test Documentation

Release 1.0

Ryan Cohn

Aug 20, 2020

CONTENTS

CHAPTER

ONE

TUTORIALS

1.1 Installation

AMPIS depends on several packages, including PyTorch and Detectron2. The PyTorch installation depends on the available CUDA environment. Detectron2 depends on both the PyTorch installation and available CUDA environment. Therefore, the process is as follows. First, clone the repository, create a virtual environment, and install the standard python packages used by AMPIS. Then, install PyTorch according to your system. Next, install the corresponding Detectron2 build. Finally, install AMPIS.

1.1.1 1) Clone git repository

```
git clone https://github.com/rccohn/AMPIS.git
cd AMPIS
```

1.1.2 2) Set up and activate virtual environment

To create and activate an environment called **ampis_env:**

```
python3 -m venv ampis_env
source ampis_env/bin/activate
```

1.1.3 3) Install requirements file

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

Note- you may see some error messages pop up, but this is ok- pip should resolve the conflicts.

1.1.4 4) Install COCO Python API

Note `pip install pycocotools` does not work in some cases, I found this method to be better:

```
pip install git+https://github.com/cocodataset/cocoapi.git#subdirectory=PythonAPI
```

1.1.5 5) Install PyTorch and TorchVision

The version of torch and torchvision will depend on your system and cuda installation. For more information, see https://pytorch.org/.

For example, for installing pytorch 1.5 and the compatible version of torchvision for CUDA 10.1:

```
pip install torch==1.5.0+cu101 torchvision==0.6.0+cu101 -f https://download.pytorch.
↪org/whl/torch_stable.html
```

1.1.6 6) Install Detectron2

The version of detectron2 will depend on the versions of pytorch and CUDA installed on your system. There are different methods of installation. For Linux, the most straightforward method is to install a pre-built wheel. For other operating systems, detectron2 can be built from source, see their documentation for more info.

For pytorch 1.5 + CUDA 10.1:

```
python -m pip install detectron2 -f https://dl.fbaipublicfiles.com/detectron2/wheels/
↪cu101/torch1.5/index.html
```

More detailed information and additional builds can be found on the detectron2 installation page (https://github.com/facebookresearch/detectron2/blob/master/INSTALL).

1.1.7 7) Install AMPIS

The recommended method for installing AMPIS is with pip's 'editable' mode. After navigating to `AMPIS/`:

```
pip install -e .
```

1.1.8 8) Verify installation

To verify everything is correctly installed, open a python terminal and enter the following:

```
import pycocotools
import torch
import detectron2
import ampis
```

If the modules import without errors, you are good to go!

1.2 Getting Started

AMPIS provides comprehensive example for data preparation, training, and evaluating instance segmentation models in `examples/powder` and `examples/spheroidite`. The powder example is the most comprehensive and shows how instance segmentation can be applied to measure powder samples on a particle-by-particle basis. The spheroidite example shows the process for data formatted in a different way and shows how this technique can be used for segmentation of microconstituents in steel.

These examples follow the same process, which is summarized here.

1.2.1 Data labeling

Labeling data is a slow and painful process. Fortunately in these examples we provide labels for you! There are two formats for labels that can be used with AMPIS. The powder example shows how to use data annotated with the vgg image annotator (http://www.robots.ox.ac.uk/~vgg/software/via/), which stores polygons for the segmentation masks. The spheroidite example shows how labels stored in separate images or arrays can be used instead.

How much data do I need?

Surprisingly, not much. Deep learning models typically require thousainds, or even millions, of images to fully train (the ImageNet dataset currently contains over 14 million labeled images.) However, by leveraging transfer learning, the use of pre-trained models allow us to achieve good results with very few (10 or fewer) labeled images. Both examples show good results achieved from small datasets.

1.2.2 Loading data

Loading data dictionaries

After labeling data it is ready to be loaded into AMPIS. This is accomplished with the `data_utils.get_ddicts()` function. Currently, the data loader only works for single-class instance segmentation in the formats specified above in Data labeling. The powder example shows how separate label files can be used for multi-class segmentation (ie powder particles and satellites.) Detailed info for the format required can be found in the detectron2 documentation under using custom datasets. (https://detectron2.readthedocs.io/tutorials/datasets.html)

Dataset registration

After loading the data, it must be registered. Registration associates a name with a method for retrieving the data in a format that can be used by a model. Both examples demonstrate how to register datasets. Again, detailed information can be found in the detectron2 docs with the above link.

After loading and/or registering data, you can visually verify that the data is correctly loaded using `visualize.display_ddicts()'.

1.2.3 Model configuration and training

Model selection

After the data is annotated, loaded, and registered, it's time to set up the model. Detectron2 provides several pre-trained models with different configurations in the model zoo (https://github.com/facebookresearch/detectron2/blob/master/MODEL_ZOO). Both examples in AMPIS use `mask_rcnn_R_50_FPN_3x`, which gives good performance for instance segmentation. By leveraging transfer learning, we can get away with achieving high-performance instance segmentation with very small sets of labeled training data.

Configurations

The models have many hyperparameters which can be selected to adjust their performance on different datasets. The model architecture and parameters are specified in the `config` settings. Both examples show the most common settings that should be defined to tune the performance of the models. A complete list of configuration options and descriptions of what each option does can be found in the detectron2 config references (https://detectron2.readthedocs.io/modules/config.html#config-references).

Training

After defining the configurations, the model is very straightforward to train. The detectron2 DefaultTrainer (available in the detectron2engine.defaults (https://detectron2.readthedocs.io/modules/engine.html#module-detectron2.engine.defaults) module) provides all the basic functionality needed for training. If you have enough training data to use a training, validation, and test set, we also provide the AmpisTrainer object, which can compute the validation loss during training. To do this, the validation dataset must be registered separately from the training dataset. Both examples use small training sets and therefore use the standard DefaultTrainer.

1.2.4 Model inference

After training the model detectron2 DefaultEvaluator (also located in detectron2engine.defaults (https://detectron2.readthedocs.io/modules/engine.html#module-detectron2.engine.defaults)) is used to generate predictions on both the training images as well as unseen or held-out images.

The outputs of instance segmentation models are very large. Fortunately, the majority of the analysis and visualization can be done on a compressed representation of the data. `data_utils.format_output()` compresses the results and formats them for storage/later use.

After formatting, `visualize.display_ddicts()` can be used to view the model predictions on an image.

1.2.5 Model evaluation

Evaluation consists of comparing the model predictions to the ground truth labels for a given image. The `structures.InstanceSet` class is suited for evaluation. InstanceSet objects can load either ground truth labels from ddicts or formatted model predictions that were saved to disk. InstanceSet objects can be visualized with `visualize.display_iset()`.

Detection and segmentation scores

Typical computer vision models for instance segmentation are evaluated by the COCO metrics such as AP50 or mAP. These metrics are good for large datasets. The detectron2 COCOEvaluator (https://detectron2.readthedocs.io/modules/evaluation.html) can be used to compute these scores, if desired. (However, AMPIS provides metrics which provide more detail for smaller datasets. The scores are based on precision (ratio of true positive to all positive predictions) and recall (ratio of true positives to true positive and false negative predictions.)

These metrics are reported for two sets of scores. **Detection** scores describe how many predicted masks matched with a ground truth mask on the basis of IOU score. **Segmentation** scores describe how well each pair of matched masks agree with each other. Details for the metrics are provided in the examples and in the paper.

To compute the scores for each pair of ground truth/predicted results for a given image, use `analyze.det_seg_scores`.

The detection and segmentation results can also be visualized by first calling `analyze.det_perf_iset()` or `analyze.seg_perf_iset()`. This generates an InstanceSet containing the true positives, false positives, and false negative instances for each metric. Then, the InstanceSet can be visualized with the same `visualize.display_iset()` function.

1.2.6 Sample characterization

Up until now, this process has been entirely computer vision. But AMPIS was designed for materials scientists! After training the model and generating predictions, it's time to use the model for scientific exploration!

InstanceSet objects contain the segmentation masks for each image. We can gather some information from these directly. Calling `iset.compute_rprops()` returns a table of mesaurements of the masks using `skimage.measure.regionprops_table()`. The full list of available quantities that can be measured is avaliable in the skimage documents (https://scikit-image.org/docs/dev/api/skimage.measure.html#skimage.measure.regionprops).

More in-depth analysis requires additional functionality. `ampis.applications` is designated for modules with tools for specific applications. Currently, there is one module- `ampis.applications.powder`. This provides tools for quickly generating particle size distributions from image data as well as the ability to measure the satellite content of powder samples. The implementation of both of these techniques is included in the powder example. Currently this is the only method of directly measuring the satellite contents in powder samples, demonstrating the utility of instance segmentation for applications in materials characterization and quality control!

1.2.7 Documentation

For more information, see the AMPIS documentation (https://ampis.readthedocs.io/).

CHAPTER

TWO

API DOCUMENTATION

2.1 ampis.analyze package

Provides tools mostly for analyzing and evaluating model predictions after training and inference have been run. Notably, gives tools for matching ground truth to predicted instances for the same image, and methods to quantify the performance of the predictions.

ampis.analyze.**align_instance_sets**(*a*, *b*)

Reorders lists of instance sets so they are consistent with each other.

For lists of instance set objects *a* and *b*, rearranges *b* to match the order of *a*. Matching is performed on the basis of filenames. Only instance sets from *a* and *b* that have corresponding filenames are kept. This is useful for tasks like comparing the performance of instance predictions to their ground truth labels for multiple images.

Parameters

- **a** (*List of instance set objects.*) – Lists that will be matched to each other.
- **b** (*List of instance set objects.*) – Lists that will be matched to each other.

Returns **a_ordered, b_ordered** – Lists of matched instance set objects with consistent order by filenames.

Return type List of instance set objects.

ampis.analyze.**det_perf_iset**(*gt*, *pred*, *match_results=None*, *colormap=None*, *tp_gt=False*)

Stores detection true positives, false positives, and false negatives in an instance set for visualization.

Computes matches between gt and pred masks (unless match_results is supplied). Returns an instance set object containing the masks, boxes, and colors for true positive (matched,) false positive (unmatched pred) and false negative (unmatched gt) instances. Colors correspond to each class of instance. The instance set can be visualized using visualize.display_iset() to evaluate the detection performance of a model.

Parameters

- **gt** (*masks object (RLEMasks, list(dic), etc)*) – ground truth (gt) and predicted (predicted) masks that will be matched
- **pred** (*masks object (RLEMasks, list(dic), etc)*) – ground truth (gt) and predicted (predicted) masks that will be matched
- **match_results** (*dict or None*) – if dict: output from rle_instance_matcher() containing match scores and indices if None: rle_instance_matcher() will be called to compute scores with default iou_thresh=0.5
- **colormap** (*dict or None*) – if dict: dictionary with keys in ['TP','FP','FN'] for true positive, false positiev, and false negative, respectively, and values of RGB or RGBA colors (ie 3 or 4 element array of floats between 0 and 1)

- **tp_gt** (*bool.*) – If True, true positives will be displayed from ground truth instances. If False, true positives will be displayed from predicted instances.

Returns

- **iset** (*InstanceSet*) – instance set whose instances contain the tp, fp, fn instances (the type is indicated by the color)
- **colormap (optional)** (*dict*) – only returned if the colormap input is None. Dictionary with keys in ['TP','FP','FN'] and values are the default colors corresponding to each type of instance.

ampis.analyze.**det_seg_scores**(*gt*, *pred*, *iou_thresh=0.5*, *size=None*)

Computes detection and segmentation scores for a give pair of sets of masks.

Masks are matched on the basis of IOU score using rle_instance_matcher(). Then the scores are computed from the results. For each set of tests, the precision and recall are reported. Precision is the ratio of true positives to true positives + false positives. Recall is the ratio of true positives to true positives + false negatives.

Detection scores are calculated for the whole set of masks and describe the number of instances that were correctly matched with IOU above the threshold. True positives are matched masks, false positives are unmatched predicted masks, and false negatives are unmatched ground truth masks.

Segmentation scores are calculated for each matched pair of masks and describe how well the pair of masks agree with each other. In each matched pair of masks, true positives are pixels included in both the ground truth masks, false positives are pixels only included in the predicted mask, and false negatives are pixels only included in the ground truth mask.

Parameters

- **gt** (*RLEMasks or polygonmasks*) – ground truth (gt) or predicted (pred) masks that will be matched to each other to compute the match scores.
- **pred** (*RLEMasks or polygonmasks*) – ground truth (gt) or predicted (pred) masks that will be matched to each other to compute the match scores.
- **iou_thresh** (*float*) – IOU threshold for matching (see rle_instance_matcher())
- **size** (*None or tuple(int, int)*) –

  size passed to masks_to_rle. Can be None for rle masks or bitmasks. For polygonmasks, must be tuple (r, c) for the image height and width in pixels, respectively.

Returns **output** – Match results. See *Notes* for detailed information on the format of the dictionary.

Return type dictionary

Notes

The format of the output dictionary is as follows:

{'det_precision': float between 0 and 1, detection precision for instances

'det_recall': float between 0 and 1, detection recall for instances

'seg_precision': n_match element array containing the segmentation precision scores

for each matched pair of instances.

'seg_recall': n_match element array containing the segmentation recall scores for each matched pair of instances.

'det_tp' [n_match x 2 array of indices of the ground truth and predicted instances that were matched,] respectively. For example, match_tp[i] gives [gt_idx, pred_idx] for the i'th detection match.

'det_fn' : n_fn element array of detection false negative masks.

'det_fp' : n_fp element array of detection false positive masks.

'seg_tp': n_match element array containing the total number of segmentation true positives for each matched pair of masks.

'seg_fn': n_match element array of containing the total number of segmentation false negatives for each matched pair of masks.

'seg_fp': n_match element array of containing the total number of segmentation false positives for each matched pair of masks.

'det_tp_iou': n_match element array of IOU scores for each detection true positive}

ampis.analyze.**mask_edge_distance**(*gt_mask*, *pred_mask*, *gt_box*, *pred_box*, *matches*, *device='auto'*)

Investigate the disagreement between the boundaries of predicted and ground truth masks.

For every matched pair of masks in pred and gt, determine false positive and false negative pixels. For every false positive pixel, compute the distance to the nearest ground truth pixel. For every false negative pixel, compute the distance to the nearest predicted pixel.

Parameters

- **gt_mask** (*list or RLEMasks*) – ground truth and predicted masks, RLE format
- **pred_mask** (*list or RLEMasks*) – ground truth and predicted masks, RLE format
- **gt_box** (*array*) – array of bbox coordinates
- **pred_box** (*array*) – array of bbox coordinates
- **matches** (*array*) – n_match x 2 element array where matches[i] gives the index of the ground truth and predicted masks in gt and pred corresponding to match i. This can be obtained from mask_match_stats (results['match_tp'])
- **device** (*str*) – Determines which device is used. 'cpu': cpu 'cuda': CUDA-compatible gpu, if available, otherwise cpu will be used.

Returns **FP_distances, FN_distances** – List of results for each match in matches. Each element is a tensor containing the euclidean distances (in pixels) from each false positive to its nearest ground truth pixel(FP_distances) or the distance from each false negative to the nearest predicted pixel(FN_distances).

Return type list(torch.tensor)

ampis.analyze.**merge_boxes**(*box1*, *box2*)

Finds the smallest bounding box that fully encloses box1 and box2.

Boxes are of the form [r1, r2, c1, c2] (indices, not coordinates). The region of image *im* in the box can be accessed by im[r1:r2,c1:c2].

Parameters

- **box1** (*ndarray*) – 4-element array of the form [r1, r2, c1, c2], the indices of the top left and bottom right corners of the box
- **box2** (*ndarray*) – 4-element array of the form [r1, r2, c1, c2], the indices of the top left and bottom right corners of the box

Returns **bbox_merge** – 4-element array containing the combined boxes.

Return type ndarray

Examples

TODO example

ampis.analyze.**rle_instance_matcher**(*gt*, *pred*, *iou_thresh=0.5*, *size=None*)

Performs instance matching (single class) to determine true positives, false positives, and false negative instance predictions.

Instances are matched on the basis of Intersection over Union (IOU,) or ratio of areas of overlap between 2 masks to the total area occupied by both masks. IOU (A,B) = sum(A & B)/sum(A | B). This function wraps _piecewise_rle_match().

Parameters

- **gt** (*ampis.structures.RLEMasks or detectron2.structures.polygonmasks*) – ground truth (gt) or predicted (pred) instances that will be matched.
- **pred** (*ampis.structures.RLEMasks or detectron2.structures.polygonmasks*) – ground truth (gt) or predicted (pred) instances that will be matched.
- **iou_thresh** (*float*) – Minimum IOU threshold for a pair of masks to be considered a match.
- **size** (*None or tuple(int, int)*) – Not needed if gt and pred are RLEMasks. If either mask is a polygonmasks object, size needs to be specified to convert masks to RLE. Size is specified as image (height, width) in pixels.

Returns

- **results** (*dictionary with the following structure:*)
- *{*
- **'tp'** (*n_match x 2 array of indices of matches. The first element corresponds to the index of the gt instance*)
- *for match i. The second element corresponds to the index of the pred index for match i.*
- **'fn'** (*n_fn element array where each element is a ground truth instance that was not matched (false negative)*)
- **'fp'** (*n_fp element array where each element is a predicted instance that was not matched (false positive)*)
- **'iou'** (*n_match element array of IOU scores for each match.*)
- *}*

ampis.analyze.**seg_perf_iset**(*gt_masks*, *pred_masks*, *match_results=None*, *mode='reduced'*)

Stores segmentation true positives, false positives, and false negatives in an instance set for visualization.

Computes matches between gt and pred masks (unless match_results is supplied). Returns an instance set object containing the masks, boxes, and colors for true positive (included in both gt and pred masks that are matched), false positive (included in pred but not gt) and false negative (included in gt but not pred), and other(the same pixel is included in multiple instances due to overlap). Colors correspond to each class of instance. The instance set can be visualized using visualize.display_iset() to evaluate the quality of matched instances.

Parameters

- **gt** (*masks object (RLEMasks, list(dic), etc)*) – ground truth (gt) and predicted (predicted) masks that will be matched
- **pred** (*masks object (RLEMasks, list(dic), etc)*) – ground truth (gt) and predicted (predicted) masks that will be matched
- **match_results** (*dict or None*) – if dict: output from rle_instance_matcher() containing match scores and indices if None: rle_instance_matcher() will be called to compute scores with default iou_thresh=0.5
- **mode** (*str*) – 'all' or 'reduced' if 'all', there will be 8 colors corresponding to all possible combinations of pixels (described in colormap keys) if 'reduced', only pixels are only classified as tp, fp, fn, or'other'

Returns

- **iset** (*InstanceSet*) – instance set whose instances contain the tp, fp, fn instances (the type is indicated by the color)
- **colormap** (*dict*) – Dictionary with keys describing the pixel class and values are the default colors corresponding to each type of instance.

2.2 ampis.applications package

Contains modules that provide tools for domain specific applications (for example, powder characterization.)

2.3 ampis.data_utils package

Provides tools for loading labeled data, model training, and formatting model outputs for storage, visualization, and analysis.

Also provides the AmpisTrainer, which can get loss statistics on a validation set during training.

class ampis.data_utils.**AmpisTrainer**(*cfg*, *val_dataset=None*)
Extents detectron2 DefaultTrainer with validation loss metrics during training.

If you do not have a validation set or do not need the validation metrics during training, just use the detectron2 DefaultTrainer class.

build_hooks()
Adds hooks to the trainer. This is needed to get the validation loss during training.

class ampis.data_utils.**LossEvalHook**(*eval_period*, *model*, *data_loader*)
Adds validation loss statistics to AmpisTrainer class during training.

after_step()
method needed for hook to be automatically executed after training iterations.

ampis.data_utils.**compress_pred**(*pred*)
Compresses predicted masks to RLE and converts other outputs to numpy arrays.

Results in significantly smaller data structures that are easier to store and load into memory.

Parameters **pred** (*detectron2 Instances*) –

Outputs to be compressed. Outputs are obtained by calling detectron2 predictor on an image.

Returns **pred_compressed** – pred with masks compressed to RLE format and other outputs converted to numpy.

Return type detectron2 Instances

ampis.data_utils.**extract_boxes**(*masks*, *mask_mode='detectron2'*, *box_mode='detectron2'*)

Extracts bounding boxes from boolean masks.

For a boolean numpy array, bounding boxes are extracted from the min and max x and y coordinates containing pixels in each mask. Masks and boxes can be formatted for use with either detectron2 (default) or the matterport visualizer.

Detectron2 formatting: masks: n_mask x r x c boolean array boxes: (x0,y0,x1,y1) floating point box coordinates

Matterport formatting: masks: r x c x n_mask boolean array boxes: [r1,r2,c1,c2] integer box indices

Parameters

- **masks** (*ndarray*) – boolean array of masks. Can be 2 dimensions for 1 mask or 3 dimensions for array of masks.
- **mask_mode** (*str*) – if 'detectron2,' masks are shape n_mask x r x c. if 'matterport,' masks are r x c x n_masks.
- **box_mode** (*str*) – if 'detectron2', boxes will be returned in [x1,y1,x2,y2] floating point format. (XYXY_ABS box mode) if 'matterport,' boxes will be returned in [y1,y2,x1,x2] integer format.

Returns **boxes** – n_mask x 4 array with bbox coordinates in the format and dtype and specified by box_mode.

Return type ndarray

ampis.data_utils.**format_outputs**(*filename*, *dataset*, *pred*)

Formats model outputs consistently to make analysis easier later.

Note that this function applies compress_pred() to pred, which modifies the instance predictions in-place to drastically reduce the space they take up. See compress_pred() documentation for more info.

Parameters

- **filename** (*path or str*) – path of image corresponding to outputs
- **dataset** (*str*) – 'train' 'test' 'validation' etc that describes the class of the data
- **pred** (*detectron2 Instances*) – model outputs from detectron2 predictor.

Returns

Return type results- dictionary of outputs

ampis.data_utils.**get_ddicts**(*label_fmt*, *im_root*, *ann_root=None*, *pattern='*'*, *dataset_class=None*)

Reads images and their corresponding ground truth annotations in a format AMPIS can use.

Data-dicts are read for single-class instance segmentation labels. Currently, this only works for single-class sets of labels (ie all category_id labels are loaded as 0.) Extending it to multi-class labels should be straightforward but depends on the format of how the class labels are stored.

Annotations can be stored in the following format, specified by *label_fmt*.

- **binary: Annotations are stored in separate images or numpy files. Annotation images must be the same size as** the original images. In these images, background pixels are labeled 0, and pixels included in instances are labeled 1. skimage.measure.label() will be called to detect multiple instances

included in the image, so it is assumed that 1) instances do not touch/overlap and 2) instances are not discontinuous.

- **label: Annotations are stored in separate images or numpy files, with the same size and naming requirements** described above in 'binary'. In these images, background pixels are 0, and other pixel values indicate the unique instance ID that the pixel belongs to (similar to label images from skimage.measure.label().)
- **via2: Annotations are stored in JSON files generated from the VIA Image Annotator (version 2.) The exact form** for the annotations can be found in the examples/documentation.

Parameters

- **label_fmt** (*str*) –

  Format of ground truth instance labels for each image. See above for detailed description. Options are: 'binary': data_dicts will be read from binary images or npy files 'label': data_dicts will be read from label images or npy files 'via2': data_dicts will be read from VIA image annotator (version 2) JSON file

- **im_root** (*str, Path object.*) – path to directory containing images. If label_fmt == 'via2', im_root is the path to the JSON file generated by the VGG image annotator, which contains the paths to the images. Otherwise, im_root is the path to a folder containing images to be loaded in the dataset.
- **ann_root** (*str, Path object, or None*) – Only needed if label_fmt != 'via2' Path to directory containing annotations for each image in im_root. The annotation should have the same filename, except it may have a different extension. For example, the annotation for im_root/img1.png should be ann_root/img1.{png, tif, npy, etc...)
- **pattern** (*str*) – Glob pattern which can be used to select a subset files in im_root. By default, all files will be selected.
- **dataset_class** (*str or None*) – optional, can be used to indicate what set the data belongs to (ie 'Training','Testing','Validation',etc)

Returns **ddicts** – list of dictionaries containing the annotations for each image. Detailed information about the format is specified in the detectron2 documentation.

Return type list(dict)

2.4 ampis.structures package

Provides convenient data structures and methods for working with different masks and collections of instances.

class ampis.structures.**InstanceSet**(*mask_format=None*, *bbox_mode=None*, *filepath=None*, *annotations=None*, *instances=None*, *img=None*, *dataset_class=None*, *pred_or_gt=None*, *HFW=None*, *HFW_units=None*, *randomstate=None*)

Bases: object

Simple way to organize a set of instances for a single image.

Ensures that formatting is consistent for convenient analysis.

mask_format

'bitmask' for RLE encoded bitmasks or 'polygon' for polygon masks.

Type str or None

bbox_mode
indicates format in which bbox coords are stored. Should usually be BoxMode.XYXY_ABS.

Type detectron2.structures.BoxMode or None

filepath
path or filename of image which masks correspond to.

Type str, Path, or None

annotations
ddicts containing manual annotations for images (see `data_utils.get_ddicts()` for format.)

Type list(dict) or None

instances
contains information about class label, bbox, and segmentation mask for each instance. Also assigns random colors for each instance for visualization.

Type detectron2.structures.Instances object or None

img
image corresponding to annotations or predictions

Type np.ndarray or None

dataset_class
Describes if the image is in the training, validation, or test set.

Type str or None

pred_or_gt
'pred' for model predictions, 'gt' for ground truth labels.

Type str or None

HFW
Horizontal field width of *img*.

Type float or None

HFW_units
units of length for HFW (ie 'um')

Type str or None

rprops
if defined, contains region properties (ie area, perimeter, convex_hull, etc) of each mask.

Type None or list of skimage.measure.RegionProperties

compute_rprops(*keys=None*, *return_df=False*)
Computes region properties of segmentation masks (ie perimeter, convex area, etc) in self.instances.

Applies skimage.measure.regionprops_table to self.instances.masks for analysis. Allows for convenient measurements of many properties of the segmentation masks. A list of available parameters is available in the skimage documentation (see link below.) Stores results to self.rprops.

Parameters

- **keys** (*list(str) or None*) – Properties to measure. Passed to skimage.measure.regionprops_table().
- **return_df** (*bool*) – if True, returns the region properties as a pandas dataframe

Returns **rprops (optional)** – Pandas dataframe containing region properties. Only returned if input argument return_df=True

Return type DataFrame object

Notes

scikit image regionprops_table (https://scikit-image.org/docs/dev/api/skimage.measure.html#skimage.measure.regionprops_

copy()
Returns a copy of the InstanceSet object.

Returns a deep copy (ie everything is copied in memory, does not create pointers to the original class instance.)

Parameters **None** –

Returns **self** – returns a copy of InstanceSet.

Return type InstanceSet object

filter_mask_size(*min_thresh=100*, *max_thresh=100000*, *to_rle=False*)
Remove instances with mask areas outside of the interval (min_thresh, max_thresh.)

Useful for removing small instances (ie 1 or even 0 pixels in segmentation mask) or abnormally large outliers (ie many instances combined in a giant blob.) Note that this does not modify the InstanceSet in place and returns an Instances object.

Parameters

- **min_thresh** (*int, float or None*) – only instances with mask areas greater than min thresh and smaller than max_thresh are kept. If either threshold is None, it is not applied (ie if both min_thresh and max_thresh are None then all masks are kept.)
- **max_thresh** (*int, float or None*) – only instances with mask areas greater than min thresh and smaller than max_thresh are kept. If either threshold is None, it is not applied (ie if both min_thresh and max_thresh are None then all masks are kept.)
- **to_rle** (*bool*) – if True, masks are converted to RLE before filtering. The inlier masks will be returned as RLE. Otherwise, mask format is preserved.

Returns **instances_filtered** – Instances object only containing instances with mask areas in the threshold range.

Return type detectron2.structures.Instances object

read_from_ddict(*ddict*, *inplace=True*)
test

read_from_model_out(*outs*, *inplace=True*)
Read model predictions formatted with data_utils.format_outputs() function.

The relevant entries are stored as attributes in the InstanceSet object.

Parameters **outs** (*dict*) –

dictionary of formatted model outputs with the following format:

-'file_name': str or Path filename of image corresponding to predictions

- 'dataset': str string describing dataset image is in (ie training, validation , test, etc)
- **'pred': detectron2.structures.Instances object**

model outputs formatted with format_outputs(). Should have the following fields:

- **pred_masks: list** list of dictionaries of RLE encodings for the predicted masks
- **pred_boxes: ndarray** nx4 array of bounding box coordinates
- **scores: ndarray** n-element array of confidence scores for each instance (output from softmax of class label)
- **pred_classes: ndarray** n-element array of integer class indices for each predicted index

inplace: bool if True, the InstanceSet object is modified in place. Otherwise, the object is returned.

class ampis.structures.**RLEMasks**(*rle*)

Bases: object

Class for adding RLE masks to Instances object. Supports advanced indexing (such as indexing with an array or another list). This allows for selecting a subset of masks, which is not possible with the standard list(dic) of RLE masks.

ampis.structures.**boxes_to_array**(*boxes*)

Helper function to convert any type of object representing bounding boxes to a n_box * 4 numpy array

Parameters **boxes** (*object*) – list, array, or detectron2.structures.Boxes object that contains the bounding boxes which will be converted

Returns **box_array** – n x 4 array of bounding box coordinates

Return type ndarray

ampis.structures.**mask_areas**(*masks*)

Computes area in pixels of each mask in masks.

Mask areas are computed from counting pixels (bitmasks) or the shoelace formula (polygon masks.)

Parameters **masks** (*bitmask, polygonmask, or ndarray*) – Masks for which the areas will be calculated.

Returns **areas** – n_mask element array where each element contains the Farea of the corresponding mask in masks

Return type ndarray

ampis.structures.**masks_to_bitmask_array**(*masks*, *size=None*)

Converts various mask types to an n_mask x r x c boolean array of masks.

Parameters **masks** (*object*) – list, ndarray, RLEMasks, or PolygonMasks masks to be converted

size: tuple(int, int) or None Only needed to be specified for polygon masks. Tuple containing the image height and width in pixels needed to convert polygon coordinates into full segmentation masks.

Returns

- **mask_array** (*ndarray*)
- *n_mask x r x c boolean array of pixel values*

ampis.structures.**masks_to_rle**(*masks*, *size=None*)

Converts various objects to RLE masks

Parameters

- **masks** (*object*) – list, RLEMasks, or PolygonMasks object, masks to convert to RLE.
- **size** (*tuple(int, int) or None*) – Only needed to be specified for polygon masks. Tuple containing the image height and width in pixels needed to convert polygon coordinates into full segmentation masks.

Returns **rle_masks** – list of dictionaries with RLE encodings for each mask

Return type list(dic)

2.5 ampis.visualize package

Contains functions for visualizing images with segmentation masks overlaid.

ampis.visualize.**display_ddicts**(*ddict*, *outpath=None*, *dataset=''*, *gt=True*, *img_path=None*, *suppress_labels=False*, *summary=True*)

Visualize gt annotations overlaid on the image.

Displays the image in img_path. Overlays the bounding boxes and segmentation masks of each instance in the image.

ddict: list(dict) or for ground truth- data dict containing masks. The format of ddict is described below in notes.

outpath: str or path-like object, or None If None, figure is displayed with plt.show() and not written to disk If string/path, this is the location where figure will be saved to

dataset: str name of dataset, included in filename and figure title. The dataset should be registered in both the DatasetCatalog and MetadataCatalog for proper plotting. (see detectron2 datasetcatalog for more info.)

gt: bool if True, visualizer.draw_dataset_dict() is used for GROUND TRUTH instances if False, visualizer.draw_instance_predictions is used for PREDICTED instances

img_path: str or path-like object if None, img_path is read from ddict (ground truth) otherwise, it is a string or path to the image file

suppress_labels: bool if True, class names will not be shown on visualizer

summary: bool If True, prints summary of the ddict to terminal

None

Ddict should have the following format:

-'file_name': str or Path object path to image corresponding to annotations

-'mask_format': str 'polygonmask' if segmentation masks are lists of XY coordinates, or 'bitmask' if segmentation masks are RLE encoded segmentation masks

-'height': int image height in pixels

-'width': int image width in pixels

-'annotations': list(dic) list of annotations. See the annotation format below.

-'num_instances': int equal to len(annotations)- number of instances present in the image

The dictionary format for the annotation dictionaries is as follows:

-'category_id': int numeric class label for the instance.

-'bbox_mode': detectron2.structures.BoxMode object describes the format of the bounding box coordinates. The default is BoxMode.XYXY_ABS.

-'bbox': list(int) 4-element list of bbox coordinates

-'segmentation': list

list containing:

- a list of polygon coordinates (mask format is polygonmasks)
- dictionaries of RLE mask encodings (mask format is bitmasks)

ampis.visualize.**display_iset**(*img*, *iset*, *metadata=None*, *show_class_idx=False*, *show_scores=False*, *ax=None*, *colors=None*, *apply_correction=False*, *get_img=False*)

Visualize instances in *iset* overlaid on image *img*.

Displays the image and overlays the instances (masks, boxes, labels, etc.) If no axis object is provided to *ax*, creates and displays the figure. Otherwise, visualization is plotted on *ax* in place.

img: ndarray r x c {x 3} array of pixel values. Can be grayscale (2 dimensions) or RGB (3 dimensions)

iset: InstanceSet object iset.instances, a detectron2 Instances object, is used to get the masks, boxes, class_ids, scores that will be displayed on the visualization.

metadata: dict or None If None, metadata (ie string class labels) will not be shown on image. Else, metadata contains the metadata passed to detectron2 visualizer. In most cases, this should be a dictionary with the following structure:

- **'thing_classes': list** list of strings corresponding to integer indices of class labels. For example, if the classes are 0 for 'particle' and 1 for 'satellite', then metadata['thing_classes'] = ['particle','satellite']

get_img: bool if True, image will be returned as numpy array and no plot will be shown.

show_class_idx: bool if True, displays the class label (metadata['thing_classes'][class_idx]) on each instance in the image default: False

show_scores: bool if True, displays the confidence scores (output from softmax) on each instance in the image. default: False

ax: matplotlib axis object or None If an axis is supplied, the visualization is displayed on the axis. If ax is None, a new figure is created, and plt.show() is called for the visualization.

colors: ndarray or None Colors for each instance to be displayed. if colors is an ndarray, should be a n_mask x 3 array of colors for each mask. if colors is None and iset.instances.colors is defined, these colors are used. if colors is None and iset.instances.colors is not defined, colors are randomly assigned.

apply_correction: bool The visualizer appears to fill in masks. Applying the mask correction forces hollow masks to appear correctly. This is mostly used when displaying the results from analyze.mask_perf_iset(). In other cases, it is not needed.

None

Ddict should have the following format:

-'file_name': str or Path object path to image corresponding to annotations

-'mask_format': str 'polygonmask' if segmentation masks are lists of XY coordinates, or 'bitmask' if segmentation masks are RLE encoded segmentation masks

-'height': int image height in pixels

-'width': int image width in pixels

-'annotations': list(dic) list of annotations. See the annotation format below.

-'num_instances': int equal to len(annotations)- number of instances present in the image

The dictionary format for the annotation dictionaries is as follows:

-'category_id': int numeric class label for the instance.

-'bbox_mode': detectron2.structures.BoxMode object describes the format of the bounding box coordinates. The default is BoxMode.XYXY_ABS.

-'bbox': list(int) 4-element list of bbox coordinates

-'segmentation': list

list containing:

- a list of polygon coordinates (mask format is polygonmasks)
- dictionaries of RLE mask encodings (mask format is bitmasks)

ampis.visualize.**random_colors**(*n*, *seed*, *bright=True*)

Generate random colors for mask visualization.

To get visually distinct colors, generate colors in HSV with uniformly distributed hue and then convert to RGB. Taken from Matterport Mask R-CNN visualize, but added seed to allow for reproducability.

Parameters

- **n** (*int*) – number of colors to generate
- **seed** (*None or int*) – seed used to control random number generater. If None, a randomly generated seed will be used
- **bright** (*bool*) – if True, V=1 used in HSV space for colors. Otherwise, V=0.7.

Returns colors – n x 3 array of RGB pixel values

Return type ndarray

Examples

TODO quick example

PYTHON MODULE INDEX

a

INDEX

R

S