters** (*int*) – number of filters in each convolutional layer
- **kern_size** (*tuple*) – kernel size for upconv, e.g. (2,2)
- **activation** (*str or tf.Keras.layers.Activation*) – name of custom activation or Keras activation layer
- **kern_init** (*str*) – kernel initialization method
- **strides** (*tuple*) – strides e.g. (2,2)
- **padding** (*str*) – type of padding
- **batch_norm** (*bool*) – True to insert a BN layer

3.4.7 losses

This module defines some custom loss functions and metrics that are used to train and evaluate a U-net-like tf.Keras.model. This require the input to be a tensor.

Note: Some of these metrics are implemented in ct_segnet.stats to receive numpy.array as inputs.

ct_segnet.model_utils.losses.**IoU**(*y_true*, *y_pred*)

Returns intersection over union accuracy

Parameters

- **y_true** (*tensor*) – Ground truth tensor of shape (batch_size, n_rows, n_cols, n_channels)
- **y_pred** (*tensor*) – Predicted tensor of shape (batch_size, n_rows, n_cols, n_channels)

ct_segnet.model_utils.losses.**acc_ones**(*y_true*, *y_pred*)

Returns accuracy in predicting ones = TP/(TP + FN)

Parameters

- **y_true** (*tensor*) – Ground truth tensor of shape (batch_size, n_rows, n_cols, n_channels)
- **y_pred** (*tensor*) – Predicted tensor of shape (batch_size, n_rows, n_cols, n_channels)

ct_segnet.model_utils.losses.**acc_zeros**(*y_true*, *y_pred*)

Returns accuracy in predicting zero values = TN/(TN + FP)

Parameters

- **y_true** (*tensor*) – Ground truth tensor of shape (batch_size, n_rows, n_cols, n_channels)
- **y_pred** (*tensor*) – Predicted tensor of shape (batch_size, n_rows, n_cols, n_channels)

ct_segnet.model_utils.losses.**focal_loss**(*y_true*, *y_pred*)

Returns loss value

Focal loss is defined here: https://arxiv.org/abs/1708.02002 Using this provides improved fidelity in unbalanced datasets: Tekawade et al. https://doi.org/10.1117/12.2540442

Parameters

- **y_true** (*tensor*) – Ground truth tensor of shape (batch_size, n_rows, n_cols, n_channels)
- **y_pred** (*tensor*) – Predicted tensor of shape (batch_size, n_rows, n_cols, n_channels)

ct_segnet.model_utils.losses.**my_binary_crossentropy**(*y_true*, *y_pred*)

Returns loss value

This is my own implementation of binary cross-entropy. Nothing special.

Parameters

- **y_true** (*tensor*) – Ground truth tensor of shape (batch_size, n_rows, n_cols, n_channels)
- **y_pred** (*tensor*) – Predicted tensor of shape (batch_size, n_rows, n_cols, n_channels)

ct_segnet.model_utils.losses.**weighted_crossentropy**(*y_true*, *y_pred*)

Returns loss value

Weighted cross-entropy allows prioritizing accuracy in a certain class (either 1s or 0s).

Parameters

- **y_true** (*tensor*) – Ground truth tensor of shape (batch_size, n_rows, n_cols, n_channels)
- **y_pred** (*tensor*) – Predicted tensor of shape (batch_size, n_rows, n_cols, n_channels)

3.5 Credits

3.5.1 Citations

We kindly request that you cite the following article [] if you use CTSegNet for your project.

PYTHON MODULE INDEX

C

INDEX

S

U

V

W