Part-aware Panoptic Segmentation

Release 2.0rc5

Panagiotis Meletis and Xiaoxiao (Vincent) Wen

May 01, 2022

GET STARTED

CHAPTER

ONE

INTRODUCTION

This repository contains code and tools for reading, processing, evaluating on, and visualizing Panoptic Parts datasets. Moreover, it contains code for reproducing our CVPR 2021 paper results.

1.1 Datasets

Cityscapes-Panoptic-Parts and *PASCAL-Panoptic-Parts* are created by extending two established datasets for image scene understanding, namely Cityscapes and PASCAL datasets. Detailed description of the datasets and various statistics are presented in our technical report in arxiv. The datasets can be downloaded from:

- Cityscapes Panoptic Parts
- PASCAL Panoptic Parts (alternative link, code: i7ap)

1.2 API and code reference

We provide a public, stable API, and various code utilities that are documented here.

1.3 Reproducing CVPR 2021 paper

The part-aware panoptic segmentation results from the paper can be reproduced using this guide.

1.4 Evaluation metrics

We provide two metrics for evaluating performance on Panoptic Parts datasets.

- Part-aware Panoptic Quality (PartPQ): here.
- Intersection over Union (IoU): *TBA*

1.5 Citations

Please cite us if you find our work useful or you use it in your research:

```
@inproceedings{degeus2021panopticparts,
    title = {Part-aware Panoptic Segmentation},
    author = {Daan de Geus and Panagiotis Meletis and Chenyang Lu and Xiaoxiao Wen and␣
↪Gijs Dubbelman},
    booktitle = {IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)},
    year = {2021}
}
```

```
@article{meletis2020panopticparts,
    title = {Cityscapes-Panoptic-Parts and PASCAL-Panoptic-Parts datasets for Scene␣
↪Understanding},
    author = {Panagiotis Meletis and Xiaoxiao Wen and Chenyang Lu and Daan de Geus and␣
↪Gijs Dubbelman},
    type = {Technical report},
    institution = {Eindhoven University of Technology},
    date = {16/04/2020},
    url = {https://github.com/tue-mps/panoptic_parts},
    eprint={2004.07944},
    archivePrefix={arXiv},
    primaryClass={cs.CV}
}
```

CHAPTER

TWO

INSTALLATION

The code can be installed from the PyPI and requires at least Python 3.7. It is recommended to install it in a Python virtual environment.

```
pip install panoptic_parts
```

Some functionality requires extra packages to be installed, e.g. evaluation scripts (tqdm) or Pytorch/Tensorflow (torch/tensorflow). These can be installed separately or by downloading the `optional.txt` file from this repo and running the following command in the virtual environment:

```
pip install -r optional.txt
```

After installation you can use the package as:

```
import panoptic_parts as pp

print(pp.VERSION)
```

There are three scripts defined as entry points by the package:

```
pp_merge_to_panoptic <args>
pp_merge_to_pps <args>
pp_visualize_label_with_legend <args>
```

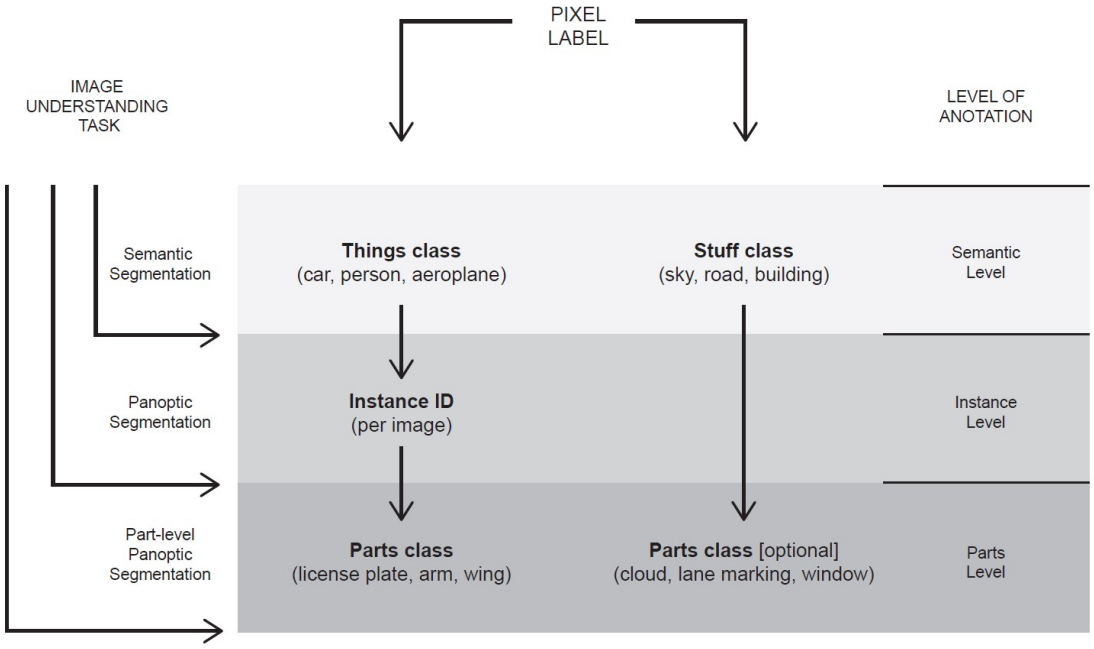
CHAPTER

THREE

SERIALIZATION FORMAT: HIERARCHICAL INFORMATION ENCODING

The goal of the format is to include (per image) all annotations in a single, image-like file with consistent representations across all abstractions and information levels. This enables easy transfer, reading, and compactly handling annotations. The following hierarchical structure is chosen, which extends the Cityscapes serialization format.

The goal of the format is to include (per image) all annotations in a single, image-like label file with a consistent encoding across all abstractions and information levels. This enables easy transfer, reading, and compact handling of the annotations. The following hierarchical structure is chosen, which extends the Cityscapes serialization format.



We encode three levels of labels: semantic, instance, and parts in a single image-like file. Labels for both datasets follow this format. Each pixel in our hierarchical label format has an up to 7-digit *universal id* (*uid*) containing:

- An up to 2-digit *semantic id* (*sid*), encoding the semantic-level *things* or *stuff* class.
- An up to 3-digit *instance id* (*iid*), a counter of instances per *things* class and per image. This is optional.
- An up to 2-digit *part id* (*pid*), encoding the parts-level semantic class per-instance and per-image. This is optional, but if provided requires also an *iid*. Only *things* parts are covered by this format.

We compactly encode the aforementioned *ids* (*sid*, *iid*, *pid*) into an up to 7-digit *uid*. Starting from the left, the first one or two digits encode the semantic class, the next 3 encode the instance (after zero pre-padding), and the final two encode the parts class (after zero pre-padding).

Using the above encoding:

- 1-2 digit *uids* encode only semantic-level labels
- 4-5 digit *uids* encode semantic-instance-level labels
- 6-7 digit *uids* encode semantic-instance-parts-level labels

For example, in *Cityscapes-Panoptic-Parts*, a *sky* (*stuff*) pixel will have *uid* = 23, a *car* (*things*) pixel that is labeled only on the semantic level will have *uid* = 26, if it's labeled also on instance level it can have *uid* = 26002, and a *person* (*things*) pixel that is labeled on all three levels can have *uid* = 2401002.

> The format covers parts-level classes for *stuff* semantic classes using a dummy instance id (`iid = 0`). Cityscapes Panoptic Parts and PASCAL Panoptic Parts do not currently define any *stuff* with part-level classes. This is a feature that can be used in future extensions.

3.1 Unlabeled/Ignored pixels

We handle the unlabeled / void / ignored / "do not care pixels" in the three levels as follows:

- Semantic level: For *Cityscapes-Panoptic-Parts* we use the original Cityscapes void class. For *PASCAL-Panoptic-Parts* we use the class with *uid* = 0.
- Instance level: For instances the void class is not required. If a pixel does not belong to an object or cannot be labeled on instance level then it has only an up to 2-digit *semantic id*.
- Parts level: For both datasets we use the convention that, for each semantic class, the part-level class with *pid* = 0 represents the void pixels, e.g., for a *person* pixel, *uid* = 2401000 represents the void parts pixels of instance 10. The need for a void class arises during the manual annotation process but in principle it is not needed at the parts level. Thus, we try to minimize void parts level pixels and assign them instead only the semantic- or semantic-instance -level labels.

CHAPTER

FOUR

API REFERENCE

We provide a public, stable API consisting of tested modules. However, in members of the API you may encounter experimental features (e.g. arguments or functions). These have the prefix *experimental_* and are exempted from stability guarantees.

The functions of the API are exported (apart from their original modules) also in the panoptic_parts namespace, so they can be imported and used as:

```
import panoptic_parts as pp
pp.decode_uids(uids)
```

4.1 Label format handling

panoptic_parts.utils.format.**decode_uids**(*uids*, *, *return_sids_iids=False*, *return_sids_pids=False*, *experimental_noinfo_id=- 1*, *experimental_dataset_spec=None*, *experimental_correct_range=False*)

Decode the compact panoptic-parts *uids* into consituent ids.

Given the universal ids *uids* encoded according to the panoptic-parts format described in https://panoptic-parts.readthedocs.io/en/stable/label_format.html, this function returns element-wise the semantic ids (sids), instance ids (iids), and part ids (pids). Optionally it returns the sids_iids and sids_pids as well. sids_iids represent the semantic-instance-level (two-level) labeling, e.g., sids_iids from Cityscapes-Panoptic-Parts ids from Cityscapes-Original. sids_pids represent the semantic-part-level (semantics) labeling.

Examples

- decode_uids(23, return_sid_pid=True) → (23, -1, -1, 23)
- decode_uids(23003, return_sid_pid=True) → (23, 3, -1, 23)
- decode_uids(2300304, return_sid_pid=True) → (23, 3, 4, 2304)
- decode_uids(tf.constant([1, 12, 1234, 12345, 123456, 1234567])) → ([1, 12, 1, 12, 1, 12],
  [-1, -1, 234, 345, 234, 345], [-1, -1, -1, -1, 56, 67])
- decode_uids(np.array([[1, 12], [1234, 12345]])) → ([[1, 12], [1, 12]],
  [[-1, -1], [234, 345]], [[-1, -1], [-1, -1]])

Each output has the same type and shape as *uids* (not shown for clarity).

Parameters

- **uids** – The panoptic-parts uids. Can be a tf.Tensor of dtype tf.int32 and arbitrary shape, or a np.ndarray of dtype np.int32 and arbitrary shape, or a torch.tensor of dtype torch.int32 and arbitrary shape, or a Python int, or a np.int32 integer, with elements encoded according to the panoptic-parts format.
- **return_sids_iids** (bool) – Optionally return sids_iids.
- **return_sids_pids** (bool) – Optionally return sids_pids.
- **experimental_noinfo_id** (int) – The integer representing the "no info"/void value.
- **experimental_dataset_spec** (typing.Optional[*panoptic_parts.specs.dataset_spec.DatasetSpec*]) – a DatasetSpec is used a) for removing the part-level instance information layer from the pids, this layer is not useful for Part-aware Panoptic Segmentation but is present in the encoded uids of some datasets (e.g. PPP), b) for ids range validity checking and correction according to that DatasetSpec (provide experimental_correct_range=True) for this functionality.
- **experimental_correct_range** (bool) – If a DatasetSpec is provided, the invalid ids according to that DatasetSpec, will be replaced with the experimental_noinfo_id value.

Returns

There are 4 return signatures according to the given return_* keyword arguments. All return values have the same type and shape as *uids*, where non-relevant/void pixels have value -1.

if return_sids_iids and return_sids_pids are False (default behavior): sids, iids, pids = decode_uids(uids)

if return_sids_iids is True: sids, iids, pids, sids_iids = decode_uids(uids, return_sids_iids=True)

if return_sids_pids is True: sids, iids, pids, sids_pids = decode_uids(uids, return_sids_pids=True)

if return_sids_iids and return_sids_pids are both True: sids, iids, pids, sids_iids, sids_pids = decode_uids(uids, return_sids_iids=True, return_sids_pids=True)

sids have no -1. iids have -1 for pixels labeled with semantic-level labels only. pids have -1 for pixels labeled with semantic-level or semantic-instance-level labels only. sids_iids: have no -1. sids_pids: have no -1.

panoptic_parts.utils.format.**encode_ids**(*sids*, *iids*, *pids*)

Given semantic ids (sids), instance ids (iids), and part ids (pids) this function encodes them element-wise to uids according to the hierarchical format described in README.

This function is the opposite of decode_uids, i.e., uids = encode_ids(decode_uids(uids)).

Parameters

- **sids** – all of the same type with -1 for non-relevant pixels with elements according to hierarchical format (see README). Can be: tf.Tensor of dtype tf.int32 and arbitrary shape, or np.ndarray of dtype np.int32 and arbitrary shape, or torch.tensor of dtype torch.int32 and arbitrary shape, or Python int, or np.int32 integer.
- **iids** – all of the same type with -1 for non-relevant pixels with elements according to hierarchical format (see README). Can be: tf.Tensor of dtype tf.int32 and arbitrary shape, or np.ndarray of dtype np.int32 and arbitrary shape, or torch.tensor of dtype torch.int32 and arbitrary shape, or Python int, or np.int32 integer.
- **pids** – all of the same type with -1 for non-relevant pixels with elements according to hierarchical format (see README). Can be: tf.Tensor of dtype tf.int32 and arbitrary shape,

or np.ndarray of dtype np.int32 and arbitrary shape, or torch.tensor of dtype torch.int32 and arbitrary shape, or Python int, or np.int32 integer.

Returns same type and shape as the args according to hierarchical format (see README).

Return type uids

4.2 Visualization

panoptic_parts.utils.visualization.**random_colors**(*num*)

Returns a list of *num* random Python int RGB color tuples in range [0, 255]. Colors can be repeated. This is desired behavior so we don't run out of colors.

Parameters **num** – Python int, the number of colors to produce

Returns a list of tuples representing RGB colors in range [0, 255]

Return type colors

panoptic_parts.utils.visualization.**uid2color**(*uids*, *sid2color=None*, *experimental_deltas=(60, 60, 60)*, *experimental_alpha=0.5*)

Generate an RGB palette for all unique uids in *uids*. The palette is a dictionary mapping each uid from *uids* to an RGB color tuple, with values in range [0, 255]. A uid is an up to 7-digit integer that is interpreted according to our panoptic parts format (see README), i.e., decode_uids(uid) = (sid, iid, pid).

The colors are generated in the following way:

- if uid represents a semantic-level label, i.e. uid = (sid, N/A, N/A), then `sid2color`[sid] is used.
- if uid represents a semantic-instance-level label, i.e. uid = (sid, iid, N/A), then a random shade of *sid2color`[sid] is generated, controlled by `experimental_deltas*. The shades are generated so they are as diverse as possible and the variability depends on the number of iids per sid. The more the instances per sid in the *uids*, the less discriminable the shades are.
- if uid represents a semantic-instance-parts-level label, i.e. uid = (sid, iid, pid), then a random shade is generated as in the semantic-instance-level case above and then it is mixed with a single color from the parula colormap, controlled by *experimental_alpha*. A different parula colormap is generated for each sid to achieve best discriminability of parts colors per sid.

If *sid2color* is not provided (is None) then random colors are used. If *sid2color* is provided but does not contain all the sids of *uids* an error is raised.

Example usage in {cityscapes, pascal}_panoptic_parts/visualize_from_paths.py.

Parameters

- **uids** – a list of Python int, or a np.int32 np.ndarray, with elements following the panoptic parts format (see README)
- **sid2color** – a dict mapping each sid of uids to an RGB color tuple of Python ints with values in range [0, 255], sids that are not present in uids will be ignored
- **experimental_deltas** – the range per color (Red, Green, Blue) in which to create shades, a small range provides shades that are close to the sid color but makes instance colors to have less contrast, a higher range provides better contrast but may create similar colors between different sid instances
- **experimental_alpha** – the mixing coeffient of the shade and the parula color, a higher value will make the semantic-instance-level shade more dominant over the parula color

Returns a dict mapping each uid to a color tuple of Python int in range [0, 255]

Return type uid2color

4.3 Misc

panoptic_parts.utils.utils.**safe_write**(*path*, *image*, ***params*)

Saves *image* to *path* by creating all intermediate directories. If the *path* already exists it does not override it and returns False. Extra params passed to the PIL.Image.save writer can provided by keyword arguments (e.g. optimize=True or compress_level=9).

Parameters

- **path** – a path passed to os.path.exists, os.makedirs and PIL.Image.save()
- **image** – a numpy image passed to PIL.Image.fromarray()

Returns False is path exists. True if the *image* is successfully written.

CHAPTER

FIVE

CODE REFERENCE

Documented/Undocumented functionality of the rest of the code his repo lies here. This functionality will be added to the API in the future. Until then, the following functions may be moved or be unstable.

5.1 Dataset & Evaluation specifications

class panoptic_parts.specs.dataset_spec.**DatasetSpec**(*spec_path*)

This class creates a dataset specification from a YAML specification file, so properties in the specification are easily accessed. Moreover, it provides defaults and specification checking.

Specification attribute fields:

- l: list of str, the names of the scene-level semantic classes
- l_things: list of str, the names of the scene-level things classes
- l_stuff: list of str, the names of the scene-level stuff classes
- l_parts: list of str, the names of the scene-level classes with parts
- l_noparts: list of str, the names of the scene-level classes without parts
- **scene_class2part_classes: dict, mapping for scene-level class name to part-level class names,** the ordering of elements in scene_class2part_classes.keys() and scene_class2part_classes.values() implicitly defines the sid and pid respectively, which can be retrieved with the functions below
- sid2scene_class: dict, mapping from sid to scene-level semantic class name
- sid2scene_color: dict, mapping from sid to scene-level semantic class color
- **sid_pid2scene_class_part_class: dict, mapping from sid_pid to a tuple of** (scene-level class name, part-level class name)

Specification attribute functions:

- scene_class_from_sid(sid)
- sid_from_scene_class(name)
- part_classes_from_sid(sid)
- part_classes_from_scene_class(name)
- scene_color_from_scene_class(name)
- scene_color_from_sid(sid)
- scene_class_part_class_from_sid_pid(sid_pid)

- sid_pid_from_scene_class_part_class(scene_name, part_name)

Examples (from Cityscapes Panoptic Parts):

- for the 'bus' scene-level class and the 'wheel' part-level class it holds: - 'bus' in l_things → True - 'bus' in l_parts → True - sid_from_scene_class('bus') → 28 - scene_color_from_scene_class('bus') → [0, 60, 100] - part_classes_from_scene_class('bus') → ['UNLABELED', 'window', 'wheel', 'light', 'license plate', 'chassis'] - sid_pid_from_scene_class_part_class('bus', 'wheel') → 2802

Experimental (format/API may change):

- l_allparts: list of str, a list of all parts in str with format f"{scene_class}-{part_class}", contains at position 0 the special 'UNLABELED' class

Notes

- **A special 'UNLABELED' semantic class is defined for the scene-level and part-level abstractions.** This class must have sid/pid = 0 and is added by befault to the attributes of this class if it does not exist in yaml specification.
- It holds that: - the special 'UNLABELED' class l, l_stuff, l_noparts - l = l_things l_stuff - l = l_parts l_noparts
- sids are continuous and zero-based
- iids do not need to be continuous
- pids are continuous and zero-based per sid

part_classes_from_scene_class(*name*)

part_classes_from_sid(*sid*)

scene_class_from_sid(*sid*)

scene_class_part_class_from_sid_pid(*sid_pid*)

scene_color_from_scene_class(*name*)

scene_color_from_sid(*sid*)

sid_from_scene_class(*name*)

sid_pid_from_scene_class_part_class(*scene_name*, *part_name*)

class panoptic_parts.specs.eval_spec.**PartPQEvalSpec**(*spec_path*)

This class creates an evaluation specification from a YAML specification file and provides convenient attributes from the specification and useful functions. Moreover, it provides defaults and specification checking.

class panoptic_parts.specs.eval_spec.**SegmentationPartsEvalSpec**(*spec_path*)

This class creates an evaluation specification from a YAML specification file and provides convenient attributes from the specification and useful functions. Moreover, it provides defaults and specification checking.

Accessible specification attributes:

- dataset_spec: the associated dataset specification
- Nclasses: the number of evaluated classes (including ignored and background)
- **scene_part_classes: list of str, the names of the scene-part classes for evaluation,** ordered by the eval id

- eid_ignore: the eval_id to be ignored in evaluation
- **sid_pid2eval_id: dict, maps all sid_pid (0-99_99) to an eval_id,** according to the template in specification yaml
- **sp2e_np: np.ndarray, shape: (10000,), sid_pid2eval_id as an array for dense gathering,** position i has the sid_pid2eval_id[i] value

Member functions:

-

5.2 Visualization

panoptic_parts.visualization.visualize_label_with_legend.**visualize_from_paths**(*datasetspec_path*, *label_path*)

Visualizes in a pyplot window a label from the provided path.

For visualization pixels are colored on:

- semantic-level: according to colors defined in dataspec.sid2scene_color
- semantic-instance-level: with random shades of colors defined in dataspec.sid2scene_color
- semantic-instance-parts-level: with a mixture of parula colormap and the shades above

See panoptic_parts.utils.visualization.uid2color for more information on color generation.

Parameters

- **datasetspec_path** – a YAML file path, including keys: *sid2scene_color*, *scene_class_part_class_from_sid_pid*
- **label_path** – a label path, will be passed to Pillow.Image.open

panoptic_parts.utils.visualization.**experimental_colorize_label**(*label*, *, *sid2color=None*, *return_sem=False*, *return_sem_inst=False*, *emphasize_instance_boundaries=True*, *return_uid2color=False*, *experimental_deltas=(60, 60, 60)*, *experimental_alpha=0.5*)

Colorizes a *label* with semantic-instance-parts-level colors based on sid2color. Optionally, semantic-level and semantic-instance-level colorings can be returned. The option emphasize_instance_boundaries will draw a 4-pixel white line around instance boundaries for the semantic-instance-level and semantic-instance-parts-level outputs. If a sid2color dict is provided colors from that will be used otherwise random colors will be generated. See panoptic_parts.utils.visualization.uid2color for how colors are generated.

Parameters

- **label** – 2-D, np.int32, np.ndarray with up to 7-digit uids, according to format in README
- **sid2color** – a dictionary mapping sids to RGB color tuples in [0, 255], all sids in *labels* must be in *sid2color*, otherwise provide None to use random colors
- **return_sem** – if True returns *sem_colored*
- **return_sem_inst** – if True returns *sem_inst_colored*

Returns

3-D, np.ndarray with RGB colors in [0, 255], colorized *label* with colors that distinguish scene-level semantics, part-level semantics, and instance-level ids

sem_colored: 3-D, np.ndarray with RGB colors in [0, 255], returned if return_sem=True, colorized *label* with colors that distinguish scene-level semantics

sem_inst_colored: 3-D, np.ndarray with RGB colors in [0, 255], returned if return_sem_inst=True, colorized *label* with colors that distinguish scene-level semantics and part-level semantics

Return type sem_inst_parts_colored

panoptic_parts.utils.visualization.**_generate_shades**(*center_color*, *deltas*, *num_of_shades*)

panoptic_parts.utils.visualization.**_num_instances_per_sid**(*uids*)

panoptic_parts.utils.visualization.**_num_parts_per_sid**(*uids*)

panoptic_parts.utils.visualization.**_sid2iids**(*uids*)

panoptic_parts.utils.visualization.**_sid2pids**(*uids*)

5.3 Evaluation

class panoptic_parts.utils.experimental_evaluation_IOU.**ConfusionMatrixEvaluator_v2**(*eval_spec*, *filepaths_pairs*, *pred_reader_fn*, *experimental_validate_args=False*)

Bases: object

Computes the confusion matrix for the provided ground truth and prediction pairs filepaths using a tf.data pipeline for fast execution.

A standard use of this class is: evaluator = ConfusionMatrixEvaluator_v2(eval_spec, list_of_filepaths_pairs, pred_reader_fn) confusion_matrix = evaluator.compute_cm() metrics = compute_metrics_with_any_external_function(confusion_matrix)

compute_cm()

print_metrics(**args*, ***kwargs*)

5.4 Misc

panoptic_parts.utils.utils.**compare_pixelwise**(*l1*, *l2*)

Compare numpy arrays l1, l2 with same shape and dtype in a pixel-wise manner and return the unique tuples of elements that do not match for the same spatial position.

Parameters

- **l1** (*np.ndarray*) – array 1
- **l2** (*np.ndarray*) – array 2

Examples (supposing the following lists are np.ndarrays):

- compare_pixelwise([1,2,3], [1,2,4]) → [[3], [4]]
- compare_pixelwise([1,2,4,3], [1,2,3,5]) → [[3, 4], [5, 3]]

Returns unique_diffs: 2D, with columns having the differences for the same position sorted in ascending order using the l1 elements

Return type np.ndarray

panoptic_parts.utils.utils.**_sparse_ids_mapping_to_dense_ids_mapping**(*ids_dict*, *void*, *length=None*, *dtype=np.int32*)

Create a dense np.array from an ids dictionary. The array can be used for indexing, e.g. numpy advanced indexing or tensorflow gather. This method is useful to transform a dictionary of uids to class mappings (e.g. {2600305: 3}), to a dense np.array that has in position 2600305 the value 3. This in turn can be used in gathering operations. The reason that the mapping is given in a dictionary is due to its sparseness, e.g. we may not want to hard-code an array with 2600305 elements in order to have the mapping for the 2600305th element.

ids.values() and void must have the same shape and dtype.

The length of the dense_mapping is infered from the maximum value of ids_dict.keys(). If you need a longer dense_mapping provide the length in *length*.

Parameters

- **ids_dict** – dictionary mapping ids to numbers (usually classes),
- **void** – int, list of int, tuple of int, the positions of the dense array that don't appear in ids_dict.keys() will be filled with the void value,
- **length** – the length of the dense mapping can be explicitly provided
- **dtype** – the dtype of the returned dense mapping

CHAPTER

SIX

EVALUATE ON PARTPQ METRIC

To evaluate on the PartPQ metric, you need to follow three steps:

1. Select or prepare the EvalSpec for your data
2. Prepare the part-aware panoptic segmentation predictions in the correct format
3. Run the evaluation script

6.1 1. Select EvalSpec

In the `EvalSpec`, we define how we wish to evaluate the dataset. Specifically, we define:

- The classes that are to be evaluated, both on scene-level and part-level
- The split between *things* and *stuff* categories, and *parts* and *no-parts* categories
- The category definition and numbering that we expect for the predictions.

The `EvalSpec`s have the following filename format:

```
{metric-name}_{dataset-name}_{num-scene-classes}_{num-part-classes}_{specific-setting}_
↪evalspec.yaml
```

For the datasets that we define and use in our paper, we provide the `EvalSpec` that we use:

- ppq_cpp_19_23_cvpr21_default_evalspec.yaml: Cityscapes Panoptic Parts default (parts not grouped)
- ppq_cpp_19_23_cvpr21_grouped_evalspec.yaml: Cityscapes Panoptic Parts default (similar parts grouped)
- ppq_ppp_59_57_cvpr21_default_evalspec.yaml: PASCAL Panoptic Parts default

6.2 2. Prepare the predictions

Before we can evaluate the results, you should make sure that the predictions are in the proper format. There are two things to be considered:

1. The correct category ids should be used
2. The data should be encoded and provided in the proper 3-channel PNG format.

6.2.1 2.1. Category ids

The category ids in the prediction – both for scene classes and part classes – should be provided as defined in the `EvalSpec`.

1. For scene-level classes:
    - In `eval_sid2scene_label`, we provide the scene category ids that are used during evaluation, and their corresponding names.
    - In the prediction, these category ids should be used.
2. For part-level classes:
    - `eval_sid_parts` is a list of scene categories for which we expect part labels.
    - In `eval_sid_pid2eval_pid_flat`, we provide all the `sid_pid` category combinations that are evaluated.
        - The first part of the `sid_pid` is the scene category id (`sid`), the second is the and part category id (`pid`)
        - To see the corresponding category names for these `sid_pid`, see the mapping to the unique `eval_pid_flat`, and the provided class labels in `eval_pid_flat2scene_part_label`.
    - The `pid` from the `sid_pid`is the part category id that we expect in the predictions.

Example for CPP default:

1. As follows from `eval_sid2scene_label`:
    - The scene id for `car` is `26`, and `road` is `7`.
2. As follows from `eval_sid_pid2eval_pid_flat` and `eval_pid_flat2scene_part_label`:
    - The combined `sid_pid` prediction label for `person-head` is `24_02`
    - ==> The part id is `2` (and the scene id is `24`).

6.2.2 2.2. 3-channel PNG format

In the evaluation script, we expect the predictions to be encoded as a 3-channel PNG (i.e., `HxWx3`), where the channels should encode:

1. Scene category id
2. Instance id (unique for each instance within a scene category)
3. Part category id

These should be encoded as unsigned integers (`uint8`), and the filename of the PNG should include the filename of the original input image for which the prediction is the result.

For regions where there is no prediction, or regions with `unknown` predictions, the category id should be set to `255`.

6.3 3. Run evaluation script

To run the evaluation script, you need to have a json file containing information on the images that you wish to evaluate on. Here, we describe how to generate this `images.json` using `evaluation/prepare_data.py`.

Run the evaluation script from the top-level `panoptic_parts` directory as:

```
python -m panoptic_parts.evaluation.eval_PartPQ \
    $EVAL_SPEC_PATH \
    $GT_PATH \
    $PRED_PATH \
    $IMAGES_JSON \
    --save_dir=$SAVE_DIR
```

where:

- `$EVAL_SPEC_PATH`: selected evaluation specification from Step 1
- `$GT_PATH`: directory with ground truth files
- `$PRED_PATH`: directory with prediction files
- `$IMAGES_JSON`: the images.json file with a list of images and corresponding image ids
- `$SAVE_DIR`: a directory to save the json file with results (optional)

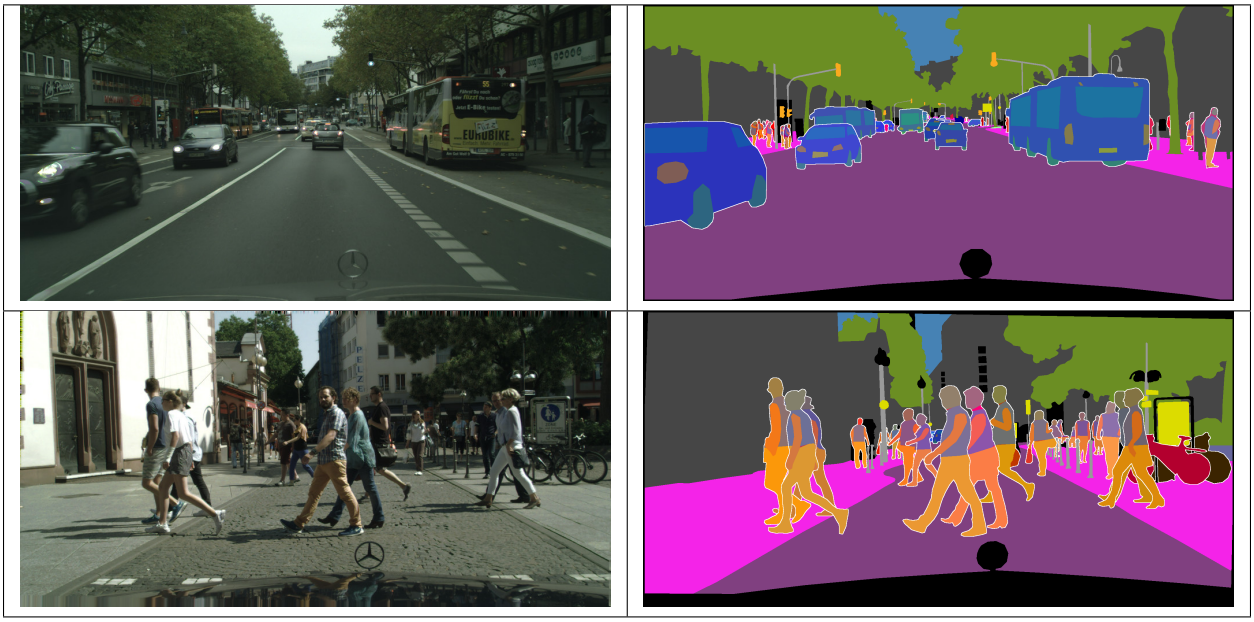
For more information on the arguments run `python -m panoptic_parts.evaluation.eval_PartPQ.py -h`.

CHAPTER

SEVEN

VISUALIZATION OF GROUND TRUTH

7.1 Cityscapes-Panoptic-Parts



7.2 PASCAL-Panoptic-Parts

CHAPTER

EIGHT

GENERATE PART-AWARE PANOPTIC SEGMENTATION RESULTS

Here, we provide a guide for generating Part-aware Panoptic Segmentation (PPS) results as in in our CVPR paper.

After publication we fixed a regression in the PartPQ metric (integrated in v2.x release of `panoptic_parts` package), which results in a slight increase of the PartPQ results. The correct results are provided in the CVPR 2021 Errata.

8.1 Prepare EvalSpec and dataset information

Before generating the Part-aware Panoptic Segmentation (PPS) results, you have to specify the dataset you wish to do this for. This consists of two parts:

1. Defining what category definition you wish to use, by using the EvalSpec.
2. Defining which images your dataset contains, and what their properties are.

8.1.1 EvalSpec

In the `EvalSpec`, we define the following properties

- The classes that are to be evaluated, both on scene-level and part-level
- The split between *things* and *stuff* categories, and *parts* and *no-parts* categories
- The category definition and numbering that we expect for the predictions.

For the datasets that we define and use in our paper, we provide the `EvalSpec` that we use:

- ppq_cpp_19_23_cvpr21_default_evalspec.yaml: Cityscapes Panoptic Parts default (parts not grouped)
- ppq_cpp_19_23_cvpr21_grouped_evalspec.yaml: Cityscapes Panoptic Parts default (similar parts grouped)
- ppq_ppp_59_57_cvpr21_default_evalspec.yaml: PASCAL Panoptic Parts default

Using these `EvalSpec` definitions, we map the label definition for the raw ground-truth to the definition that we use for evaluation.

NOTE: This `EvalSpec` also determines how our merging code expects the predictions. If you do not use the merging code, we expect you to deliver the predictions directly in the 3-channel format, as explained *here*.

Examples for CPP default:

- In `eval_sid2_scene_label`, we list the evaluation ids for the scene-level classes and their labels.
  - Following this, the prediction label for `road` is `7`, `car` is `26`, etc.
- In `eval_pid_flat2scene_part_class`, we list the flat evaluation ids for part-level classes as we expect it in a part segmentation output:

  - Each part has a unique id (unless part grouping is used)
  - Following this, the prediction label for `person-head` is `2`, `rider-head` is `6`, etc.

You can adjust the `EvalSpec` according to your needs, so you can adjust the mappings and the label definition you use for evaluation.

8.1.2 Dataset information

To run the merging scripts, we need to know what images are in a given split of a dataset. Therefore, for each split (e.g., Cityscapes Panoptic Parts val), we create a json file called `images.json`.

This `images.json` follows the format also used in the panopticapi, and contains of:

- A dictionary with the key `'images'`, for which the value is:
  - A list of dictionaries with image information. For each image, the dictionary contains:
    * `file_name`: the file name of the RGB image (NOT the ground-truth file).
    * `image_id`: a unique identifier for each image.
    * `height` and `width`: the pixel dimensions of the RGB image (and ground-truth file).

NOTE: the `image_id` defined here, should be unique, and should be used in the names of all prediction files, as explained later.

To generate the `images.json` file for Cityscapes, run the following script from the main `panoptic_parts` directory:

```
python -m panoptic_parts.evaluation.prepare_data \
    $DATASET_DIR \
    $OUTPUT_DIR \
    $DATASET
```

where

- `$DATASET_DIR`: path to the PPS ground-truths file for the data split (e.g. '~/Cityscapes/gtFinePanopticParts_trainval/gtFinePanopticParts/val')
- `$OUTPUT_DIR`: directory where the images.json file will be stored
- `$DATASET`: dataset name ('Cityscapes' or 'Pascal')

8.2 Get results for subtasks

To generate Part-aware Panoptic Segmentation (PPS) predictions, we need to merge panoptic segmentation and part segmentation predictions. Here, we explain how to retrieve and format the predictions on these subtasks, before merging to PPS.

8.2.1 Panoptic segmentation

There are two options to get panoptic segmentation results:

1. Merge semantic segmentation and instance segmentation predictions. See below how to format and merge these predictions.
2. Do predictions with network that outputs panoptic segmentation results directly.

In the case of option 2, the output needs to be stored in the format as defined for the COCO dataset:

1. A folder with PNG files storing the ids for all predicted segments.
2. A single .json file storing the semantic information for all images.

For more details on the format, check here.

Example Cityscapes Panoptic Parts: for a baseline in our paper, we generate results for Cityscapes using the provided ResNet-50 model from the UPSNet repository.

8.2.2 Semantic segmentation

For semantic segmentation:

- For each image, the semantic segmentation prediction should be stored as a single PNG
- Shape: the shape of the corresponding image, i.e., `2048 x 1024` for Cityscapes.
- Each pixel has one value: the scene-level `category_id`, as defined in the `EvalSpec`.
- Name of the files: should include the unique `image_id` as defined in `images.json`.

Example Cityscapes Panoptic Parts: for a baseline in our paper, we generate results for Cityscapes using the provided Xception-65 model from the official DeepLabv3+ repository.

8.2.3 Instance segmentation

For instance segmentation, we accept two formats:

1. COCO format (as defined here.)
2. Cityscapes format (as defined in the comments for Instance Level Semantic Labeling here.)

For the **COCO format**, we expect:

- A single .json file per image
- Each json file named as `image_id.json`, with the `image_id` as defined in `images.json`.
- The category id in the json file should be the scene-level id as defined in the `EvalSpec`.

For the **Cityscapes format**, we expect:

- A single .txt file per image, containing per-instance info on each line:`relPathPrediction1 labelIDPrediction1 confidencePrediction1`
- The category id, (`labelIDPrediction` in the example), should be the scene-level id as defined in the `EvalSpec`.
- The name of each .txt file contains the `image_id` as defined in `images.json`.
- A singe .png containing with a mask prediction for each individual detected instance.
- See the official Cityscapes repository for more details.

When merging with semantic segmentation to panoptic segmentation, indicate which instance segmentation format ('COCO' or 'Cityscapes') is used.

Example Cityscapes Panoptic Parts: for a baseline in our paper, we generate results for Cityscapes using the official provided ResNet-50-FPN Mask R-CNN model from the Detectron2 repository.

8.2.4 Part segmentation

For part segmentation, we expect predictions in the same format as semantic segmentation:

- For each image, the part segmentation prediction should be stored as a single PNG
- Shape: the shape of the corresponding image, i.e., `2048 x 1024` for Cityscapes.
- Each pixel has one value: the *flat* part-level `category_id`, as defined in the `EvalSpec`.
- Name of the files: should include the unique `image_id` as defined in `images.json`.

Example Cityscapes Panoptic Parts: for a baseline in our paper, we have trained a BSANet model with ResNet-101 backbone on our part annotations for the Cityscapes dataset. These can be downloaded here.

8.3 Merge instance and semantic segmentation to panoptic segmentation

To use the merging script, you need pycocotools and panopticapi.

These can be installed through pip:

```
pip install pycocotools
pip install git+https://github.com/cocodataset/panopticapi.git
```

To merge to panoptic, run the command below. This generates the images and JSON file with the panoptic segmentation predictions in the format as defined here, and saves them in `$OUTPUT_DIR`.

From the main `panoptic_parts` directory, run:

```
python -m panoptic_parts.merging.merge_to_panoptic \
    $EVAL_SPEC_PATH \
    $INST_PRED_PATH \
    $SEM_PRED_PATH \
    $OUTPUT_DIR \
    $IMAGES_JSON \
    --instseg_format=$INSTSEG_FORMAT
```

where

- `$EVAL_SPEC_PATH`: path to the EvalSpec
- `$INST_PRED_PATH`: path where the instance segmentation predictions are stored (a directory when instseg_format='Cityscapes', a JSON file when instseg_format='COCO')
- `$SEM_PRED_PATH`: path where the semantic segmentation predictions are stored
- `$OUTPUT_DIR`: directory where you wish to store the panoptic segmentation predictions
- `$IMAGES_JSON`: the json file with a list of images and corresponding image ids

- `$INSTSEG_FORMAT`: instance segmentation encoding format, i.e., 'COCO' or 'Cityscapes' (optional, default is 'COCO')

8.4 Merge panoptic and part segmentation to PPS

To merge panoptic segmentation and part segmentation to the Part-aware Panoptic Segmentation (PPS) format, run the code below. It stores the PPS predictions as a 3-channel PNG in shape `[height x width x 3]`, where the 3 channels encode the `[scene_category_id, scene_instance_id, part_category_id]`.

From the main `panoptic_parts` directory, run:

```
python -m panoptic_parts.merging.merge_to_pps \
    $EVAL_SPEC_PATH \
    $PANOPTIC_PRED_DIR \
    $PANOPTIC_PRED_JSON \
    $PART_PRED_PATH \
    $IMAGES_JSON \
    $OUTPUT_DIR
```

where

- `$EVAL_SPEC_PATH`: path to the EvalSpec
- `$PANOPTIC_PRED_DIR`: directory where the panoptic segmentation predictions (png files) are stored
- `$PANOPTIC_PRED_JSON`: path to the .json file with the panoptic segmentation predictions
- `$PART_PRED_PATH`: directory where the part predictions are stored
- `$IMAGES_JSON`: the json file with a list of images and corresponding image ids
- `$OUTPUT_DIR`: directory where you wish to store the part-aware panoptic segmentation predictions

8.5 Evaluate results

We provide a step-by-step guide for evaluating PPS results here.

8.6 References and useful links

- Cityscapes dataset
- Cityscapes scripts
- COCO dataset
- COCO API
- COCO Panoptic API
- Pascal VOC 2010 dataset

CHAPTER

NINE

GROUND TRUTH USAGE CASES

We provide for each image a single (image-like) ground truth file encoding semantic-, instance-, and parts- levels annotations. Our compact *Label format* together with *panoptic_parts.utils.format.decode_uids()* function enable easy decoding of the labels for various image understanding tasks including:

```
# labels: Python int, or np.ndarray, or tf.Tensor, or torch.tensor

# Semantic Segmentation
semantic_ids, _, _ = decode_uids(labels)

# Instance Segmentation
semantic_ids, instance_ids, _ = decode_uids(labels)

# Panoptic Segmentation
_, _, _, semantic_instance_ids = decode_uids(labels, return_sids_iids=True)

# Parts Segmentation / Parts Parsing
_, _, _, semantic_parts_ids = decode_uids(labels, return_sids_pids=True)

# Instance-level Parts Parsing
semantic_ids, instance_ids, parts_ids = decode_uids(labels)

# Parts-level Panoptic Segmentation
_, _, _, semantic_instance_ids, semantic_parts_ids = decode_uids(labels, return_sids_
↪iids=True, return_sids_pids=True)
```

CHAPTER

TEN

TOOLS

CHAPTER

ELEVEN

SCRIPTS

CHAPTER

TWELVE

CONTACT

Please feel free to contact us for any suggestions or questions.

panoptic.parts@outlook.com

Correspondence: Panagiotis Meletis, Vincent (Xiaoxiao) Wen

The Panoptic Parts datasets team

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- search

INDEX

Symbols

C

D

E

P

R

S

U

V