cvpods Documentation

Release 0.1

cvpods development team

Nov 03, 2021

CONTENTS

CHAPTER

ONE

API DOCUMENTATION

1.1 cvpods.checkpoint package

class cvpods.checkpoint.**Checkpointer**(*model: torch.nn.modules.module.Module*, *save_dir: str = ''*, *resume: bool = False*, ***, *save_to_disk: bool = True*, ***checkpointables: object*)

Bases: object

A checkpointer that can save/load model as well as extra checkpointable objects.

__init__(*model: torch.nn.modules.module.Module*, *save_dir: str = ''*, *resume: bool = False*, ***, *save_to_disk: bool = True*, ***checkpointables: object*)

Parameters

- **model** (*nn.Module*) – model.
- **save_dir** (*str*) – a directory to save and find checkpoints.
- **save_to_disk** (*bool*) – if True, save checkpoint to disk, otherwise disable saving for this checkpointer.
- **checkpointables** (*object*) – any checkpointable objects, i.e., objects that have the *state_dict()* and *load_state_dict()* method. For example, it can be used like *Checkpointer(model, "dir", optimizer=optimizer)*.

save(*name: str*, *tag_checkpoint: bool = True*, ***kwargs: dict*)

Dump model and checkpointables to a file.

Parameters

- **name** (*str*) – name of the file.
- **kwargs** (*dict*) – extra arbitrary data to save.

load(*path: str*)

Load from the given checkpoint. When path points to network file, this function has to be called on all ranks.

Parameters **path** (*str*) – path or url to the checkpoint. If empty, will not load anything.

Returns *dict* – extra data loaded from the checkpoint that has not been processed. For example, those saved with `save(**extra_data)()`.

has_checkpoint()

Returns *bool* – whether a checkpoint exists in the target directory.

get_checkpoint_file()

Returns *str* – The latest checkpoint file in target directory.

get_all_checkpoint_files()

Returns

list –

All available checkpoint files (.pth files) in target directory.

resume_or_load(*path: str*, *, *resume: bool = True*)

If *resume* is True, this method attempts to resume from the last checkpoint, if exists. Otherwise, load checkpoint from the given path. This is useful when restarting an interrupted training job.

Parameters

- **path** (*str*) – path to the checkpoint.
- **resume** (*bool*) – if True, resume from the last checkpoint if it exists.

Returns same as *load()*.

tag_last_checkpoint(*last_filename_basename: str*)

Tag the last checkpoint.

Parameters **last_filename_basename** (*str*) – the basename of the last filename.

class cvpods.checkpoint.**DefaultCheckpointer**(*model*, *save_dir=''*, *resume=False*, *, *save_to_disk=None*, ***checkpointables*)

Bases: *cvpods.checkpoint.checkpoint.Checkpointer*

Same as *Checkpointer*, but is able to handle models in detectron & cvpods model zoo, and apply conversions for legacy models.

__init__(*model*, *save_dir=''*, *resume=False*, *, *save_to_disk=None*, ***checkpointables*)

Parameters

- **model** (*nn.Module*) – model.
- **save_dir** (*str*) – a directory to save and find checkpoints.
- **resume** (*bool*) – indicate whether to resume from latest checkpoint or start from scratch.
- **save_to_disk** (*bool*) – if True, save checkpoint to disk, otherwise disable saving for this checkpointer.
- **checkpointables** (*object*) – any checkpointable objects, i.e., objects that have the *state_dict()* and *load_state_dict()* method. For example, it can be used like *Checkpointer(model, "dir", optimizer=optimizer)*.

class cvpods.checkpoint.**PeriodicCheckpointer**(*checkpointer: Any*, *period: int*, *max_iter: int = None*, *max_epoch: Optional[int] = None*)

Bases: object

Save checkpoints periodically. When *.step(iteration)* is called, it will execute *checkpointer.save* on the given checkpointer, if iteration is a multiple of period or if *max_iter* is reached.

__init__(*checkpointer: Any*, *period: int*, *max_iter: int = None*, *max_epoch: Optional[int] = None*)

Parameters

- **checkpointer** (*Any*) – the checkpointer object used to save
- **checkpoints.** –
- **period** (*int*) – the period to save checkpoint.

- **max_iter** (*int*) – maximum number of iterations. When it is reached, a checkpoint named "model_final" will be saved.

step(*iteration: int*, ***kwargs: Any*)
Perform the appropriate action at the given iteration.

Parameters

- **iteration** (*int*) – the current iteration, ranged in [0, max_iter-1].
- **kwargs** (*Any*) – extra data to save, same as in *Checkpointer.save()*.

save(*name: str*, ***kwargs: Any*)
Same argument as *Checkpointer.save()*. Use this method to manually save checkpoints outside the schedule.

Parameters

- **name** (*str*) – file name.
- **kwargs** (*Any*) – extra data to save, same as in *Checkpointer.save()*.

1.1.1 cvpods.checkpoint.checkpoint module

class cvpods.checkpoint.checkpoint.**Checkpointer**(*model: torch.nn.modules.module.Module*, *save_dir: str = ''*, *resume: bool = False*, ***, *save_to_disk: bool = True*, ***checkpointables: object*)

Bases: object

A checkpointer that can save/load model as well as extra checkpointable objects.

__init__(*model: torch.nn.modules.module.Module*, *save_dir: str = ''*, *resume: bool = False*, ***, *save_to_disk: bool = True*, ***checkpointables: object*)

Parameters

- **model** (*nn.Module*) – model.
- **save_dir** (*str*) – a directory to save and find checkpoints.
- **save_to_disk** (*bool*) – if True, save checkpoint to disk, otherwise disable saving for this checkpointer.
- **checkpointables** (*object*) – any checkpointable objects, i.e., objects that have the *state_dict()* and *load_state_dict()* method. For example, it can be used like *Checkpointer(model, "dir", optimizer=optimizer)*.

save(*name: str*, *tag_checkpoint: bool = True*, ***kwargs: dict*)
Dump model and checkpointables to a file.

Parameters

- **name** (*str*) – name of the file.
- **kwargs** (*dict*) – extra arbitrary data to save.

load(*path: str*)
Load from the given checkpoint. When path points to network file, this function has to be called on all ranks.

Parameters **path** (*str*) – path or url to the checkpoint. If empty, will not load anything.

Returns *dict* – extra data loaded from the checkpoint that has not been processed. For example, those saved with `save(**extra_data)()`.

has_checkpoint()

Returns *bool* – whether a checkpoint exists in the target directory.

get_checkpoint_file()

Returns *str* – The latest checkpoint file in target directory.

get_all_checkpoint_files()

Returns

list –

All available checkpoint files (.pth files) in target directory.

resume_or_load(*path: str*, *, *resume: bool = True*)

If *resume* is True, this method attempts to resume from the last checkpoint, if exists. Otherwise, load checkpoint from the given path. This is useful when restarting an interrupted training job.

Parameters

- **path** (*str*) – path to the checkpoint.
- **resume** (*bool*) – if True, resume from the last checkpoint if it exists.

Returns same as *load()*.

tag_last_checkpoint(*last_filename_basename: str*)

Tag the last checkpoint.

Parameters **last_filename_basename** (*str*) – the basename of the last filename.

class cvpods.checkpoint.checkpoint.**PeriodicCheckpointer**(*checkpointer: Any*, *period: int*, *max_iter: int = None*, *max_epoch: Optional[int] = None*)

Bases: object

Save checkpoints periodically. When *.step(iteration)* is called, it will execute *checkpointer.save* on the given checkpointer, if iteration is a multiple of period or if *max_iter* is reached.

__init__(*checkpointer: Any*, *period: int*, *max_iter: int = None*, *max_epoch: Optional[int] = None*)

Parameters

- **checkpointer** (*Any*) – the checkpointer object used to save
- **checkpoints.** –
- **period** (*int*) – the period to save checkpoint.
- **max_iter** (*int*) – maximum number of iterations. When it is reached, a checkpoint named "model_final" will be saved.

step(*iteration: int*, ***kwargs: Any*)

Perform the appropriate action at the given iteration.

Parameters

- **iteration** (*int*) – the current iteration, ranged in [0, max_iter-1].
- **kwargs** (*Any*) – extra data to save, same as in *Checkpointer.save()*.

save(*name: str*, ***kwargs: Any*)
Same argument as *Checkpointer.save()*. Use this method to manually save checkpoints outside the schedule.

Parameters

- **name** (*str*) – file name.
- **kwargs** (*Any*) – extra data to save, same as in *Checkpointer.save()*.

class cvpods.checkpoint.checkpoint.**DefaultCheckpointer**(*model*, *save_dir=''*, *resume=False*, ***, *save_to_disk=None*, ***checkpointables*)

Bases: *cvpods.checkpoint.checkpoint.Checkpointer*

Same as *Checkpointer*, but is able to handle models in detectron & cvpods model zoo, and apply conversions for legacy models.

__init__(*model*, *save_dir=''*, *resume=False*, ***, *save_to_disk=None*, ***checkpointables*)

Parameters

- **model** (*nn.Module*) – model.
- **save_dir** (*str*) – a directory to save and find checkpoints.
- **resume** (*bool*) – indicate whether to resume from latest checkpoint or start from scratch.
- **save_to_disk** (*bool*) – if True, save checkpoint to disk, otherwise disable saving for this checkpointer.
- **checkpointables** (*object*) – any checkpointable objects, i.e., objects that have the *state_dict()* and *load_state_dict()* method. For example, it can be used like *Checkpointer(model, "dir", optimizer=optimizer)*.

1.2 cvpods.configs package

class cvpods.configs.base_config.**ConfigDict**(*d=None*, ***kwargs*)

Bases: dict

funcname_not_in_attr()

update(*e=None*, ***f*)

pop(*k*, *d=None*)

merge(*config=None*, ***kwargs*)
merge all key and values of config as BaseConfig's attributes. Note that kwargs will override values in config if they have the same keys

Parameters **config** (*dict*) – custom config dict

merge_from_list(*cfg_list*)
Merge config (keys, values) in a list (e.g., from command line) into this config dict.

Parameters

- **cfg_list** (*list*) – cfg_list must be divided exactly.
- **example, cfg_list = ['FOO.BAR', 0.5]** (*For*) –

diff(*cfg*) → dict
diff given config with current config object

Parameters **cfg** (ConfigDict) – given config, could be any subclass of ConfigDict

Returns *ConfigDict* – contains all diff pair

find(*key: str*, *show=True*, *color='\x1b[30m\x1b[43m'*) → dict
find a given key and its value in config

Parameters

- **key** (*str*) – the string you want to find
- **show** (*bool*) – if show is True, print find result; or return the find result
- **color** (*str*) – color of *key*, default color is black(foreground) yellow(background)

Returns *dict* – if show is False, return dict that contains all find result

Example:

```
>>> from config import config        # suppose you are in your training dir
>>> config.find("weights")
```

clear() → None. Remove all items from D.

copy() → a shallow copy of D

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get()
Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

values() → an object providing a view on D's values

class cvpods.configs.base_config.**BaseConfig**(*d=None*, ***kwargs*)
Bases: *cvpods.configs.base_config.ConfigDict*

link_log(*link_name='log'*)
create a softlink to output dir.

Parameters **link_name** (*str*) – name of softlink

funcname_not_in_attr()

clear() → None. Remove all items from D.

copy() → a shallow copy of D

diff(*cfg*) → dict
diff given config with current config object

Parameters **cfg** (ConfigDict) – given config, could be any subclass of ConfigDict

Returns *ConfigDict* – contains all diff pair

find(*key: str*, *show=True*, *color='\x1b[30m\x1b[43m'*) → dict
find a given key and its value in config

Parameters

- **key** (*str*) – the string you want to find
- **show** (*bool*) – if show is True, print find result; or return the find result
- **color** (*str*) – color of *key*, default color is black(foreground) yellow(background)

Returns *dict* – if show is False, return dict that contains all find result

Example:

```
>>> from config import config        # suppose you are in your training dir
>>> config.find("weights")
```

fromkeys()
Create a new dictionary with keys from iterable and values set to value.

get()
Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

merge(*config=None*, ***kwargs*)
merge all key and values of config as BaseConfig's attributes. Note that kwargs will override values in config if they have the same keys

Parameters **config** (*dict*) – custom config dict

merge_from_list(*cfg_list*)
Merge config (keys, values) in a list (e.g., from command line) into this config dict.

Parameters

- **cfg_list** (*list*) – cfg_list must be divided exactly.
- **example, cfg_list = ['FOO.BAR', 0.5]** (*For*) –

pop(*k*, *d=None*)

popitem() → (k, v), remove and return some (key, value) pair as a
2-tuple; but raise KeyError if D is empty.

setdefault()
Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

update(*e=None*, ***f*)

values() → an object providing a view on D's values

1.2.1 cvpods.configs.config_helper module

cvpods.configs.config_helper.**highlight**(*keyword*, *target*, *color='\x1b[30m\x1b[43m'*)
use given color to highlight keyword in target string

Parameters

- **keyword** (*str*) – highlight string
- **target** (*str*) – target string
- **color** (*str*) – string represent the color, use black foreground
- **yellow background as default** (*and*) –

Returns (str) target string with keyword highlighted

cvpods.configs.config_helper.**find_key**(*param_dict: dict*, *key: str*) → dict
find key in dict

Parameters

- **param_dict** (*dict*) –
- **key** (*str*) –

Returns (dict)

Examples::

```
>>> d = dict(abc=2, ab=4, c=4)
>>> find_key(d, "ab")
{'abc': 2, 'ab':4}
```

cvpods.configs.config_helper.**diff_dict**(*src*, *dst*)
find difference between src dict and dst dict

Parameters

- **src** (*dict*) – src dict
- **dst** (*dict*) – dst dict

Returns (dict) dict contains all the difference key

cvpods.configs.config_helper.**version_update**(*config*)
Backward compatibility of old config's Augmentation pipelines and Optimizer configs; Convert old format into new ones.

1.3 cvpods.data package

cvpods.data.**build_dataset**(*config*, *dataset_names*, *transforms=[]*, *is_train=True*)
dataset_names: List[str], in which elemements must be in format of "dataset_task_version"

cvpods.data.**build_test_loader**(*cfg*)
Similar to *build_train_loader*. But this function uses the given *dataset_name* argument (instead of the names in cfg), and uses batch size 1.

Parameters **cfg** – a cvpods config dict

Returns *DataLoader* – a torch DataLoader, that loads the given detection dataset, with test-time transformation and batching.

cvpods.data.**build_train_loader**(*cfg*)
A data loader is created by the following steps: 1. Use the dataset names in config to query DatasetCatalog, and obtain a list of dicts. 2. Start workers to work on the dicts. Each worker will:

- Map each metadata dict into another format to be consumed by the model.
- Batch them by simply putting dicts into a list.

The batched list[mapped_dict] is what this dataloader will return.

Parameters **cfg** (*config dict*) – the config

Returns an infinite iterator of training data

cvpods.data.**build_transform_gens**(*pipelines*)
Create a list of TransformGen from config.

Transform list is a list of tuple which includes Transform name and parameters. :param pipelines: cfg.INPUT.TRAIN_PIPELINES and cfg.INPUT.TEST_PIPELINES are used here

Returns *list[TransformGen]* – a list of several TransformGen.

class cvpods.data.**ConcatDataset**(*datasets*)
Bases: torch.utils.data.dataset.ConcatDataset

A wrapper of concatenated dataset. Same as torch.utils.data.dataset.ConcatDataset, but concat the group flag for image aspect ratio. :param datasets: A list of datasets. :type datasets: list[Dataset]

class cvpods.data.**RepeatDataset**(*dataset*, *times*)
Bases: object

A wrapper of repeated dataset. The length of repeated dataset will be *times* larger than the original dataset. This is useful when the data loading time is long but the dataset is small. Using RepeatDataset can reduce the data loading time between epochs. :param dataset: The dataset to be repeated. :type dataset: Dataset :param times: Repeat times. :type times: int

1.3.1 cvpods.data.catalog module

1.3.2 cvpods.data.detection_utils module

exception cvpods.data.detection_utils.**SizeMismatchError**
Bases: ValueError

When loaded image has difference width/height compared with annotation.

cvpods.data.detection_utils.**convert_PIL_to_numpy**(*image*, *format*)
Convert PIL image to numpy array of target format. :param image: a PIL image :type image: PIL.Image :param format: the format of output image :type format: str

Returns *(np.ndarray)* – also see *read_image*

cvpods.data.detection_utils.**convert_image_to_rgb**(*image*, *format*)
Convert an image from given format to RGB. :param image: an HWC image :type image: np.ndarray or Tensor :param format: the format of input image, also see *read_image* :type format: str

Returns *(np.ndarray)* – (H,W,3) RGB image in 0-255 range, can be either float or uint8

cvpods.data.detection_utils.**read_image**(*file_name*, *format=None*)
Read an image into the given format. Will apply rotation and flipping if the image has such exif information. :param file_name: image file path :type file_name: str :param format: one of the supported image modes in PIL, or "BGR" or "YUV-BT.601". :type format: str

Returns

image (np.ndarray) –

an HWC image in the given format, which is 0-255, uint8 for supported image modes in PIL or "BGR"; float (0-1 for Y) for YUV-BT.601.

cvpods.data.detection_utils.**check_image_size**(*dataset_dict*, *image*)

Raise an error if the image does not match the size specified in the dict.

cvpods.data.detection_utils.**transform_proposals**(*dataset_dict*, *image_shape*, *transforms*, *min_box_side_len*, *proposal_topk*)

Apply transformations to the proposals in dataset_dict, if any.

Parameters

- **dataset_dict** (*dict*) – a dict read from the dataset, possibly contains fields "proposal_boxes", "proposal_objectness_logits", "proposal_bbox_mode"
- **image_shape** (*tuple*) – height, width
- **transforms** (TransformList) –
- **min_box_side_len** (*int*) – keep proposals with at least this size
- **proposal_topk** (*int*) – only keep top-K scoring proposals

The input dict is modified in-place, with abovementioned keys removed. A new key "proposals" will be added. Its value is an *Instances* object which contains the transformed proposals in its field "proposal_boxes" and "objectness_logits".

cvpods.data.detection_utils.**annotations_to_instances**(*annos*, *image_size*, *mask_format='polygon'*)

Create an Instances object used by the models, from instance annotations in the dataset dict.

Parameters

- **annos** (*list[dict]*) – a list of instance annotations in one image, each element for one instance.
- **image_size** (*tuple*) – height, width

Returns *Instances* – It will contain fields "gt_boxes", "gt_classes", "gt_masks", "gt_keypoints", if they can be obtained from *annos*. This is the format that builtin models expect.

cvpods.data.detection_utils.**annotations_to_instances_rotated**(*annos*, *image_size*)

Create an Instances object used by the models, from instance annotations in the dataset dict. Compared to *annotations_to_instances*, this function is for rotated boxes only

Parameters

- **annos** (*list[dict]*) – a list of instance annotations in one image, each element for one instance.
- **image_size** (*tuple*) – height, width

Returns *Instances* – Containing fields "gt_boxes", "gt_classes", if they can be obtained from *annos*. This is the format that builtin models expect.

cvpods.data.detection_utils.**filter_empty_instances**(*instances*, *by_box=True*, *by_mask=True*)

Filter out empty instances in an *Instances* object.

Parameters

- **instances** (Instances) –

- **by_box** (*bool*) – whether to filter out instances with empty boxes
- **by_mask** (*bool*) – whether to filter out instances with empty masks

Returns *Instances* – the filtered instances.

cvpods.data.detection_utils.**create_keypoint_hflip_indices**(*dataset_names*, *meta*)

Parameters **dataset_names** (*list[str]*) – list of dataset names

Returns *ndarray[int]* – a vector of size=#keypoints, storing the horizontally-flipped keypoint indices.

cvpods.data.detection_utils.**gen_crop_transform_with_instance**(*crop_size*, *image_size*, *instance*)

Generate a CropTransform so that the cropping region contains the center of the given instance.

Parameters

- **crop_size** (*tuple*) – h, w in pixels
- **image_size** (*tuple*) – h, w
- **instance** (*dict*) – an annotation dict of one instance, in cvpods's dataset format.

cvpods.data.detection_utils.**check_metadata_consistency**(*key*, *dataset_names*, *meta*)

Check that the datasets have consistent metadata.

Parameters

- **key** (*str*) – a metadata key
- **dataset_names** (*list[str]*) – a list of dataset names

Raises

- **AttributeError** – if the key does not exist in the metadata
- **ValueError** – if the given datasets do not have the same metadata values defined by key

cvpods.data.detection_utils.**check_sample_valid**(*args*)

cvpods.data.detection_utils.**imdecode**(*data*, *, *require_chl3=True*, *require_alpha=False*)

decode images in common formats (jpg, png, etc.) :param data: encoded image data :type data: bytes :param require_chl3: whether to convert gray image to 3-channel BGR image :param require_alpha: whether to add alpha channel to BGR image :rtype: numpy.ndarray

1.3.3 cvpods.data.datasets module

class cvpods.data.datasets.**CityPersonsDataset**(*cfg*, *dataset_name*, *transforms=[]*, *is_train=True*)

Bases: cvpods.data.base_dataset.BaseDataset

__getitem__(*index*)

Load data, apply transforms, converto to Instances.

evaluate(*predictions*)

Dataset must provide a evaluation function to evaluate model.

property ground_truth_annotations

class cvpods.data.datasets.**CityScapesDataset**(*cfg*, *dataset_name*, *transforms=[]*, *is_train=True*)

Bases: cvpods.data.base_dataset.BaseDataset

__getitem__(*index*)
Load data, apply transforms, converto to Instances.

evaluate(*predictions*)
Dataset must provide a evaluation function to evaluate model.

property ground_truth_annotations

class cvpods.data.datasets.**COCODataset**(*cfg*, *dataset_name*, *transforms=[]*, *is_train=True*)
Bases: cvpods.data.base_dataset.BaseDataset

__getitem__(*index*)
Load data, apply transforms, converto to Instances.

evaluate(*predictions*)
Dataset must provide a evaluation function to evaluate model.

property ground_truth_annotations

class cvpods.data.datasets.**CrowdHumanDataset**(*cfg*, *dataset_name*, *transforms=[]*, *is_train=True*)
Bases: cvpods.data.base_dataset.BaseDataset

__getitem__(*index*)
Load data, apply transforms, converto to Instances.

evaluate(*predictions*)
Dataset must provide a evaluation function to evaluate model.

property ground_truth_annotations

class cvpods.data.datasets.**ImageNetDataset**(*cfg*, *dataset_name*, *transforms=[]*, *is_train=True*)
Bases: cvpods.data.base_dataset.BaseDataset

__getitem__(*index*)
Load data, apply transforms, converto to Instances.

class cvpods.data.datasets.**ImageNetLTDataset**(*cfg*, *dataset_name*, *transforms=[]*, *is_train=True*)
Bases: cvpods.data.base_dataset.BaseDataset

__getitem__(*index*)
Load data, apply transforms, converto to Instances.

class cvpods.data.datasets.**LVISDataset**(*cfg*, *dataset_name*, *transforms=[]*, *is_train=True*)
Bases: cvpods.data.base_dataset.BaseDataset

__getitem__(*index*)
Load data, apply transforms, converto to Instances.

evaluate(*predictions*)
Dataset must provide a evaluation function to evaluate model.

property ground_truth_annotations

class cvpods.data.datasets.**Objects365Dataset**(*cfg*, *dataset_name*, *transforms=[]*, *is_train=True*)
Bases: cvpods.data.base_dataset.BaseDataset

__getitem__(*index*)
Load data, apply transforms, converto to Instances.

evaluate(*predictions*)
Dataset must provide a evaluation function to evaluate model.

property ground_truth_annotations

class cvpods.data.datasets.**CIFAR10Dataset**(*cfg*, *dataset_name*, *transforms*, *is_train=True*, ***kwargs*)
Bases: torchvision.datasets.cifar.CIFAR10

class cvpods.data.datasets.**STL10Datasets**(*cfg*, *dataset_name*, *transforms=[]*, *is_train=True*, ***kwargs*)
Bases: torchvision.datasets.stl10.STL10

class cvpods.data.datasets.**VOCDataset**(*cfg*, *dataset_name*, *transforms=[]*, *is_train=True*)
Bases: cvpods.data.base_dataset.BaseDataset

__getitem__(*index*)
Load data, apply transforms, converto to Instances.

class cvpods.data.datasets.**WiderFaceDataset**(*cfg*, *dataset_name*, *transforms=[]*, *is_train=True*)
Bases: cvpods.data.base_dataset.BaseDataset

__getitem__(*index*)
Load data, apply transforms, converto to Instances.

evaluate(*predictions*)
Dataset must provide a evaluation function to evaluate model.

property ground_truth_annotations

1.3.4 cvpods.data.samplers module

class cvpods.data.samplers.**DistributedGroupSampler**(*dataset*, *samples_per_gpu=1*, *num_replicas=None*, *rank=None*)
Bases: torch.utils.data.sampler.Sampler

Sampler that restricts data loading to a subset of the dataset. It is especially useful in conjunction with torch.nn.parallel.DistributedDataParallel. In such case, each process can pass a DistributedSampler instance as a DataLoader sampler, and load a subset of the original dataset that is exclusive to it. .. note:

```
Dataset is assumed to be of constant size.
```

__init__(*dataset*, *samples_per_gpu=1*, *num_replicas=None*, *rank=None*)

Parameters

- **dataset** (*Dataset*) – Dataset used for sampling.
- **num_replicas** (*optional*) – Number of processes participating in distributed training.
- **rank** (*optional*) – Rank of the current process within num_replicas.

set_epoch(*epoch*)

class cvpods.data.samplers.**InferenceSampler**(*size: int*)
Bases: torch.utils.data.sampler.Sampler

Produce indices for inference. Inference needs to run on the __exact__ set of samples, therefore when the total number of samples is not divisible by the number of workers, this sampler produces different number of samples on different workers.

__init__(*size: int*)

Parameters **size** (*int*) – the total number of data of the underlying dataset to sample from

class cvpods.data.samplers.**RepeatFactorTrainingSampler**(*dataset*, *repeat_thresh*, *shuffle=True*, *seed=None*)

Bases: torch.utils.data.sampler.Sampler

Similar to TrainingSampler, but suitable for training on class imbalanced datasets like LVIS. In each epoch, an image may appear multiple times based on its "repeat factor". The repeat factor for an image is a function of the frequency the rarest category labeled in that image. The "frequency of category c" in [0, 1] is defined as the fraction of images in the training set (without repeats) in which category c appears.

See https://arxiv.org/abs/1908.03195 (>= v2) Appendix B.2.

__init__(*dataset*, *repeat_thresh*, *shuffle=True*, *seed=None*)

Parameters

- **dataset** (*Dataset*) – dataset used for sampling.
- **repeat_thresh** (*float*) – frequency threshold below which data is repeated.
- **shuffle** (*bool*) – whether to shuffle the indices or not.
- **seed** (*int*) – the initial seed of the shuffle. Must be the same across all workers. If None, will use a random seed shared among workers (require synchronization among all workers).

1.3.5 cvpods.data.transforms module

class cvpods.data.transforms.**ExpandTransform**(*left*, *top*, *ratio*, *mean=0, 0, 0*)

Bases: cvpods.data.transforms.transform.Transform

Expand the image and boxes according the specified expand ratio.

__init__(*left*, *top*, *ratio*, *mean=0, 0, 0*)

Parameters

- **top** (*left,*) – crop the image by img[top: top+h, left:left+w].
- **ratio** (*float*) – image expand ratio.
- **mean** (*tuple*) – mean value of dataset.

apply_image(*img*)

Randomly place the original image on a canvas of 'ratio' x original image size filled with mean values. The ratio is in the range of ratio_range.

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray

Apply expand transform on coordinates.

Parameters **coords** (*ndarray*) – floating point array of shape Nx2. Each row is (x, y).

Returns *ndarray* – expand coordinates.

class cvpods.data.transforms.**AffineTransform**(*src*, *dst*, *output_size*, *pad_value=[0, 0, 0]*)

Bases: cvpods.data.transforms.transform.Transform

Augmentation from CenterNet

__init__(*src*, *dst*, *output_size*, *pad_value=[0, 0, 0]*)

output_size:(w, h)

apply_image(*img: numpy.ndarray*) → numpy.ndarray

Apply AffineTransform for the image(s).

Parameters **img** (*ndarray*) – of shape HxW, HxWxC, or NxHxWxC. The array can be of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255].

Returns *ndarray* – the image(s) after applying affine transform.

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray
Affine the coordinates.

Parameters **coords** (*ndarray*) – floating point array of shape Nx2. Each row is (x, y).

Returns *ndarray* – the flipped coordinates.

Note: The inputs are floating point coordinates, not pixel indices. Therefore they are flipped by *(W - x, H - y)*, not *(W - 1 - x, H 1 - y)*.

---

class cvpods.data.transforms.**BlendTransform**(*src_image: numpy.ndarray*, *src_weight: float*, *dst_weight: float*)
Bases: cvpods.data.transforms.transform.Transform

Transforms pixel colors with PIL enhance functions.

__init__(*src_image: numpy.ndarray*, *src_weight: float*, *dst_weight: float*)
Blends the input image (dst_image) with the src_image using formula: `src_weight * src_image + dst_weight * dst_image`

Parameters

- **src_image** (*ndarray*) – Input image is blended with this image
- **src_weight** (*float*) – Blend weighting of src_image
- **dst_weight** (*float*) – Blend weighting of dst_image

apply_image(*img: numpy.ndarray*, *interp: str = None*) → numpy.ndarray
Apply blend transform on the image(s).

Parameters

- **img** (*ndarray*) – of shape NxHxWxC, or HxWxC or HxW. The array can be of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255].
- **interp** (*str*) – keep this option for consistency, perform blend would not require interpolation.

Returns *ndarray* – blended image(s).

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray
Apply no transform on the coordinates.

apply_segmentation(*segmentation: numpy.ndarray*) → numpy.ndarray
Apply no transform on the full-image segmentation.

class cvpods.data.transforms.**IoUCropTransform**(*x0: int*, *y0: int*, *w: int*, *h: int*)
Bases: cvpods.data.transforms.transform.Transform

Perform crop operations on images.

This crop operation will checks whether the center of each instance's bbox is in the cropped image.

__init__(*x0: int*, *y0: int*, *w: int*, *h: int*)

Parameters **y0, w, h** (*x0,*) – crop the image(s) by img[y0:y0+h, x0:x0+w].

apply_image(*img: numpy.ndarray*) → numpy.ndarray
Crop the image(s).

Parameters **img** (*ndarray*) – of shape NxHxWxC, or HxWxC or HxW. The array can be of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255].

Returns *ndarray* – cropped image(s).

apply_box(*box: numpy.ndarray*) → numpy.ndarray
Apply the transform on an axis-aligned box. By default will transform the corner points and use their minimum/maximum to create a new axis-aligned box. Note that this default may change the size of your box, e.g. in rotations.

Parameters **box** (*ndarray*) – Nx4 floating point array of XYXY format in absolute coordinates.

Returns *ndarray* – box after apply the transformation.

Note: The coordinates are not pixel indices. Coordinates on an image of shape (H, W) are in range [0, W] or [0, H].

---

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray
Apply crop transform on coordinates.

Parameters **coords** (*ndarray*) – floating point array of shape Nx2. Each row is (x, y).

Returns *ndarray* – cropped coordinates.

apply_polygons(*polygons: list*) → list
Apply crop transform on a list of polygons, each represented by a Nx2 array. It will crop the polygon with the box, therefore the number of points in the polygon might change.

Parameters **polygon** (*list[ndarray]*) – each is a Nx2 floating point array of (x, y) format in absolute coordinates.

Returns *ndarray* – cropped polygons.

class cvpods.data.transforms.**CropTransform**(*x0: int*, *y0: int*, *w: int*, *h: int*)
Bases: cvpods.data.transforms.transform.Transform

Perform crop operations on images.

__init__(*x0: int*, *y0: int*, *w: int*, *h: int*)

Parameters **y0, w, h** (*x0,*) – crop the image(s) by img[y0:y0+h, x0:x0+w].

apply_image(*img: numpy.ndarray*) → numpy.ndarray
Crop the image(s).

Parameters **img** (*ndarray*) – of shape NxHxWxC, or HxWxC or HxW. The array can be of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255].

Returns *ndarray* – cropped image(s).

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray
Apply crop transform on coordinates.

Parameters **coords** (*ndarray*) – floating point array of shape Nx2. Each row is (x, y).

Returns *ndarray* – cropped coordinates.

apply_polygons(*polygons: list*) → list
Apply crop transform on a list of polygons, each represented by a Nx2 array. It will crop the polygon with the box, therefore the number of points in the polygon might change.

Parameters **polygon** (*list[ndarray]*) – each is a Nx2 floating point array of (x, y) format in absolute coordinates.

Returns *ndarray* – cropped polygons.

class cvpods.data.transforms.**CropPadTransform**(*x0: int*, *y0: int*, *w: int*, *h: int*, *new_w: int*, *new_h: int*, *img_value=None*, *seg_value=None*)

Bases: cvpods.data.transforms.transform.Transform

get_pad_offset(*ori: int*, *tar: int*)

apply_image(*img: numpy.ndarray*) → numpy.ndarray

Crop and Pad the image(s).

Parameters **img** (*ndarray*) – of shape NxHxWxC, or HxWxC or HxW. The array can be of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255].

Returns *ndarray* – cropped and padded image(s).

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray

Apply crop and pad transform on coordinates.

Parameters **coords** (*ndarray*) – floating point array of shape Nx2. Each row is (x, y).

Returns *ndarray* – cropped and padded coordinates.

apply_polygons(*polygons: list*) → list

Apply crop and pad transform on a list of polygons, each represented by a Nx2 array.

Parameters **polygon** (*list[ndarray]*) – each is a Nx2 floating point array of (x, y) format in absolute coordinates.

Returns *ndarray* – cropped and padded polygons.

apply_segmentation(*segmentation: numpy.ndarray*) → numpy.ndarray

Apply crop and pad transform on the full-image segmentation.

Parameters **segmentation** (*ndarray*) – of shape HxW. The array should have integer or bool dtype.

Returns *ndarray* – cropped and padded segmentation.

class cvpods.data.transforms.**GridSampleTransform**(*grid: numpy.ndarray*, *interp: str*)

Bases: cvpods.data.transforms.transform.Transform

__init__(*grid: numpy.ndarray*, *interp: str*)

Parameters

- **grid** (*ndarray*) – grid has x and y input pixel locations which are used to compute output. Grid has values in the range of [-1, 1], which is normalized by the input height and width. The dimension is *N x H x W x 2*.
- **interp** (*str*) – interpolation methods. Options include *nearest* and *bilinear*.

apply_image(*img: numpy.ndarray*, *interp: str = None*) → numpy.ndarray

Apply grid sampling on the image(s).

Parameters

- **img** (*ndarray*) – of shape NxHxWxC, or HxWxC or HxW. The array can be of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255].
- **interp** (*str*) – interpolation methods. Options include *nearest* and *bilinear*.

Returns *ndarray* – grid sampled image(s).

apply_coords(*coords: numpy.ndarray*)
Not supported.

apply_segmentation(*segmentation: numpy.ndarray*) → numpy.ndarray
Apply grid sampling on the full-image segmentation.

Parameters **segmentation** (*ndarray*) – of shape HxW. The array should have integer or bool dtype.

Returns *ndarray* – grid sampled segmentation.

class cvpods.data.transforms.**RotationTransform**(*h*, *w*, *angle*, *expand=True*, *center=None*, *interp=None*)
Bases: cvpods.data.transforms.transform.Transform

This method returns a copy of this image, rotated the given number of degrees counter clockwise around its center.

__init__(*h*, *w*, *angle*, *expand=True*, *center=None*, *interp=None*)

Parameters

- **w** (*h,*) – original image size
- **angle** (*float*) – degrees for rotation
- **expand** (*bool*) – choose if the image should be resized to fit the whole rotated image (default), or simply cropped
- **center** (*tuple (width, height)*) – coordinates of the rotation center if left to None, the center will be fit to the center of each image center has no effect if expand=True because it only affects shifting
- **interp** – cv2 interpolation method, default cv2.INTER_LINEAR

apply_image(*img*, *interp=None*)
img should be a numpy array, formatted as Height * Width * Nchannels

apply_coords(*coords*)
coords should be a N * 2 array-like, containing N couples of (x, y) points

apply_segmentation(*segmentation*)

create_rotation_matrix(*offset=0*)

inverse()
The inverse is to rotate it back with expand, and crop to get the original shape.

class cvpods.data.transforms.**HFlipTransform**(*width: int*)
Bases: cvpods.data.transforms.transform.Transform

Perform horizontal flip.

apply_image(*img: numpy.ndarray*) → numpy.ndarray
Flip the image(s).

Parameters **img** (*ndarray*) – of shape HxW, HxWxC, or NxHxWxC. The array can be of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255].

Returns *ndarray* – the flipped image(s).

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray
Flip the coordinates.

Parameters **coords** (*ndarray*) – floating point array of shape Nx2. Each row is (x, y).

Returns *ndarray* – the flipped coordinates.

Note: The inputs are floating point coordinates, not pixel indices. Therefore they are flipped by *(W - x, H - y)*, not *(W - 1 - x, H 1 - y)*.

apply_rotated_box(*rotated_boxes*)
Apply the horizontal flip transform on rotated boxes.

Parameters **rotated_boxes** (*ndarray*) – Nx5 floating point array of (x_center, y_center, width, height, angle_degrees) format in absolute coordinates.

class cvpods.data.transforms.**VFlipTransform**(*height: int*)
Bases: cvpods.data.transforms.transform.Transform

Perform vertical flip.

apply_image(*img: numpy.ndarray*) → numpy.ndarray
Flip the image(s).

Parameters **img** (*ndarray*) – of shape HxW, HxWxC, or NxHxWxC. The array can be of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255].

Returns *ndarray* – the flipped image(s).

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray
Flip the coordinates.

Parameters **coords** (*ndarray*) – floating point array of shape Nx2. Each row is (x, y).

Returns *ndarray* – the flipped coordinates.

Note: The inputs are floating point coordinates, not pixel indices. Therefore they are flipped by *(W - x, H - y)*, not *(W - 1 - x, H - 1 - y)*.

class cvpods.data.transforms.**NoOpTransform**
Bases: cvpods.data.transforms.transform.Transform

A transform that does nothing.

apply_image(*img: numpy.ndarray*) → numpy.ndarray

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray

apply_rotated_box(*x*)

class cvpods.data.transforms.**ScaleTransform**(*h: int*, *w: int*, *new_h: int*, *new_w: int*, *interp: str = 'BILINEAR'*)
Bases: cvpods.data.transforms.transform.Transform

Resize the image to a target size.

__init__(*h: int*, *w: int*, *new_h: int*, *new_w: int*, *interp: str = 'BILINEAR'*)

Parameters

- **w** (*h,*) – original image size.
- **new_w** (*new_h,*) – new image size.
- **interp** (*str*) – the interpolation method. Options includes: * “NEAREST” * “BILINEAR” * “BICUBIC” * “LANCZOS” * “HAMMING” * “BOX”

apply_image(*img: numpy.ndarray*, *interp: str = None*) → numpy.ndarray
Resize the image(s).

Parameters **img** (*ndarray*) – of shape NxHxWxC, or HxWxC or HxW. The array can be of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255].

Returns *ndarray* – resized image(s).

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray
Compute the coordinates after resize.

Parameters **coords** (*ndarray*) – floating point array of shape Nx2. Each row is (x, y).

Returns *ndarray* – resized coordinates.

apply_segmentation(*segmentation: numpy.ndarray*) → numpy.ndarray
Apply resize on the full-image segmentation.

Parameters **segmentation** (*ndarray*) – of shape HxW. The array should have integer or bool dtype.

Returns *ndarray* – resized segmentation.

class cvpods.data.transforms.**DistortTransform**(*hue*, *saturation*, *exposure*, *image_format*)
Bases: cvpods.data.transforms.transform.Transform

Distort image w.r.t hue, saturation and exposure.

apply_image(*img: numpy.ndarray*) → numpy.ndarray

Parameters **img** (*ndarray*) – of shape HxW, HxWxC, or NxHxWxC. The array can be of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255].

Returns *ndarray* – the distorted image(s).

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray

apply_segmentation(*segmentation: numpy.ndarray*) → numpy.ndarray

class cvpods.data.transforms.**Transform**
Bases: object

Base class for implementations of __deterministic__ transformations for image and other data structures. "Deterministic" requires that the output of all methods of this class are deterministic w.r.t their input arguments. In training, there should be a higher-level policy that generates (likely with random variations) these transform ops. Each transform op may handle several data types, e.g.: image, coordinates, segmentation, bounding boxes. Some of them have a default implementation, but can be overwritten if the default isn't appropriate. The implementation of each method may choose to modify its input data in-place for efficient transformation.

abstract apply_image(*img: numpy.ndarray*)
Apply the transform on an image.

Parameters **img** (*ndarray*) – of shape NxHxWxC, or HxWxC or HxW. The array can be of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255].

Returns *ndarray* – image after apply the transformation.

abstract apply_coords(*coords: numpy.ndarray*)
Apply the transform on coordinates.

Parameters **coords** (*ndarray*) – floating point array of shape Nx2. Each row is (x, y).

Returns *ndarray* – coordinates after apply the transformation.

Note: The coordinates are not pixel indices. Coordinates on an image of shape (H, W) are in range [0, W] or [0, H].

---

apply_segmentation (*segmentation: numpy.ndarray*) → numpy.ndarray
Apply the transform on a full-image segmentation. By default will just perform "apply_image".

Parameters

- **segmentation** (*ndarray*) – of shape HxW. The array should have integer
- **bool dtype.** (*or*) –

Returns *ndarray* – segmentation after apply the transformation.

apply_box (*box: numpy.ndarray*) → numpy.ndarray
Apply the transform on an axis-aligned box. By default will transform the corner points and use their minimum/maximum to create a new axis-aligned box. Note that this default may change the size of your box, e.g. in rotations.

Parameters **box** (*ndarray*) – Nx4 floating point array of XYXY format in absolute coordinates.

Returns *ndarray* – box after apply the transformation.

Note: The coordinates are not pixel indices. Coordinates on an image of shape (H, W) are in range [0, W] or [0, H].

---

apply_polygons (*polygons: list*) → list
Apply the transform on a list of polygons, each represented by a Nx2 array. By default will just transform all the points.

Parameters **polygon** (*list[ndarray]*) – each is a Nx2 floating point array of (x, y) format in absolute coordinates.

Returns *list[ndarray]* – polygon after apply the transformation.

Note: The coordinates are not pixel indices. Coordinates on an image of shape (H, W) are in range [0, W] or [0, H].

__call__ (*image*, *annotations=None*, ***kwargs*)
Apply transfrom to images and annotations (if exist)

classmethod register_type (*data_type: str*, *func: Callable*)
Register the given function as a handler that this transform will use for a specific data type.

Parameters

- **data_type** (*str*) – the name of the data type (e.g., box)
- **func** (*callable*) – takes a transform and a data, returns the transformed data.

Examples:

```
def func(flip_transform, voxel_data):
    return transformed_voxel_data
HFlipTransform.register_type("voxel", func)

# ...
transform = HFlipTransform(...)
transform.apply_voxel(voxel_data)  # func will be called
```

class cvpods.data.transforms.**TransformList** (*transforms: list*)
Bases: object

Maintain a list of transform operations which will be applied in sequence. .. attribute:: transforms

type list[Transform]

__init__(*transforms: list*)

Parameters **transforms** (*list[Transform]*) – list of transforms to perform.

__getattr__(*name: str*)

Parameters **name** (*str*) – name of the attribute.

__add__(*other: cvpods.data.transforms.transform.TransformList*) → cvpods.data.transforms.transform.TransformList

Parameters **other** (TransformList) – transformation to add.

Returns *TransformList* – list of transforms.

__iadd__(*other: cvpods.data.transforms.transform.TransformList*) → cvpods.data.transforms.transform.TransformList

Parameters **other** (TransformList) – transformation to add.

Returns *TransformList* – list of transforms.

__radd__(*other: cvpods.data.transforms.transform.TransformList*) → cvpods.data.transforms.transform.TransformList

Parameters **other** (TransformList) – transformation to add.

Returns *TransformList* – list of transforms.

insert(*idx: int*, *other: cvpods.data.transforms.transform.TransformList*) → cvpods.data.transforms.transform.TransformList

Parameters

- **idx** (*int*) – insert position.
- **other** (TransformList) – transformation to insert.

Returns None

class cvpods.data.transforms.**ExtentTransform**(*src_rect*, *output_size*, *interp=2*, *fill=0*)

Bases: cvpods.data.transforms.transform.Transform

Extracts a subregion from the source image and scales it to the output size.

The fill color is used to map pixels from the source rect that fall outside the source image.

See: https://pillow.readthedocs.io/en/latest/PIL.html#PIL.ImageTransform.ExtentTransform

__init__(*src_rect*, *output_size*, *interp=2*, *fill=0*)

Parameters

- **src_rect** (*x0, y0, x1, y1*) – src coordinates
- **output_size** (*h, w*) – dst image size
- **interp** – PIL interpolation methods
- **fill** – Fill color used when src_rect extends outside image

apply_image(*img*, *interp=None*)

apply_coords(*coords*)

apply_segmentation(*segmentation*)

class cvpods.data.transforms.**ResizeTransform**(*h*, *w*, *new_h*, *new_w*, *interp*)
Bases: cvpods.data.transforms.transform.Transform

Resize the image to a target size.

__init__(*h*, *w*, *new_h*, *new_w*, *interp*)

Parameters

- **w** (*h,*) – original image size
- **new_w** (*new_h,*) – new image size
- **interp** – PIL interpolation methods

apply_image(*img*, *interp=None*)

apply_coords(*coords*)

apply_segmentation(*segmentation*)

apply_rotated_box(*rotated_boxes*)
Apply the resizing transform on rotated boxes. For details of how these (approximation) formulas are derived, please refer to RotatedBoxes.scale().

Parameters **rotated_boxes** (*ndarray*) – Nx5 floating point array of (x_center, y_center, width, height, angle_degrees) format in absolute coordinates.

class cvpods.data.transforms.**GaussianBlurTransform**(*sigma*, *p=1.0*)
Bases: cvpods.data.transforms.transform.Transform

GaussianBlur using PIL.ImageFilter.GaussianBlur

__init__(*sigma*, *p=1.0*)

Parameters

- **sigma** (*List(float)*) – sigma of gaussian
- **p** (*float*) – probability of perform this augmentation

apply_image(*img: numpy.ndarray*) → numpy.ndarray

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray

class cvpods.data.transforms.**GaussianBlurConvTransform**(*kernel_size*, *p=1.0*)
Bases: cvpods.data.transforms.transform.Transform

apply_image(*img: numpy.ndarray*) → numpy.ndarray

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray

class cvpods.data.transforms.**SolarizationTransform**(*thresh=128*, *p=0.5*)
Bases: cvpods.data.transforms.transform.Transform

apply_image(*img: numpy.ndarray*) → numpy.ndarray

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray

class cvpods.data.transforms.**ComposeTransform**(*tfms*)
Bases: object

Composes several transforms together.

__init__(*tfms*)

Parameters **transforms** (*list[Transform]*) – list of transforms to compose.

class cvpods.data.transforms.**LabSpaceTransform**
Bases: cvpods.data.transforms.transform.Transform

Convert image from RGB into Lab color space

apply_image(*img: numpy.ndarray*) → numpy.ndarray

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray

class cvpods.data.transforms.**PadTransform**(*top: int*, *left: int*, *target_h: int*, *target_w: int*, *pad_value=0*, *seg_value=255*)
Bases: cvpods.data.transforms.transform.Transform

Pad image with *pad_value* to the specified *target_h* and *target_w*.

Adds *top* rows of *pad_value* on top, *left* columns of *pad_value* on the left, and then pads the image on the bottom and right with *pad_value* until it has dimensions *target_h*, *target_w*.

This op does nothing if *top* and *left* is zero and the image already has size *target_h* by *target_w*.

__init__(*top: int*, *left: int*, *target_h: int*, *target_w: int*, *pad_value=0*, *seg_value=255*)

Parameters

- **top** (*int*) – number of rows of *pad_value* to add on top.
- **left** (*int*) – number of columns of *pad_value* to add on the left.
- **target_h** (*int*) – height of output image.
- **target_w** (*int*) – width of output image.
- **pad_value** (*int*) – the value used to pad the image.
- **seg_value** (*int*) – the value used to pad the semantic seg annotaions.

apply_image(*img: numpy.ndarray*, *pad_value=None*) → numpy.ndarray

apply_coords(*coords: numpy.ndarray*) → numpy.ndarray

apply_segmentation(*segmentation: numpy.ndarray*) → numpy.ndarray
Apply pad transform on the full-image segmentation.

Parameters **segmentation** (*ndarray*) – of shape HxW. The array should have integer or bool dtype.

Returns *ndarray* – padded segmentation.

class cvpods.data.transforms.**Pad**(*top*, *left*, *target_h*, *target_w*, *pad_value=0*)
Bases: cvpods.data.transforms.transform_gen.TransformGen

Pad image with *pad_value* to the specified *target_h* and *target_w*.

Adds *top* rows of *pad_value* on top, *left* columns of *pad_value* on the left, and then pads the image on the bottom and right with *pad_value* until it has dimensions *target_h*, *target_w*.

This op does nothing if *top* and *left* is zero and the image already has size *target_h* by *target_w*.

__init__(*top*, *left*, *target_h*, *target_w*, *pad_value=0*)

Parameters

- **top** (*int*) – number of rows of *pad_value* to add on top.
- **left** (*int*) – number of columns of *pad_value* to add on the left.
- **target_h** (*int*) – height of output image.
- **target_w** (*int*) – width of output image.

- **pad_value** (*int*) – the value used to pad the image.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RandomScale**(*output_size*, *ratio_range=0.1, 2*, *interp='BILINEAR'*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Randomly scale the image according to the specified output size and scale ratio range.

This transform has the following three steps:

1. select a random scale factor according to the specified scale ratio range.
2. recompute the accurate scale_factor using rounded scaled image size.
3. select non-zero random offset (x, y) if scaled image is larger than output_size.

__init__(*output_size*, *ratio_range=0.1, 2*, *interp='BILINEAR'*)

Parameters

- **output_size** (*tuple*) – image output size.
- **ratio_range** (*tuple*) – range of scale ratio.
- **interp** (*str*) – the interpolation method. Options includes: * “NEAREST” * “BILINEAR” * “BICUBIC” * “LANCZOS” * “HAMMING” * “BOX”

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**Expand**(*ratio_range=1, 4*, *mean=0, 0, 0*, *prob=0.5*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Random Expand the image & bboxes.

__init__(*ratio_range=1, 4*, *mean=0, 0, 0*, *prob=0.5*)

Parameters

- **ratio_range** (*tuple*) – range of expand ratio.
- **mean** (*tuple*) – mean value of dataset.
- **prob** (*float*) – probability of applying this transformation.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**MinIoURandomCrop**(*min_ious=0.1, 0.3, 0.5, 0.7, 0.9*, *min_crop_size=0.3*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Random crop the image & bboxes, the cropped patches have minimum IoU requirement with original image & bboxes, the IoU threshold is randomly selected from min_ious.

__init__(*min_ious=0.1, 0.3, 0.5, 0.7, 0.9*, *min_crop_size=0.3*)

Parameters

- **min_ious** (*tuple*) – minimum IoU threshold for all intersections with bounding boxes
- **min_crop_size** (*float*) – minimum crop’s size (i.e. h,w := a*h, a*w, where a >= min_crop_size).

get_transform(*img*, *annotations*)

Parameters

- **img** (*ndarray*) – of shape HxWxC(RGB). The array can be of type uint8 in range [0, 255], or floating point in range [0, 255].
- **annotations** (*list[dict[str->str]]*) –

  Each item in the list is a bbox label of an object. The object is represented by a dict,

  which contains:

  - bbox (list): bbox coordinates, top left and bottom right.
  - **bbox_mode (str): bbox label mode, for example: *XYXY_ABS*,** *XYWH_ABS* and so on...

class cvpods.data.transforms.**RandomSwapChannels**(*prob=0.5*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Randomly swap image channels.

__init__(*prob=0.5*)

Parameters **prob** (*float*) – probability of swap channels.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**CenterAffine**(*boarder*, *output_size*, *pad_value=[0, 0, 0]*, *random_aug=True*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Affine Transform for CenterNet

__init__(*boarder*, *output_size*, *pad_value=[0, 0, 0]*, *random_aug=True*)

output_size (w, h) shape

get_transform(*img*, *annotations=None*)

generate_center_and_scale(*img_shape*)

generate center shpae : (h, w)

static generate_src_and_dst(*center*, *scale*, *output_size*)

class cvpods.data.transforms.**RandomBrightness**(*intensity_min*, *intensity_max*, *prob=1.0*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Randomly transforms image brightness.

Brightness intensity is uniformly sampled in (intensity_min, intensity_max). - intensity < 1 will reduce brightness - intensity = 1 will preserve the input image - intensity > 1 will increase brightness

See: https://pillow.readthedocs.io/en/3.0.x/reference/ImageEnhance.html

__init__(*intensity_min*, *intensity_max*, *prob=1.0*)

Parameters

- **intensity_min** (*float*) – Minimum augmentation.
- **intensity_max** (*float*) – Maximum augmentation.
- **prob** (*float*) – probability of transforms image brightness.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RandomContrast**(*intensity_min*, *intensity_max*, *prob=1.0*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Randomly transforms image contrast.

Contrast intensity is uniformly sampled in (intensity_min, intensity_max). - intensity < 1 will reduce contrast - intensity = 1 will preserve the input image - intensity > 1 will increase contrast

See: https://pillow.readthedocs.io/en/3.0.x/reference/ImageEnhance.html

__init__(*intensity_min*, *intensity_max*, *prob=1.0*)

Parameters

- **intensity_min** (*float*) – Minimum augmentation.
- **intensity_max** (*float*) – Maximum augmentation.
- **prob** (*float*) – probability of transforms image contrast.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RandomCrop**(*crop_type: str*, *crop_size*, *strict_mode=True*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Randomly crop a subimage out of an image.

__init__(*crop_type: str*, *crop_size*, *strict_mode=True*)

Parameters

- **crop_type** (*str*) – one of "relative_range", "relative", "absolute". See *config/defaults.py* for explanation.
- **crop_size** (*tuple[float]*) – the relative ratio or absolute pixels of height and width
- **strict_mode** (*bool*) – if *True*, the target *crop_size* must be smaller than the original image size.

get_transform(*img*, *annotations=None*)

get_crop_size(*image_size*)

Parameters **image_size** (*tuple*) – height, width

Returns *crop_size (tuple)* – height, width in absolute pixels

class cvpods.data.transforms.**RandomCropWithInstance**(*crop_type: str*, *crop_size*, *strict_mode=True*)

Bases: cvpods.data.transforms.transform_gen.RandomCrop

Make sure the cropping region contains the center of a random instance from annotations.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RandomCropWithMaxAreaLimit**(*crop_type: str*, *crop_size*, *strict_mode=True*, *single_category_max_area=1.0*, *ignore_value=255*)

Bases: cvpods.data.transforms.transform_gen.RandomCrop

Find a cropping window such that no single category occupies more than *single_category_max_area* in *sem_seg*.

The function retries random cropping 10 times max.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RandomCropPad**(*crop_type: str*, *crop_size*, *img_value=None*, *seg_value=None*)

Bases: cvpods.data.transforms.transform_gen.RandomCrop

Randomly crop and pad a subimage out of an image.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RandomExtent**(*scale_range*, *shift_range*)
Bases: cvpods.data.transforms.transform_gen.TransformGen

Outputs an image by cropping a random "subrect" of the source image.

The subrect can be parameterized to include pixels outside the source image, in which case they will be set to zeros (i.e. black). The size of the output image will vary with the size of the random subrect.

__init__(*scale_range*, *shift_range*)

Parameters

- **scale_range** (*l, h*) – Range of input-to-output size scaling factor.
- **shift_range** (*x, y*) – Range of shifts of the cropped subrect. The rect is shifted by [w / 2 * Uniform(-x, x), h / 2 * Uniform(-y, y)], where (w, h) is the (width, height) of the input image. Set each component to zero to crop at the image's center.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RandomFlip**(*prob=0.5*, ***, *horizontal=True*, *vertical=False*)
Bases: cvpods.data.transforms.transform_gen.TransformGen

Flip the image horizontally or vertically with the given probability.

__init__(*prob=0.5*, ***, *horizontal=True*, *vertical=False*)

Parameters

- **prob** (*float*) – probability of flip.
- **horizontal** (*boolean*) – whether to apply horizontal flipping
- **vertical** (*boolean*) – whether to apply vertical flipping

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RandomSaturation**(*intensity_min*, *intensity_max*, *prob=1.0*)
Bases: cvpods.data.transforms.transform_gen.TransformGen

Randomly transforms image saturation.

Saturation intensity is uniformly sampled in (intensity_min, intensity_max). - intensity < 1 will reduce saturation (make the image more grayscale) - intensity = 1 will preserve the input image - intensity > 1 will increase saturation

See: https://pillow.readthedocs.io/en/3.0.x/reference/ImageEnhance.html

__init__(*intensity_min*, *intensity_max*, *prob=1.0*)

Parameters

- **intensity_min** (*float*) – Minimum augmentation (1 preserves input).
- **intensity_max** (*float*) – Maximum augmentation (1 preserves input).
- **prob** (*float*) – probability of transforms image saturation.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RandomLighting**(*scale*)
Bases: cvpods.data.transforms.transform_gen.TransformGen

Randomly transforms image color using fixed PCA over ImageNet.

The degree of color jittering is randomly sampled via a normal distribution, with standard deviation given by the scale parameter.

__init__(*scale*)

Parameters **scale** (*float*) – Standard deviation of principal component weighting.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RandomDistortion**(*hue*, *saturation*, *exposure*, *image_format='BGR'*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Random distort image's hue, saturation and exposure.

__init__(*hue*, *saturation*, *exposure*, *image_format='BGR'*)

RandomDistortion Initialization. :param hue: value of hue :type hue: float :param saturation: value of saturation :type saturation: float :param exposure: value of exposure :type exposure: float

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**Resize**(*shape*, *interp=2*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Resize image to a target size

__init__(*shape*, *interp=2*)

Parameters

- **shape** – (h, w) tuple or a int.
- **interp** – PIL interpolation method.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**ResizeShortestEdge**(*short_edge_length*, *max_size=9223372036854775807*, *sample_style='range'*, *interp=2*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Scale the shorter edge to the given size, with a limit of *max_size* on the longer edge. If *max_size* is reached, then downscale so that the longer edge does not exceed max_size.

__init__(*short_edge_length*, *max_size=9223372036854775807*, *sample_style='range'*, *interp=2*)

Parameters

- **short_edge_length** (*list[int]*) – If `sample_style=="range"`, a [min, max] interval from which to sample the shortest edge length. If `sample_style=="choice"`, a list of shortest edge lengths to sample from.
- **max_size** (*int*) – maximum allowed longest edge length.
- **sample_style** (*str*) – either "range" or "choice".
- **interp** – PIL interpolation method.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**ResizeLongestEdge**(*long_edge_length*, *sample_style='range'*, *interp=2*, *jitter=0.0, 32*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Scale the longer edge to the given size.

__init__(*long_edge_length*, *sample_style='range'*, *interp=2*, *jitter=0.0, 32*)

Parameters

- **long_edge_length** (*list[int]*) – If `sample_style=="range"`, a [min, max] interval from which to sample the shortest edge length. If `sample_style=="choice"`, a list of shortest edge lengths to sample from.
- **sample_style** (*str*) – either "range" or "choice".
- **interp** – PIL interpolation method.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**ShuffleList**(*transforms*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Randomly shuffle the *transforms* order.

__init__(*transforms*)

Parameters **transforms** (*list[TransformGen]*) – List of transform to be shuffled.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RandomList**(*transforms*, *num_layers=2*, *choice_weights=None*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Random select subset of provided augmentations.

__init__(*transforms*, *num_layers=2*, *choice_weights=None*)

Parameters

- **transforms** (*List[TorchTransformGen]*) – list of transforms need to be performed.
- **num_layers** (*int*) – parameters of np.random.choice.
- **choice_weights** (*optional, float*) – parameters of np.random.choice.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**RepeatList**(*transforms*, *repeat_times*)

Bases: cvpods.data.transforms.transform_gen.TransformGen

Forward several times of provided transforms for a given image.

__init__(*transforms*, *repeat_times*)

Parameters

- **transforms** (*list[TransformGen]*) – List of transform to be repeated.
- **repeat_times** (*int*) – number of duplicates desired.

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**TransformGen**

Bases: object

TransformGen takes an image of type uint8 in range [0, 255], or floating point in range [0, 1] or [0, 255] as input.

It creates a *Transform* based on the given image, sometimes with randomness. The transform can then be used to transform images or other data (boxes, points, annotations, etc.) associated with it.

The assumption made in this class is that the image itself is sufficient to instantiate a transform. When this assumption is not true, you need to create the transforms by your own.

A list of *TransformGen* can be applied with `apply_transform_gens()`.

abstract get_transform(*img*, *annotations=None*)

__repr__()
Produce something like: "MyTransformGen(field1={self.field1}, field2={self.field2})"

__str__()
Produce something like: "MyTransformGen(field1={self.field1}, field2={self.field2})"

class cvpods.data.transforms.**TorchTransformGen**(*tfm*)
Bases: object

Wrapper transfrom of transforms in torchvision. It convert img (np.ndarray) to PIL image, and convert back to np.ndarray after transform.

class cvpods.data.transforms.**GaussianBlur**(*sigma*, *p=1.0*)
Bases: cvpods.data.transforms.transform_gen.TransformGen

Gaussian blur transform.

__init__(*sigma*, *p=1.0*)

Parameters

- **sigma** (*List(float)*) – sigma of gaussian
- **p** (*float*) – probability of perform this augmentation

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**GaussianBlurConv**(*kernel_size*, *p*)
Bases: cvpods.data.transforms.transform_gen.TransformGen

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**Solarization**(*threshold=128*, *p=0.5*)
Bases: cvpods.data.transforms.transform_gen.TransformGen

get_transform(*img*, *annotations=None*)

class cvpods.data.transforms.**AutoAugment**(*name*, *prob=0.5*, *magnitude=10*, *hparams=None*)
Bases: cvpods.data.transforms.transform_gen.TransformGen
Convert any of AutoAugment into a cvpods-fashion Transform such that can be configured in config.py
__init__(*name*, *prob=0.5*, *magnitude=10*, *hparams=None*)

Parameters

- **name** (*str*) – any type of transforms list in _RAND_TRANSFORMS.
- **prob** (*float*) – probability of perform current augmentation.
- **magnitude** (*int*) – intensity / magnitude of each augmentation.
- **hparams** (*dict*) – hyper-parameters required by each augmentation.

get_transform(*img*, *annotations=None*)

1.4 cvpods.engine package

1.4.1 cvpods.engine.hooks module

1.4.2 cvpods.engine.launch module

1.4.3 cvpods.engine.setup module

1.4.4 cvpods.engine.base_runner module

1.4.5 cvpods.engine.runner module

1.4.6 cvpods.engine.predictor module

1.5 cvpods.evaluation package

1.6 cvpods.layers package

class cvpods.layers.**MemoryEfficientSwish**
Bases: torch.nn.modules.module.Module

forward(*x*)

class cvpods.layers.**Swish**
Bases: torch.nn.modules.module.Module

Implement the Swish activation function. See: https://arxiv.org/abs/1710.05941 for more details.

forward(*x*)

class cvpods.layers.**FrozenBatchNorm2d**(*num_features*, *eps=1e-05*)
Bases: torch.nn.modules.module.Module

BatchNorm2d where the batch statistics and the affine parameters are fixed.

It contains non-trainable buffers called "weight" and "bias", "running_mean", "running_var", initialized to perform identity transformation.

The pre-trained backbone models from Caffe2 only contain "weight" and "bias", which are computed from the original four parameters of BN. The affine transform *x * weight + bias* will perform the equivalent computation of *(x - running_mean) / sqrt(running_var) * weight + bias*. When loading a backbone model from Caffe2, "running_mean" and "running_var" will be left unchanged as identity transformation.

Other pre-trained backbone models may contain all 4 parameters.

The forward is implemented by *F.batch_norm(..., training=False)*.

forward(*x*)

classmethod convert_frozen_batchnorm(*module*)
Convert BatchNorm/SyncBatchNorm in module into FrozenBatchNorm.

Parameters **module** (*torch.nn.Module*) –

Returns If module is BatchNorm/SyncBatchNorm, returns a new module. Otherwise, in-place convert module and return it.

Similar to convert_sync_batchnorm in https://github.com/pytorch/pytorch/blob/master/torch/nn/modules/batchnorm.py

class cvpods.layers.**NaiveSyncBatchNorm**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*)

Bases: torch.nn.modules.batchnorm.BatchNorm2d

torch.nn.SyncBatchNorm has known unknown bugs. It produces significantly worse AP (and sometimes goes NaN) when the batch size on each worker is quite different (e.g., when scale augmentation is used, or when it is applied to mask head).

Use this implementation before *nn.SyncBatchNorm* is fixed. It is slower than *nn.SyncBatchNorm*.

forward(*input*)

cvpods.layers.**get_activation**(*activation*)

Parameters **norm** (*str or callable*) –

Returns *nn.Module or None* – the normalization layer

cvpods.layers.**get_norm**(*norm*, *out_channels*)

Parameters **norm** (*str or callable*) –

Returns *nn.Module or None* – the normalization layer

class cvpods.layers.**DeformConv**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *deformable_groups=1*, *bias=False*, *norm=None*, *activation=None*)

Bases: torch.nn.modules.module.Module

__init__(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *deformable_groups=1*, *bias=False*, *norm=None*, *activation=None*)

Deformable convolution.

Arguments are similar to Conv2D. Extra arguments:

Parameters

- **deformable_groups** (*int*) – number of groups used in deformable convolution.
- **norm** (*nn.Module, optional*) – a normalization layer
- **activation** (*callable(Tensor) -> Tensor*) – a callable activation function

forward(*x*, *offset*)

extra_repr()

class cvpods.layers.**ModulatedDeformConv**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *deformable_groups=1*, *bias=True*, *norm=None*, *activation=None*)

Bases: torch.nn.modules.module.Module

__init__(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *groups=1*, *deformable_groups=1*, *bias=True*, *norm=None*, *activation=None*)

Modulated deformable convolution.

Arguments are similar to Conv2D. Extra arguments:

Parameters

- **deformable_groups** (*int*) – number of groups used in deformable convolution.

- **norm** (*nn.Module, optional*) – a normalization layer
- **activation** (*callable(Tensor) -> Tensor*) – a callable activation function

forward(*x*, *offset*, *mask*)

extra_repr()

class cvpods.layers.**DeformConvWithOff**(*in_channels*, *out_channels*, *kernel_size=3*, *stride=1*, *padding=1*, *dilation=1*, *deformable_groups=1*)

Bases: torch.nn.modules.module.Module

forward(*input*)

class cvpods.layers.**ModulatedDeformConvWithOff**(*in_channels*, *out_channels*, *kernel_size=3*, *stride=1*, *padding=1*, *dilation=1*, *deformable_groups=1*)

Bases: torch.nn.modules.module.Module

forward(*input*)

cvpods.layers.**paste_masks_in_image**(*masks*, *boxes*, *image_shape*, *threshold=0.5*)

Paste a set of masks that are of a fixed resolution (e.g., 28 x 28) into an image. The location, height, and width for pasting each mask is determined by their corresponding bounding boxes in boxes.

Parameters

- **masks** (*tensor*) – Tensor of shape (Bimg, Hmask, Wmask), where Bimg is the number of detected object instances in the image and Hmask, Wmask are the mask width and mask height of the predicted mask (e.g., Hmask = Wmask = 28). Values are in [0, 1].
- **boxes** (*Boxes or Tensor*) – A Boxes of length Bimg or Tensor of shape (Bimg, 4). boxes[i] and masks[i] correspond to the same object instance.
- **image_shape** (*tuple*) – height, width
- **threshold** (*float*) – A threshold in [0, 1] for converting the (soft) masks to binary masks.

Returns *img_masks (Tensor)* – A tensor of shape (Bimg, Himage, Wimage), where Bimg is the number of detected object instances and Himage, Wimage are the image width and height. img_masks[i] is a binary mask for object instance i.

cvpods.layers.**batched_nms**(*boxes*, *scores*, *idxs*, *iou_threshold*)

Same as torchvision.ops.boxes.batched_nms, but safer.

cvpods.layers.**batched_nms_rotated**(*boxes*, *scores*, *idxs*, *iou_threshold*)

Performs non-maximum suppression in a batched fashion.

Each index value correspond to a category, and NMS will not be applied between elements of different categories.

Parameters

- **boxes** (*Tensor[N, 5]*) – boxes where NMS will be performed. They are expected to be in (x_ctr, y_ctr, width, height, angle_degrees) format
- **scores** (*Tensor[N]*) – scores for each one of the boxes
- **idxs** (*Tensor[N]*) – indices of the categories for each one of the boxes.
- **iou_threshold** (*float*) – discards all overlapping boxes with IoU < iou_threshold

Returns *Tensor* – int64 tensor with the indices of the elements that have been kept by NMS, sorted in decreasing order of scores

cvpods.layers.**batched_softnms**(*boxes*, *scores*, *idxs*, *iou_threshold*, *score_threshold=0.001*, *soft_mode='gaussian'*)

cvpods.layers.**batched_softnms_rotated**(*boxes*, *scores*, *idxs*, *iou_threshold*, *score_threshold=0.001*, *soft_mode='gaussian'*)

cvpods.layers.**cluster_nms**(*boxes*, *scores*, *iou_threshold*)

cvpods.layers.**generalized_batched_nms**(*boxes*, *scores*, *idxs*, *iou_threshold*, *score_threshold=0.001*, *nms_type='normal'*)

cvpods.layers.**matrix_nms**(*seg_masks*, *cate_labels*, *cate_scores*, *kernel='gaussian'*, *sigma=2.0*, *sum_masks=None*)

Matrix NMS for multi-class masks. See: https://arxiv.org/pdf/2003.10152.pdf for more details.

Parameters

- **seg_masks** (*Tensor*) – shape: [N, H, W], binary masks.
- **cate_labels** (*Tensor*) – shepe: [N], mask labels in descending order.
- **cate_scores** (*Tensor*) – shape [N], mask scores in descending order.
- **kernel** (*str*) – 'linear' or 'gaussian'.
- **sigma** (*float*) – std in gaussian method.
- **sum_masks** (*Tensor*) – The sum of seg_masks.

Returns *Tensor* – cate_scores_update, tensors of shape [N].

cvpods.layers.**nms**(*boxes*, *scores*, *iou_threshold*)

Performs non-maximum suppression (NMS) on the boxes according to their intersection-over-union (IoU).

NMS iteratively removes lower scoring boxes which have an IoU greater than iou_threshold with another (higher scoring) box.

boxes [Tensor[N, 4])] boxes to perform NMS on. They are expected to be in (x1, y1, x2, y2) format

scores [Tensor[N]] scores for each one of the boxes

iou_threshold [float] discards all overlapping boxes with IoU > iou_threshold

keep [Tensor] int64 tensor with the indices of the elements that have been kept by NMS, sorted in decreasing order of scores

cvpods.layers.**nms_rotated**(*boxes*, *scores*, *iou_threshold*)

Performs non-maximum suppression (NMS) on the rotated boxes according to their intersection-over-union (IoU).

Rotated NMS iteratively removes lower scoring rotated boxes which have an IoU greater than iou_threshold with another (higher scoring) rotated box.

Note that RotatedBox (5, 3, 4, 2, -90) covers exactly the same region as RotatedBox (5, 3, 4, 2, 90) does, and their IoU will be 1. However, they can be representing completely different objects in certain tasks, e.g., OCR.

As for the question of whether rotated-NMS should treat them as faraway boxes even though their IOU is 1, it depends on the application and/or ground truth annotation.

As an extreme example, consider a single character v and the square box around it.

If the angle is 0 degree, the object (text) would be read as 'v';

If the angle is 90 degrees, the object (text) would become '>';

If the angle is 180 degrees, the object (text) would become '^';

If the angle is 270/-90 degrees, the object (text) would become '<'

All of these cases have IoU of 1 to each other, and rotated NMS that only uses IoU as criterion would only keep one of them with the highest score - which, practically, still makes sense in most cases because typically only

one of theses orientations is the correct one. Also, it does not matter as much if the box is only used to classify the object (instead of transcribing them with a sequential OCR recognition model) later.

On the other hand, when we use IoU to filter proposals that are close to the ground truth during training, we should definitely take the angle into account if we know the ground truth is labeled with the strictly correct orientation (as in, upside-down words are annotated with -180 degrees even though they can be covered with a 0/90/-90 degree box, etc.)

The way the original dataset is annotated also matters. For example, if the dataset is a 4-point polygon dataset that does not enforce ordering of vertices/orientation, we can estimate a minimum rotated bounding box to this polygon, but there's no way we can tell the correct angle with 100% confidence (as shown above, there could be 4 different rotated boxes, with angles differed by 90 degrees to each other, covering the exactly same region). In that case we have to just use IoU to determine the box proximity (as many detection benchmarks (even for text) do) unless there're other assumptions we can make (like width is always larger than height, or the object is not rotated by more than 90 degrees CCW/CW, etc.)

In summary, not considering angles in rotated NMS seems to be a good option for now, but we should be aware of its implications.

Parameters

- **boxes** (*Tensor[N, 5]*) – Rotated boxes to perform NMS on. They are expected to be in (x_center, y_center, width, height, angle_degrees) format.
- **scores** (*Tensor[N]*) – Scores for each one of the rotated boxes
- **iou_threshold** (*float*) – Discards all overlapping rotated boxes with IoU < iou_threshold

Returns *keep (Tensor)* – int64 tensor with the indices of the elements that have been kept by Rotated NMS, sorted in decreasing order of scores

cvpods.layers.**softnms**(*boxes*, *scores*, *sigma*, *score_threshold*, *soft_mode='gaussian'*)

cvpods.layers.**softnms_rotated**(*boxes*, *scores*, *sigma*, *score_threshold*, *soft_mode='gaussian'*)

class cvpods.layers.**ROIAlign**(*output_size*, *spatial_scale*, *sampling_ratio*, *aligned=True*)

Bases: torch.nn.modules.module.Module

__init__(*output_size*, *spatial_scale*, *sampling_ratio*, *aligned=True*)

Parameters

- **output_size** (*tuple*) – h, w
- **spatial_scale** (*float*) – scale the input boxes by this number
- **sampling_ratio** (*int*) – number of inputs samples to take for each output sample. 0 to take samples densely.
- **aligned** (*bool*) – if False, use the legacy implementation in Detectron. If True, align the results more perfectly.

Note: The meaning of aligned=True:

Given a continuous coordinate c, its two neighboring pixel indices (in our pixel model) are computed by floor(c - 0.5) and ceil(c - 0.5). For example, c=1.3 has pixel neighbors with discrete indices [0] and [1] (which are sampled from the underlying signal at continuous coordinates 0.5 and 1.5). But the original roi_align (aligned=False) does not subtract the 0.5 when computing neighboring pixel indices and therefore it uses pixels with a slightly incorrect alignment (relative to our pixel model) when performing bilinear interpolation.

With *aligned=True*, we first appropriately scale the ROI and then shift it by -0.5 prior to calling roi_align. This produces the correct neighbors; see cvpods/tests/test_roi_align.py for verification.

The difference does not make a difference to the model's performance if ROIAlign is used together with conv layers.

forward(*input*, *rois*)

Parameters

- **input** – NCHW images
- **rois** – Bx5 boxes. First column is the index into N. The other 4 columns are xyxy.

cvpods.layers.**roi_align**()

class cvpods.layers.**ROIAlignRotated**(*output_size*, *spatial_scale*, *sampling_ratio*)
Bases: torch.nn.modules.module.Module

__init__(*output_size*, *spatial_scale*, *sampling_ratio*)

Parameters

- **output_size** (*tuple*) – h, w
- **spatial_scale** (*float*) – scale the input boxes by this number
- **sampling_ratio** (*int*) – number of inputs samples to take for each output sample. 0 to take samples densely.

Note: ROIAlignRotated supports continuous coordinate by default: Given a continuous coordinate c, its two neighboring pixel indices (in our pixel model) are computed by floor(c - 0.5) and ceil(c - 0.5). For example, c=1.3 has pixel neighbors with discrete indices [0] and [1] (which are sampled from the underlying signal at continuous coordinates 0.5 and 1.5).

forward(*input*, *rois*)

Parameters

- **input** – NCHW images
- **rois** – Bx6 boxes. First column is the index into N. The other 5 columns are (x_ctr, y_ctr, width, height, angle_degrees).

cvpods.layers.**roi_align_rotated**()

class cvpods.layers.**ShapeSpec**
Bases: cvpods.layers.shape_spec._ShapeSpec

A simple structure that contains basic shape specification about a tensor. It is often used as the auxiliary inputs/outputs of models, to obtain the shape inference ability among pytorch modules.

channels

height

width

stride

class cvpods.layers.**SwapAlign2Nat**(*lambda_val*, *pad_val=- 6.0*)
Bases: torch.nn.modules.module.Module

The op *SwapAlign2Nat* described in https://arxiv.org/abs/1903.12174. Given an input tensor that predicts masks of shape (N, C=VxU, H, W), apply the op, it will return masks of shape (N, V'xU', H', W') where the unit lengths of (V, U) and (H, W) are swapped, and the mask representation is transformed from aligned to natural. :param lambda_val: the relative unit length ratio between (V, U) and (H, W), :type lambda_val: int :param as we always have larger unit lengths for: :type as we always have larger unit lengths for: V, U) than (H, W :param lambda_val is always >= 1.: :param pad_val: padding value for the values falling outside of the input :type pad_val: float :param tensor, default set to -6 as sigmoid: :type tensor, default set to -6 as sigmoid: -6 :param that is no masks outside of the tensor.:

forward(*X*)

cvpods.layers.**swap_align2nat**()

class cvpods.layers.**TreeFilterV2**(*guide_channels*, *in_channels*, *embed_channels*, *num_groups=1*, *eps=1e-08*)

Bases: torch.nn.modules.module.Module

num_groups = None
Embedding Layers

gamma = None
Core of Tree Filter

tree_filter_layer = None
Parameters init

reset_parameter()

split_groups(*x*)

expand_groups(*x*)

forward(*feature*, *guide*)

class cvpods.layers.**BatchNorm2d**(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*)

Bases: torch.nn.modules.batchnorm._BatchNorm

Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated per-dimension over the mini-batches and γ and β are learnable parameter vectors of size *C* (where *C* is the input size). By default, the elements of γ are set to 1 and the elements of β are set to 0.

Also by default, during training this layer keeps running estimates of its computed mean and variance, which are then used for normalization during evaluation. The running estimates are kept with a default `momentum` of 0.1.

If `track_running_stats` is set to `False`, this layer then does not keep running estimates, and batch statistics are instead used during evaluation time as well.

Note: This `momentum` argument is different from one used in optimizer classes and the conventional notion of momentum. Mathematically, the update rule for running statistics here is $\hat{x}_{\text{new}} = (1 - \text{momentum}) \times \hat{x} + \text{momentum} \times x_t$, where $\hat{x}$ is the estimated statistic and x_t is the new observed value.

Because the Batch Normalization is done over the *C* dimension, computing statistics on *(N, H, W)* slices, it's common terminology to call this Spatial Batch Normalization.

Parameters

- **num_features** – C from an expected input of size (N, C, H, W)
- **eps** – a value added to the denominator for numerical stability. Default: 1e-5
- **momentum** – the value used for the running_mean and running_var computation. Can be set to `None` for cumulative moving average (i.e. simple average). Default: 0.1
- **affine** – a boolean value that when set to `True`, this module has learnable affine parameters. Default: `True`
- **track_running_stats** – a boolean value that when set to `True`, this module tracks the running mean and variance, and when set to `False`, this module does not track such statistics and always uses batch statistics in both training and eval modes. Default: `True`

Shape:

- Input: (N, C, H, W)
- Output: (N, C, H, W) (same shape as input)

Examples:

```
>>> # With Learnable Parameters
>>> m = nn.BatchNorm2d(100)
>>> # Without Learnable Parameters
>>> m = nn.BatchNorm2d(100, affine=False)
>>> input = torch.randn(20, 100, 35, 45)
>>> output = m(input)
```

class cvpods.layers.**Conv2d**(**args*, ***kwargs*)

Bases: `torch.nn.modules.conv.Conv2d`

A wrapper around `torch.nn.Conv2d` to support empty inputs and more features.

__init__(**args*, ***kwargs*)

Extra keyword arguments supported in addition to those in *torch.nn.Conv2d*:

Parameters

- **norm** (*nn.Module, optional*) – a normalization layer
- **activation** (*callable(Tensor) -> Tensor*) – a callable activation function

It assumes that norm layer is used before activation.

forward(*x*)

bias = None

class cvpods.layers.**Conv2dSamePadding**(**args*, ***kwargs*)

Bases: `torch.nn.modules.conv.Conv2d`

A wrapper around `torch.nn.Conv2d` to support "SAME" padding mode and more features.

__init__(**args*, ***kwargs*)

Extra keyword arguments supported in addition to those in *torch.nn.Conv2d*:

Parameters

- **norm** (*nn.Module, optional*) – a normalization layer

- **activation** (*callable(Tensor) -> Tensor*) – a callable activation function

It assumes that norm layer is used before activation.

forward(*x*)

bias = None

class cvpods.layers.**ConvTranspose2d**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *output_padding=0*, *groups=1*, *bias=True*, *dilation=1*, *padding_mode='zeros'*)

Bases: torch.nn.modules.conv._ConvTransposeNd

Applies a 2D transposed convolution operator over an input image composed of several input planes.

This module can be seen as the gradient of Conv2d with respect to its input. It is also known as a fractionally-strided convolution or a deconvolution (although it is not an actual deconvolution operation).

- `stride` controls the stride for the cross-correlation.
- `padding` controls the amount of implicit zero-paddings on both sides for `dilation * (kernel_size - 1) - padding` number of points. See note below for details.
- `output_padding` controls the additional size added to one side of the output shape. See note below for details.
- `dilation` controls the spacing between the kernel points; also known as the à trous algorithm. It is harder to describe, but this link has a nice visualization of what `dilation` does.
- `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by `groups`. For example,
  - At groups=1, all inputs are convolved to all outputs.
  - At groups=2, the operation becomes equivalent to having two conv layers side by side, each seeing half the input channels, and producing half the output channels, and both subsequently concatenated.
  - At groups= `in_channels`, each input channel is convolved with its own set of filters (of size $\left\lfloor \frac{out\_channels}{in\_channels} \right\rfloor$).

The parameters `kernel_size`, `stride`, `padding`, `output_padding` can either be:

- a single `int` – in which case the same value is used for the height and width dimensions
- a `tuple` of two ints – in which case, the first *int* is used for the height dimension, and the second *int* for the width dimension

---

Note: Depending of the size of your kernel, several (of the last) columns of the input might be lost, because it is a valid cross-correlation, and not a full cross-correlation. It is up to the user to add proper padding.

Note: The `padding` argument effectively adds `dilation * (kernel_size - 1) - padding` amount of zero padding to both sizes of the input. This is set so that when a `Conv2d` and a `ConvTranspose2d` are initialized with same parameters, they are inverses of each other in regard to the input and output shapes. However, when `stride > 1`, `Conv2d` maps multiple input shapes to the same output shape. `output_padding` is provided to resolve this ambiguity by effectively increasing the calculated output shape on one side. Note that `output_padding` is only used to find output shape, but does not actually add zero-padding to output.

Parameters

- **in_channels** (*int*) – Number of channels in the input image
- **out_channels** (*int*) – Number of channels produced by the convolution

- **kernel_size** (*int or tuple*) – Size of the convolving kernel
- **stride** (*int or tuple, optional*) – Stride of the convolution. Default: 1
- **padding** (*int or tuple, optional*) – `dilation * (kernel_size - 1) - padding` zero-padding will be added to both sides of each dimension in the input. Default: 0
- **output_padding** (*int or tuple, optional*) – Additional size added to one side of each dimension in the output shape. Default: 0
- **groups** (*int, optional*) – Number of blocked connections from input channels to output channels. Default: 1
- **bias** (*bool, optional*) – If `True`, adds a learnable bias to the output. Default: `True`
- **dilation** (*int or tuple, optional*) – Spacing between kernel elements. Default: 1

Shape:

- Input: $(N, C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ where

$$H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{dilation}[0] \times (\text{kernel\_size}[0] - 1) + \text{output\_padding}[0] + 1$$

$$W_{out} = (W_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{dilation}[1] \times (\text{kernel\_size}[1] - 1) + \text{output\_padding}[1] + 1$$

weight
the learnable weights of the module of shape $(\text{in\_channels}, \frac{\text{out\_channels}}{\text{groups}}, \text{kernel\_size}[0], \text{kernel\_size}[1])$. The values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{out}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$

Type Tensor

bias
the learnable bias of the module of shape (out_channels) If *bias* is `True`, then the values of these weights are sampled from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{groups}{C_{\text{out}} * \prod_{i=0}^{1} \text{kernel\_size}[i]}$

Type Tensor

Examples:

```
>>> # With square kernels and equal stride
>>> m = nn.ConvTranspose2d(16, 33, 3, stride=2)
>>> # non-square kernels and unequal stride and with padding
>>> m = nn.ConvTranspose2d(16, 33, (3, 5), stride=(2, 1), padding=(4, 2))
>>> input = torch.randn(20, 16, 50, 100)
>>> output = m(input)
>>> # exact output size can be also specified as an argument
>>> input = torch.randn(1, 16, 12, 12)
>>> downsample = nn.Conv2d(16, 16, 3, stride=2, padding=1)
>>> upsample = nn.ConvTranspose2d(16, 16, 3, stride=2, padding=1)
>>> h = downsample(input)
>>> h.size()
torch.Size([1, 16, 6, 6])
>>> output = upsample(h, output_size=input.size())
>>> output.size()
torch.Size([1, 16, 12, 12])
```

forward(*input*, *output_size=None*)

bias = None

class cvpods.layers.**DisAlignLinear**(*in_features: int*, *out_features: int*, *bias: bool = True*)
Bases: torch.nn.modules.linear.Linear

A wrapper for nn.Linear with support of DisAlign method.

forward(*input: torch.Tensor*)

class cvpods.layers.**DisAlignNormalizedLinear**(*in_features: int*, *out_features: int*, *bias: bool = False*, ***args*)
Bases: cvpods.layers.wrappers.NormalizedLinear

A wrapper for nn.Linear with support of DisAlign method.

forward(*input: torch.Tensor*)

class cvpods.layers.**MaxPool2dSamePadding**(**args*, ***kwargs*)
Bases: torch.nn.modules.pooling.MaxPool2d

A wrapper around torch.nn.MaxPool2d to support "SAME" padding mode and more features.

See: https://github.com/pytorch/pytorch/issues/3867

forward(*x*)

class cvpods.layers.**NormalizedConv2d**(**args*, ***kwargs*)
Bases: torch.nn.modules.conv.Conv2d

A wrapper around torch.nn.Conv2d to support empty inputs and more features.

__init__(**args*, ***kwargs*)
Extra keyword arguments supported in addition to those in *torch.nn.Conv2d*:

Parameters

- **norm** (*nn.Module, optional*) – a normalization layer
- **activation** (*callable(Tensor) -> Tensor*) – a callable activation function

It assumes that norm layer is used before activation.

extra_repr()

forward(*x*)

bias = None

class cvpods.layers.**NormalizedLinear**(*in_features*, *out_features*, *bias=False*, *feat_norm=True*, *scale_mode='learn'*, *scale_init=1.0*)
Bases: torch.nn.modules.module.Module

A advanced Linear layer which supports weight normalization or cosine normalization.

reset_parameters()

forward(*inputs*)

Parameters **inputs** (*torch.Tensor*) – (N, C)

Returns *output (torch.Tensor)* – (N, D)

extra_repr()

class cvpods.layers.**SeparableConvBlock**(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *bias=True*, *norm=None*, *activation=None*)
Bases: torch.nn.modules.module.Module

Depthwise seperable convolution block.

__init__(*in_channels*, *out_channels*, *kernel_size*, *stride=1*, *padding=0*, *dilation=1*, *bias=True*, *norm=None*, *activation=None*)

Parameters

- **in_channels** (*int*) – the number of input tensor channels.
- **out_channels** (*int*) – the number of output tensor channels.
- **kernel_size** (*int*) – the kernel size.
- **stride** (*int or tuple or list*) – the stride.
- **bias** (*bool*) – if *True*, the pointwise conv applies bias.
- **apply_bn** (*bool*) – if *True*, apply BN layer after conv layer.
- **norm** (*nn.Module, optional*) – a normalization layer
- **activation** (*callable(Tensor) -> Tensor*) – a callable activation function

It assumes that norm layer is used before activation.

forward(*inputs*)

cvpods.layers.**cat**(*tensors*, *dim=0*)

Efficient version of torch.cat that avoids a copy if there is only a single element in a list

cvpods.layers.**interpolate**(*input*, *size=None*, *scale_factor=None*, *mode='nearest'*, *align_corners=None*)

A wrapper around `torch.nn.functional.interpolate()` to support zero-size tensor.

1.7 cvpods.modeling package

1.7.1 cvpods.modeilng.anchor_generator module

1.7.2 cvpods.modeilng.backbone module

1.7.3 cvpods.modeilng.basenet module

1.7.4 cvpods.modeilng.box_regression module

1.7.5 cvpods.modeilng.losses module

1.7.6 cvpods.modeilng.matcher module

1.7.7 cvpods.modeilng.meta_arch module

1.7.8 cvpods.modeilng.nn_utils module

1.7.9 cvpods.modeilng.poolers module

1.7.10 cvpods.modeilng.postprocessing module

1.7.11 cvpods.modeilng.proposal_generator module

1.7.12 cvpods.modeilng.roi_heads module

1.7.13 cvpods.modeilng.sampling module

1.7.14 cvpods.modeilng.test_time_augmentation module

1.8 cvpods.solver package

cvpods.solver.**build_lr_scheduler**(*cfg*, *optimizer: torch.optim.optimizer.Optimizer*, ***kwargs*) → torch.optim.lr_scheduler._LRScheduler

Build a LR scheduler from config.

cvpods.solver.**build_optimizer**(*cfg*, *model: torch.nn.modules.module.Module*) → torch.optim.optimizer.Optimizer

Build an optimizer with clip and LARS wraper from config.

class cvpods.solver.**AdamBuilder**

Bases: cvpods.solver.optimizer_builder.OptimizerBuilder

static build(*model*, *cfg*)

class cvpods.solver.**AdamWBuilder**

Bases: cvpods.solver.optimizer_builder.OptimizerBuilder

static build(*model*, *cfg*)

class cvpods.solver.**OptimizerBuilder**

Bases: object

static build(*model*, *cfg*)

class cvpods.solver.**SGDBuilder**
Bases: cvpods.solver.optimizer_builder.OptimizerBuilder

static build(*model*, *cfg*)

class cvpods.solver.**SGDGateLRBuilder**
Bases: cvpods.solver.optimizer_builder.OptimizerBuilder

SGD Gate LR optimizer builder, used for DynamicRouting in cvpods. This optimizer will ultiply lr for gating function.

static build(*model*, *cfg*)

class cvpods.solver.**BaseSchedulerBuilder**
Bases: object

static build(*optimizer*, *cfg*, ***kwargs*)

class cvpods.solver.**LambdaLRBuilder**
Bases: cvpods.solver.scheduler_builder.BaseSchedulerBuilder

static build(*optimizer*, *cfg*, ***kwargs*)

class cvpods.solver.**OneCycleLRBuilder**
Bases: cvpods.solver.scheduler_builder.BaseSchedulerBuilder

static build(*optimizer*, *cfg*, ***kwargs*)

class cvpods.solver.**PolyLRBuilder**
Bases: cvpods.solver.scheduler_builder.BaseSchedulerBuilder

static build(*optimizer*, *cfg*, ***kwargs*)

class cvpods.solver.**WarmupCosineLR**(*optimizer: torch.optim.optimizer.Optimizer*, *max_iters: int*, *warmup_factor: float = 0.001*, *warmup_iters: int = 1000*, *warmup_method: str = 'linear'*, *last_epoch: int = - 1*, *epoch_iters: int = - 1*)
Bases: torch.optim.lr_scheduler._LRScheduler

__init__(*optimizer: torch.optim.optimizer.Optimizer*, *max_iters: int*, *warmup_factor: float = 0.001*, *warmup_iters: int = 1000*, *warmup_method: str = 'linear'*, *last_epoch: int = - 1*, *epoch_iters: int = - 1*)
Cosine LR with warmup

Parameters

- **optimizer** (*Optimizer*) – Wrapped optimizer.
- **max_iters** (*int*) – max num of iters
- **warmup_factor** (*float*) – warmup factor to compute lr
- **warmup_iters** (*int*) – warmup iters
- **warmup_method** (*str*) – warmup method in [“constant”, “linear”, “burnin”]
- **last_epoch** – The index of last epoch. Default: -1.

get_lr() → List[float]

class cvpods.solver.**WarmupCosineLRBuilder**
Bases: cvpods.solver.scheduler_builder.BaseSchedulerBuilder

static build(*optimizer*, *cfg*, ***kwargs*)

class cvpods.solver.**WarmupMultiStepLR**(*optimizer: torch.optim.optimizer.Optimizer*, *milestones: List[int]*, *gamma: float = 0.1*, *warmup_factor: float = 0.001*, *warmup_iters: int = 1000*, *warmup_method: str = 'linear'*, *last_epoch: int = - 1*)

Bases: torch.optim.lr_scheduler._LRScheduler

__init__(*optimizer: torch.optim.optimizer.Optimizer*, *milestones: List[int]*, *gamma: float = 0.1*, *warmup_factor: float = 0.001*, *warmup_iters: int = 1000*, *warmup_method: str = 'linear'*, *last_epoch: int = - 1*)

Multi Step LR with warmup

Parameters

- **optimizer** (*torch.optim.Optimizer*) – optimizer used.
- **milestones** (*list[Int]*) – a list of increasing integers.
- **gamma** (*float*) – gamma
- **warmup_factor** (*float*) – lr = warmup_factor * base_lr
- **warmup_iters** (*int*) – iters to warmup
- **warmup_method** (*str*) – warmup method in ["constant", "linear", "burnin"]
- **last_epoch** (*int*) – The index of last epoch. Default: -1.

get_lr() → List[float]

class cvpods.solver.**WarmupMultiStepLRBuilder**

Bases: cvpods.solver.scheduler_builder.BaseSchedulerBuilder

static build(*optimizer*, *cfg*, ***kwargs*)

1.9 cvpods.structures package

class cvpods.structures.**Boxes**(*tensor: torch.Tensor*)

Bases: object

This structure stores a list of boxes as a Nx4 torch.Tensor. It supports some common methods about boxes (*area*, *clip*, *nonempty*, etc), and also behaves like a Tensor (support indexing, *to(device)*, *.device*, and iteration over all boxes)

tensor

float matrix of Nx4.

Type torch.Tensor

BoxSizeType = typing.Union[typing.List[int], typing.Tuple[int, int]]

__init__(*tensor: torch.Tensor*)

Parameters **tensor** (*Tensor[float]*) – a Nx4 matrix. Each row is (x1, y1, x2, y2).

clone() → cvpods.structures.boxes.Boxes

Clone the Boxes.

Returns Boxes

to(*device: str*) → cvpods.structures.boxes.Boxes

area() → torch.Tensor

Computes the area of all the boxes.

Returns *torch.Tensor* – a vector with areas of each box.

clip(*box_size: Union[List[int], Tuple[int, int]]*) → None
Clip (in place) the boxes by limiting x coordinates to the range [0, width] and y coordinates to the range [0, height].

Parameters **box_size** (*height, width*) – The clipping box's size.

nonempty(*threshold: int = 0*) → torch.Tensor
Find boxes that are non-empty. A box is considered empty, if either of its side is no larger than threshold.

Returns *Tensor* – a binary vector which represents whether each box is empty (False) or non-empty (True).

__getitem__(*item: Union[int, slice, torch.BoolTensor]*) → cvpods.structures.boxes.Boxes

Returns *Boxes* – Create a new *Boxes* by indexing.

The following usage are allowed:

1. *new_boxes = boxes[3]*: return a *Boxes* which contains only one box.
2. *new_boxes = boxes[2:10]*: return a slice of boxes.

3. *new_boxes = boxes[vector]*, where vector is a torch.BoolTensor with *length = len(boxes)*. Nonzero elements in the vector will be selected.

Note that the returned Boxes might share storage with this Boxes, subject to Pytorch's indexing semantics.

inside_box(*box_size: Union[List[int], Tuple[int, int]], boundary_threshold: int = 0*) → torch.Tensor

Parameters

- **box_size** (*height, width*) – Size of the reference box.
- **boundary_threshold** (*int*) – Boxes that extend beyond the reference box boundary by more than boundary_threshold are considered "outside".

Returns a binary vector, indicating whether each box is inside the reference box.

get_centers() → torch.Tensor

Returns The box centers in a Nx2 array of (x, y).

scale(*scale_x: float*, *scale_y: float*) → None
Scale the box with horizontal and vertical scaling factors

classmethod cat(*boxes_list: List[Boxes]*) → cvpods.structures.boxes.Boxes
Concatenates a list of Boxes into a single Boxes

Parameters **boxes_list** (*list[Boxes]*) –

Returns *Boxes* – the concatenated Boxes

property device

__iter__() → Iterator[torch.Tensor]
Yield a box as a Tensor of shape (4,) at a time.

class cvpods.structures.**BoxMode**
Bases: enum.IntEnum

Enum of different ways to represent a box.

XYXY_ABS
(x0, y0, x1, y1) in absolute floating points coordinates. The coordinates in range [0, width or height].

XYWH_ABS
(x0, y0, w, h) in absolute floating points coordinates.

XYXY_REL
(x0, y0, x1, y1) in range [0, 1]. They are relative to the size of the image.

XYWH_REL
(x0, y0, w, h) in range [0, 1]. They are relative to the size of the image.

XYWHA_ABS
(xc, yc, w, h, a) in absolute floating points coordinates. (xc, yc) is the center of the rotated box, and the angle a is in degrees ccw.

XYXY_ABS = 0

XYWH_ABS = 1

XYXY_REL = 2

XYWH_REL = 3

XYWHA_ABS = 4

static convert(*box: Union[List[float], Tuple[float, ...], torch.Tensor, numpy.ndarray]*, *from_mode: cvpods.structures.boxes.BoxMode*, *to_mode: cvpods.structures.boxes.BoxMode*) → Union[List[float], Tuple[float, ...], torch.Tensor, numpy.ndarray]

Parameters

- **box** – can be a k-tuple, k-list or an Nxk array/tensor, where k = 4 or 5
- **to_mode** (*from_mode,*) –

Returns The converted box of the same type.

cvpods.structures.**pairwise_ioa**(*gt: cvpods.structures.boxes.Boxes*, *boxes: cvpods.structures.boxes.Boxes*, *labels*, *ignore_label=-1*) → torch.Tensor

Given two lists of boxes of size N and M, compute the IoU (intersection over union) between __all__ N x M pairs of boxes. The box order must be (xmin, ymin, xmax, ymax).

Parameters **boxes1,boxes2** (Boxes) – two *Boxes*. Contains N & M boxes, respectively.

Returns *Tensor* – IoU, sized [N,M].

cvpods.structures.**pairwise_iou**(*boxes1: cvpods.structures.boxes.Boxes*, *boxes2: cvpods.structures.boxes.Boxes*) → torch.Tensor

Given two lists of boxes of size N and M, compute the IoU (intersection over union) between __all__ N x M pairs of boxes. The box order must be (xmin, ymin, xmax, ymax).

Parameters **boxes1,boxes2** (Boxes) – two *Boxes*. Contains N & M boxes, respectively.

Returns *Tensor* – IoU, sized [N,M].

class cvpods.structures.**ImageList**(*tensor: torch.Tensor*, *image_sizes: List[Tuple[int, int]]*)
Bases: object

Structure that holds a list of images (of possibly varying sizes) as a single tensor. This works by padding the images to the same size, and storing in a field the original sizes of each image

image_sizes
each tuple is (h, w)

Type list[tuple[int, int]]

__init__(*tensor: torch.Tensor*, *image_sizes: List[Tuple[int, int]]*)

Parameters

- **tensor** (*Tensor*) – of shape (N, H, W) or (N, C_1, ..., C_K, H, W) where K >= 1
- **image_sizes** (*list[tuple[int, int]]*) – Each tuple is (h, w).

__getitem__(*idx: Union[int, slice]*) → torch.Tensor

Access the individual image in its original size.

Returns *Tensor* – an image of shape (H, W) or (C_1, ..., C_K, H, W) where K >= 1

to(**args: Any*, ***kwargs: Any*) → cvpods.structures.image_list.ImageList

property device

static from_tensors(*tensors: Sequence[torch.Tensor]*, *size_divisibility: int = 0*, *pad_ref_long: bool = False*, *pad_value: float = 0.0*) → cvpods.structures.image_list.ImageList

Parameters

- **tensors** – a tuple or list of *torch.Tensors*, each of shape (Hi, Wi) or (C_1, ..., C_K, Hi, Wi) where K >= 1. The Tensors will be padded with *pad_value* so that they will have the same shape.
- **size_divisibility** (*int*) – If *size_divisibility* > *0*, also adds padding to ensure the common height and width is divisible by *size_divisibility*
- **pad_value** (*float*) – value to pad

Returns an *ImageList*.

class cvpods.structures.**Instances**(*image_size: Tuple[int, int]*, ***kwargs: Any*)

Bases: object

This class represents a list of instances in an image. It stores the attributes of instances (e.g., boxes, masks, labels, scores) as "fields". All fields must have the same __len__ which is the number of instances.

All other (non-field) attributes of this class are considered private: they must start with '_' and are not modifiable by a user.

Some basic usage:

1. Set/Get a field: .. code-block:: python

   instances.gt_boxes = Boxes(...) print(instances.pred_masks) # a tensor of shape (N, H, W) print('gt_masks' in instances)
2. len(instances) returns the number of instances
3. Indexing: instances[indices] will apply the indexing on all the fields and returns a new *Instances*. Typically, indices is a integer vector of indices, or a binary mask of length num_instances,

__init__(*image_size: Tuple[int, int]*, ***kwargs: Any*)

Parameters

- **image_size** (*height, width*) – the spatial size of the image.
- **kwargs** – fields to add to this *Instances*.

property image_size

Returns: tuple: height, width

set(*name: str*, *value: Any*) → None

Set the field named *name* to *value*. The length of *value* must be the number of instances, and must agree with other existing fields in this object.

has (*name: str*) → bool

Returns *bool* – whether the field called *name* exists.

remove (*name: str*) → None
Remove the field called *name*.

get (*name: str*) → Any
Returns the field called *name*.

get_fields () → Dict[str, Any]

Returns *dict* – a dict which maps names (str) to data of the fields

Modifying the returned dict will modify this instance.

to (*device: str*) → cvpods.structures.instances.Instances

Returns *Instances* – all fields are called with a *to(device)*, if the field has this method.

__getitem__ (*item: Union[int, slice, torch.BoolTensor]*) → cvpods.structures.instances.Instances

Parameters **item** – an index-like object and will be used to index all the fields.

Returns If *item* is a string, return the data in the corresponding field. Otherwise, returns an *Instances* where all fields are indexed by *item*.

static cat (*instance_lists: List[Instances]*) → cvpods.structures.instances.Instances

Parameters **instance_lists** (*list[Instances]*) –

Returns Instances

class cvpods.structures.**Keypoints** (*keypoints: Union[torch.Tensor, numpy.ndarray, List[List[float]]]*)

Bases: object

Stores keypoint annotation data. GT Instances have a *gt_keypoints* property containing the x,y location and visibility flag of each keypoint. This tensor has shape (N, K, 3) where N is the number of instances and K is the number of keypoints per instance.

The visibility flag follows the COCO format and must be one of three integers: * v=0: not labeled (in which case x=y=0) * v=1: labeled but not visible * v=2: labeled and visible

__init__ (*keypoints: Union[torch.Tensor, numpy.ndarray, List[List[float]]]*)

Parameters

- **keypoints** – A Tensor, numpy array, or list of the x, y, and visibility of each keypoint.
- **shape should be** (*The*) –
- **and K is the number of keypoints per instance.** (*instances,*) –

to (**args: Any*, ***kwargs: Any*) → cvpods.structures.keypoints.Keypoints

property device

to_heatmap (*boxes: torch.Tensor*, *heatmap_size: int*) → torch.Tensor

Parameters **boxes** – Nx4 tensor, the boxes to draw the keypoints to

Returns

***heatmaps* – A tensor of shape (N, K) containing an integer spatial label** in the range [0, heatmap_size**2 - 1] for each keypoint in the input.

valid: A tensor of shape (N, K) containing whether each keypoint is in the roi or not.

__getitem__(*item: Union[int, slice, torch.BoolTensor]*) → cvpods.structures.keypoints.Keypoints
Create a new *Keypoints* by indexing on this *Keypoints*.

The following usage are allowed:

1. *new_kpts = kpts[3]*: return a *Keypoints* which contains only one instance.
2. *new_kpts = kpts[2:10]*: return a slice of key points.
3. *new_kpts = kpts[vector]*, where vector is a torch.ByteTensor with *length = len(kpts)*. Nonzero elements in the vector will be selected.

Note that the returned Keypoints might share storage with this Keypoints, subject to Pytorch's indexing semantics.

cvpods.structures.**heatmaps_to_keypoints**(*maps: torch.Tensor*, *rois: torch.Tensor*) → torch.Tensor
Extract predicted keypoint locations from heatmaps. :param maps: (#ROIs, #keypoints, POOL_H, POOL_W). The predicted heatmap of logits for :type maps: Tensor :param each ROI and each keypoint.: :param rois: (#ROIs, 4). The box of each ROI. :type rois: Tensor

Returns Tensor of shape (#ROIs, #keypoints, 4) with the last dimension corresponding to (x, y, logit, score) for each keypoint.

When converting discrete pixel indices in an NxN image to a continuous keypoint coordinate, we maintain consistency with `Keypoints.to_heatmap()` by using the conversion from Heckbert 1990: c = d + 0.5, where d is a discrete coordinate and c is a continuous coordinate.

class cvpods.structures.**BitMasks**(*tensor: Union[torch.Tensor, numpy.ndarray]*)
Bases: object

This class stores the segmentation masks for all objects in one image, in the form of bitmaps.

tensor
bool Tensor of N,H,W, representing N instances in the image.

__init__(*tensor: Union[torch.Tensor, numpy.ndarray]*)

Parameters **tensor** – bool Tensor of N,H,W, representing N instances in the image.

to(*device: str*) → cvpods.structures.masks.BitMasks

property device

__getitem__(*item: Union[int, slice, torch.BoolTensor]*) → cvpods.structures.masks.BitMasks

Returns *BitMasks* – Create a new `BitMasks` by indexing.

The following usage are allowed:

1. *new_masks = masks[3]*: return a *BitMasks* which contains only one mask.
2. *new_masks = masks[2:10]*: return a slice of masks.
3. *new_masks = masks[vector]*, where vector is a torch.BoolTensor with *length = len(masks)*. Nonzero elements in the vector will be selected.

Note that the returned object might share storage with this object, subject to Pytorch's indexing semantics.

nonempty() → torch.Tensor
Find masks that are non-empty.

Returns

Tensor –

a BoolTensor which represents whether each mask is empty (False) or non-empty (True).

static from_polygon_masks(*polygon_masks: Union[PolygonMasks, List[List[numpy.ndarray]]]*, *height: int*, *width: int*) → cvpods.structures.masks.BitMasks

Parameters

- **polygon_masks** (*list[list[ndarray]] or PolygonMasks*) –
- **width** (*height,*) –

crop_and_resize(*boxes: torch.Tensor*, *mask_size: int*) → torch.Tensor

Crop each bitmask by the given box, and resize results to (mask_size, mask_size). This can be used to prepare training targets for Mask R-CNN. It has less reconstruction error compared to rasterization with polygons. However we observe no difference in accuracy, but BitMasks requires more memory to store all the masks.

Parameters

- **boxes** (*Tensor*) – Nx4 tensor storing the boxes for each mask
- **mask_size** (*int*) – the size of the rasterized mask.

Returns *Tensor* – A bool tensor of shape (N, mask_size, mask_size), where N is the number of predicted boxes for this image.

get_bounding_boxes() → None

Returns *Boxes* – tight bounding boxes around bit masks.

static cat(*bitmasks_list: List[BitMasks]*) → cvpods.structures.masks.BitMasks

Concatenates a list of BitMasks into a single BitMasks :param bitmasks_list: :type bitmasks_list: list[BitMasks]

Returns *BitMasks* – the concatenated BitMasks

class cvpods.structures.**PolygonMasks**(*polygons: List[List[Union[torch.Tensor, numpy.ndarray]]]*)

Bases: object

This class stores the segmentation masks for all objects in one image, in the form of polygons.

polygons

list[list[ndarray]]. Each ndarray is a float64 vector representing a polygon.

__init__(*polygons: List[List[Union[torch.Tensor, numpy.ndarray]]]*)

Parameters **polygons** (*list[list[Tensor[float]]]*) – The first level of the list correspond to individual instances, the second level to all the polygons that compose the instance, and the third level to the polygon coordinates. The third level Tensor should have the format of torch.Tensor([x0, y0, x1, y1, . . . , xn, yn]) (n >= 3).

to(*device: str*) → cvpods.structures.masks.PolygonMasks

property device

get_bounding_boxes() → cvpods.structures.boxes.Boxes

Returns *Boxes* – tight bounding boxes around polygon masks.

nonempty() → torch.Tensor

Find masks that are non-empty.

Returns *Tensor* – a BoolTensor which represents whether each mask is empty (False) or not (True).

__getitem__(*item: Union[int, slice, List[int], torch.BoolTensor]*) → cvpods.structures.masks.PolygonMasks

Support indexing over the instances and return a *PolygonMasks* object. *item* can be:

1. An integer. It will return an object with only one instance.
2. A slice. It will return an object with the selected instances.
3. A list[int]. It will return an object with the selected instances, correpsonding to the indices in the list.
4. A vector mask of type BoolTensor, whose length is num_instances. It will return an object with the instances whose mask is nonzero.

__iter__() → Iterator[List[torch.Tensor]]

Yields *list[ndarray]* – the polygons for one instance. Each Tensor is a float64 vector representing a polygon.

crop_and_resize(*boxes: torch.Tensor*, *mask_size: int*) → torch.Tensor

Crop each mask by the given box, and resize results to (mask_size, mask_size). This can be used to prepare training targets for Mask R-CNN.

Parameters

- **boxes** (*Tensor*) – Nx4 tensor storing the boxes for each mask
- **mask_size** (*int*) – the size of the rasterized mask.

Returns *Tensor* – A bool tensor of shape (N, mask_size, mask_size), where N is the number of predicted boxes for this image.

area()

Computes area of the mask. Only works with Polygons, using the shoelace formula: https://stackoverflow.com/questions/24467972/calculate-area-of-polygon-given-x-y-coordinates :returns: *Tensor* – a vector, area for each instance

static cat(*polymasks_list: List[PolygonMasks]*) → cvpods.structures.masks.PolygonMasks

Concatenates a list of PolygonMasks into a single PolygonMasks :param polymasks_list: :type polymasks_list: list[PolygonMasks]

Returns *PolygonMasks* – the concatenated PolygonMasks

cvpods.structures.**polygons_to_bitmask**(*polygons: List[numpy.ndarray]*, *height: int*, *width: int*) → numpy.ndarray

Parameters

- **polygons** (*list[ndarray]*) – each array has shape (Nx2,)
- **width** (*height,*) –

Returns *ndarray* – a bool mask of shape (height, width)

cvpods.structures.**rasterize_polygons_within_box**(*polygons: List[numpy.ndarray]*, *box: numpy.ndarray*, *mask_size: int*) → torch.Tensor

Rasterize the polygons into a mask image and crop the mask content in the given box. The cropped mask is resized to (mask_size, mask_size).

This function is used when generating training targets for mask head in Mask R-CNN. Given original ground-truth masks for an image, new ground-truth mask training targets in the size of *mask_size x mask_size* must be provided for each predicted box. This function will be called to produce such targets.

Parameters

- **polygons** (*list[ndarray[float]]*) – a list of polygons, which represents an instance.
- **box** – 4-element numpy array
- **mask_size** (*int*) –

Returns *Tensor* – BoolTensor of shape (mask_size, mask_size)

class cvpods.structures.**RotatedBoxes**(*tensor: torch.Tensor*)

Bases: cvpods.structures.boxes.Boxes

This structure stores a list of rotated boxes as a Nx5 torch.Tensor. It supports some common methods about boxes (*area*, *clip*, *nonempty*, etc), and also behaves like a Tensor (support indexing, *to(device)*, *.device*, and iteration over all boxes)

__init__(*tensor: torch.Tensor*)

Parameters **tensor** (*Tensor[float]*) – a Nx5 matrix. Each row is (x_center, y_center, width, height, angle), in which angle is represented in degrees. While there's no strict range restriction for it, the recommended principal range is between [-180, 180) degrees.

Assume we have a horizontal box B = (x_center, y_center, width, height), where width is along the x-axis and height is along the y-axis. The rotated box B_rot (x_center, y_center, width, height, angle) can be seen as:

1. When angle == 0: B_rot == B
2. When angle > 0: B_rot is obtained by rotating B w.r.t its center by $|angle|$ degrees CCW;
3. When angle < 0: B_rot is obtained by rotating B w.r.t its center by $|angle|$ degrees CW.

Mathematically, since the right-handed coordinate system for image space is (y, x), where y is top->down and x is left->right, the 4 vertices of the rotated rectangle (yr_i, xr_i) (i = 1, 2, 3, 4) can be obtained from the vertices of the horizontal rectangle (y_i, x_i) (i = 1, 2, 3, 4) in the following way ($\theta = angle * \pi/180$ is the angle in radians, (y_c, x_c) is the center of the rectangle):

$$yr_i = \cos(\theta)(y_i - y_c) - \sin(\theta)(x_i - x_c) + y_c,$$
$$xr_i = \sin(\theta)(y_i - y_c) + \cos(\theta)(x_i - x_c) + x_c,$$

which is the standard rigid-body rotation transformation.

Intuitively, the angle is (1) the rotation angle from y-axis in image space to the height vector (top->down in the box's local coordinate system) of the box in CCW, and (2) the rotation angle from x-axis in image space to the width vector (left->right in the box's local coordinate system) of the box in CCW.

More intuitively, consider the following horizontal box ABCD represented in (x1, y1, x2, y2): (3, 2, 7, 4), covering the [3, 7] x [2, 4] region of the continuous coordinate system which looks like this:

```
O--------> x
|
|  A---B
|  |   |
|  D---C
|
v y
```

Note that each capital letter represents one 0-dimensional geometric point instead of a 'square pixel' here.

In the example above, using (x, y) to represent a point we have:

$$O = (0, 0), A = (3, 2), B = (7, 2), C = (7, 4), D = (3, 4)$$

We name vector AB = vector DC as the width vector in box's local coordinate system, and vector AD = vector BC as the height vector in box's local coordinate system. Initially, when angle = 0 degree, they're aligned with the positive directions of x-axis and y-axis in the image space, respectively.

For better illustration, we denote the center of the box as E,

```
O--------> x
|
|   A---B
|   | E |
|   D---C
|
v y
```

where the center E = ((3+7)/2, (2+4)/2) = (5, 3).

Also,

$$width = |AB| = |CD| = 7 - 3 = 4, height = |AD| = |BC| = 4 - 2 = 2.$$

Therefore, the corresponding representation for the same shape in rotated box in (x_center, y_center, width, height, angle) format is:

(5, 3, 4, 2, 0),

Now, let's consider (5, 3, 4, 2, 90), which is rotated by 90 degrees CCW (counter-clockwise) by definition. It looks like this:

```
O--------> x
|    B-C
|    | |
|    |E|
|    | |
|    A-D
v y
```

The center E is still located at the same point (5, 3), while the vertices ABCD are rotated by 90 degrees CCW with regard to E: A = (4, 5), B = (4, 1), C = (6, 1), D = (6, 5)

Here, 90 degrees can be seen as the CCW angle to rotate from y-axis to vector AD or vector BC (the top->down height vector in box's local coordinate system), or the CCW angle to rotate from x-axis to vector AB or vector DC (the left->right width vector in box's local coordinate system).

$$width = |AB| = |CD| = 5 - 1 = 4, height = |AD| = |BC| = 6 - 4 = 2.$$

Next, how about (5, 3, 4, 2, -90), which is rotated by 90 degrees CW (clockwise) by definition? It looks like this:

```
O--------> x
|    D-A
|    | |
|    |E|
|    | |
|    C-B
v y
```

The center E is still located at the same point (5, 3), while the vertices ABCD are rotated by 90 degrees CW with regard to E: A = (6, 1), B = (6, 5), C = (4, 5), D = (4, 1)

$$width = |AB| = |CD| = 5 - 1 = 4, height = |AD| = |BC| = 6 - 4 = 2.$$

This covers exactly the same region as (5, 3, 4, 2, 90) does, and their IoU will be 1. However, these two will generate different RoI Pooling results and should not be treated as an identical box.

On the other hand, it's easy to see that (X, Y, W, H, A) is identical to (X, Y, W, H, A+360N), for any integer N. For example (5, 3, 4, 2, 270) would be identical to (5, 3, 4, 2, -90), because rotating the shape 270 degrees CCW is equivalent to rotating the same shape 90 degrees CW.

We could rotate further to get (5, 3, 4, 2, 180), or (5, 3, 4, 2, -180):

```
O--------> x
|
|  C---D
|  | E |
|  B---A
|
v y
```

$$A = (7, 4), B = (3, 4), C = (3, 2), D = (7, 2),$$
$$width = |AB| = |CD| = 7 - 3 = 4, height = |AD| = |BC| = 4 - 2 = 2.$$

Finally, this is a very inaccurate (heavily quantized) illustration of how (5, 3, 4, 2, 60) looks like in case anyone wonders:

```
O--------> x
|     B            |    /  C
|   /E /
|  A  /
|   `D
v y
```

It's still a rectangle with center of (5, 3), width of 4 and height of 2, but its angle (and thus orientation) is somewhere between (5, 3, 4, 2, 0) and (5, 3, 4, 2, 90).

clone() → cvpods.structures.rotated_boxes.RotatedBoxes
Clone the RotatedBoxes.

Returns RotatedBoxes

to(*device: str*) → cvpods.structures.rotated_boxes.RotatedBoxes

area() → torch.Tensor
Computes the area of all the boxes.

Returns *torch.Tensor* – a vector with areas of each box.

normalize_angles() → None
Restrict angles to the range of [-180, 180) degrees

clip(*box_size: Union[List[int], Tuple[int, int]]*, *clip_angle_threshold: float = 1.0*) → None
Clip (in place) the boxes by limiting x coordinates to the range [0, width] and y coordinates to the range [0, height].

For RRPN: Only clip boxes that are almost horizontal with a tolerance of clip_angle_threshold to maintain backward compatibility.

Rotated boxes beyond this threshold are not clipped for two reasons:

1. There are potentially multiple ways to clip a rotated box to make it fit within the image.
2. It's tricky to make the entire rectangular box fit within the image and still be able to not leave out pixels of interest.

Therefore we rely on ops like RoIAlignRotated to safely handle this.

Parameters

- **box_size** (*height, width*) – The clipping box's size.
- **clip_angle_threshold** – Iff. abs(normalized(angle)) <= clip_angle_threshold (in degrees), we do the clipping as horizontal boxes.

nonempty (*threshold: int = 0*) → torch.Tensor

Find boxes that are non-empty. A box is considered empty, if either of its side is no larger than threshold.

Returns *Tensor* – a binary vector which represents whether each box is empty (False) or non-empty (True).

__getitem__ (*item: Union[int, slice, torch.BoolTensor]*) → cvpods.structures.rotated_boxes.RotatedBoxes

Returns *RotatedBoxes* – Create a new *RotatedBoxes* by indexing.

The following usage are allowed:

1. *new_boxes = boxes[3]*: return a *RotatedBoxes* which contains only one box.
2. *new_boxes = boxes[2:10]*: return a slice of boxes.
3. *new_boxes = boxes[vector]*, where vector is a torch.ByteTensor with *length = len(boxes)*. Nonzero elements in the vector will be selected.

Note that the returned RotatedBoxes might share storage with this RotatedBoxes, subject to Pytorch's indexing semantics.

inside_box (*box_size: Union[List[int], Tuple[int, int]], boundary_threshold: int = 0*) → torch.Tensor

Parameters

- **box_size** (*height, width*) – Size of the reference box covering [0, width] x [0, height]
- **boundary_threshold** (*int*) – Boxes that extend beyond the reference box boundary by more than boundary_threshold are considered "outside".

For RRPN, it might not be necessary to call this function since it's common for rotated box to extend to outside of the image boundaries (the clip function only clips the near-horizontal boxes)

Returns a binary vector, indicating whether each box is inside the reference box.

get_centers () → torch.Tensor

Returns The box centers in a Nx2 array of (x, y).

scale (*scale_x: float, scale_y: float*) → None

Scale the rotated box with horizontal and vertical scaling factors Note: when scale_factor_x != scale_factor_y, the rotated box does not preserve the rectangular shape when the angle is not a multiple of 90 degrees under resize transformation. Instead, the shape is a parallelogram (that has skew) Here we make an approximation by fitting a rotated rectangle to the parallelogram.

classmethod cat (*boxes_list: List[RotatedBoxes]*) → cvpods.structures.rotated_boxes.RotatedBoxes

Concatenates a list of RotatedBoxes into a single RotatedBoxes

Parameters **boxes_list** (*list[RotatedBoxes]*) –

Returns *RotatedBoxes* – the concatenated RotatedBoxes

property device

__iter__() → Iterator[torch.Tensor]
Yield a box as a Tensor of shape (5,) at a time.

cvpods.structures.**pairwise_iou_rotated**(*boxes1: cvpods.structures.rotated_boxes.RotatedBoxes*, *boxes2: cvpods.structures.rotated_boxes.RotatedBoxes*) → None
Given two lists of rotated boxes of size N and M, compute the IoU (intersection over union) between __all__ N x M pairs of boxes. The box order must be (x_center, y_center, width, height, angle).

Parameters **boxes2** (*boxes1,*) – two *RotatedBoxes*. Contains N & M rotated boxes, respectively.

Returns *Tensor* – IoU, sized [N,M].

1.10 cvpods.utils package

1.10.1 cvpods.utils.benchmark.benchmark module

cvpods.utils.benchmark.benchmark.**timeit**(*num_iters: int = - 1*, *warmup_iters: int = 0*)
This is intened to be used as a decorator to time any function.

Parameters

- **num_iters** (*int*) – number of iterations used to compute the average time (sec) required to run the function. If negative, the number of iterations is determined dynamically by running the function a few times to make sure the estimate is stable.
- **warmup_iters** (*int*) – number of iterations used to warm up the function. This is useful for functions that exhibit poor performance during the first few times they run (due to caches, autotuning, etc).

Returns

Dict[str, float] –

dictionary of the aggregated timing estimates.

"iterations": number of iterations used to compute the estimated time.

"mean": averate time (sec) used to run the function. "median": median time (sec) used to run the function. "min": minimal time (sec) used to run the function. "max": maximal time (sec) used to run the function. "stddev": standard deviation of the time (sec) used to run the

function.

cvpods.utils.benchmark.benchmark.**benchmark**(*func*, *bm_name: str*, *kwargs_list: List[Dict]*, ***, *num_iters: int = - 1*, *warmup_iters: int = 0*) → None
Benchmark the input function and print out the results.

Parameters

- **func** (*callable*) – a closure that returns a function for benchmarking, where initialization can be done before the function to benchmark.
- **bm_name** (*str*) – name of the benchmark to print out, e.g. "BM_UPDATE".
- **kwargs_list** (*list*) – a list of argument dict to pass to the function. The intput function will be timed separately for each argument dict.
- **num_iters** (*int*) – number of iterations to run. Defaults to run until 0.5s.
- **warmup_iters** (*int*) – number of iterations used to warm up the function.

Outputs: For each argument dict, print out the time (in microseconds) required to run the function along with the number of iterations used to get the timing estimate. Example output:

BM_UPDATE_100 820 914 610 BM_UPDATE_1000 7655 8709 66 BM_UPDATE_10000 78062 81748 7 ——————————————————————————————-

1.10.2 cvpods.utils.benchmark.timer module

class cvpods.utils.benchmark.timer.**Timer**
Bases: object

A timer which computes the time elapsed since the start/reset of the timer.

reset()
Reset the timer.

pause()
Pause the timer.

is_paused() → bool

Returns *bool* – whether the timer is currently paused

resume()
Resume the timer.

seconds() → float

Returns

(float) –

the total number of seconds since the start/reset of the timer, excluding the time when the timer is paused.

1.10.3 cvpods.utils.distributed.comm module

This file contains primitives for multi-gpu communication. This is useful when doing distributed training.

cvpods.utils.distributed.comm.**get_host_ip**()

cvpods.utils.distributed.comm.**get_world_size**() → int

cvpods.utils.distributed.comm.**get_rank**() → int

cvpods.utils.distributed.comm.**get_local_rank**() → int
Returns The rank of the current process within the local (per-machine) process group.

cvpods.utils.distributed.comm.**get_local_size**() → int
Returns The size of the per-machine process group, i.e. the number of processes per machine.

cvpods.utils.distributed.comm.**is_main_process**() → bool

cvpods.utils.distributed.comm.**synchronize**()
Helper function to synchronize (barrier) among all processes when using distributed training

cvpods.utils.distributed.comm.**all_gather**(*data*, *group=None*)
Run all_gather on arbitrary picklable data (not necessarily tensors).
Parameters

- **data** – any picklable object

- **group** – a torch process group. By default, will use a group which contains all ranks on gloo backend.

Returns *list[data]* – list of data gathered from each rank

cvpods.utils.distributed.comm.**gather**(*data*, *dst=0*, *group=None*)

Run gather on arbitrary picklable data (not necessarily tensors).

Parameters

- **data** – any picklable object
- **dst** (*int*) – destination rank
- **group** – a torch process group. By default, will use a group which contains all ranks on gloo backend.

Returns

list[data] –

on dst, a list of data gathered from each rank. Otherwise, an empty list.

cvpods.utils.distributed.comm.**all_reduce**(*data*, *op='sum'*)

cvpods.utils.distributed.comm.**shared_random_seed**()

Returns

int –

a random number that is the same across all workers. If workers need a shared RNG, they can use this shared seed to create one.

All workers must call this function, otherwise it will deadlock.

cvpods.utils.distributed.comm.**reduce_dict**(*input_dict*, *average=True*)

Reduce the values in the dictionary from all processes so that process with rank 0 has the reduced results.

Parameters

- **input_dict** (*dict*) – inputs to be reduced. All the values must be scalar CUDA Tensor.
- **average** (*bool*) – whether to do average or sum

Returns a dict with the same keys as input_dict, after reduction.

1.10.4 cvpods.utils.env.collect_env module

cvpods.utils.env.collect_env.**collect_env_info**()

1.10.5 cvpods.utils.env.env module

cvpods.utils.env.env.**seed_all_rng**(*seed=None*)

Set the random seed for the RNG in torch, numpy and python.

Parameters **seed** (*int*) – if None, will use a strong random seed.

Returns *seed (int)* – used seed value.

cvpods.utils.env.env.**setup_environment**()

Perform environment setup work. The default setup is a no-op, but this function allows the user to specify a Python source file or a module in the $cvpods_ENV_MODULE environment variable, that performs custom setup work that may be necessary to their computing environment.

cvpods.utils.env.env.**setup_custom_environment**(*custom_module*)
Load custom environment setup by importing a Python source file or a module, and run the setup function.

1.10.6 cvpods.utils.file.download module

cvpods.utils.file.download.**download**(*url: str*, *dir: str*, ***, *filename: Optional[str] = None*, *progress: bool = True*) → str
Download a file from a given URL to a directory. If file exists, will not overwrite the existing file.

Parameters

- **url** (*str*) –
- **dir** (*str*) – the directory to download the file
- **filename** (*str or None*) – the basename to save the file. Will use the name in the URL if not given.
- **progress** (*bool*) – whether to use tqdm to draw a progress bar.

Returns *str* – the path to the downloaded file or the existing one.

1.10.7 cvpods.utils.file.file_io module

cvpods.utils.file.file_io.**get_cache_dir**(*cache_dir: Optional[str] = None*) → str
Returns a default directory to cache static files (usually downloaded from Internet), if None is provided.
Parameters

- **cache_dir** (*None or str*) – if not None, will be returned as is. If None, returns the default cache directory as:
- **$CVPODS_CACHE, if set** (*1*) –
- **otherwise ~/.torch/cvpods_cache** (*2*) –

cvpods.utils.file.file_io.**file_lock**(*path: str*)
A file lock. Once entered, it is guaranteed that no one else holds the same lock. Others trying to enter the lock will block for 30 minutes and raise an exception.

This is useful to make sure workers don't cache files to the same location.
Parameters **path** (*str*) – a path to be locked. This function will create a lock named *path* + *".lock"*
Examples:

```
>>> filename = "/path/to/file"
>>> with file_lock(filename):
        if not os.path.isfile(filename):
            do_create_file()
```

class cvpods.utils.file.file_io.**PathHandler**
Bases: object

PathHandler is a base class that defines common I/O functionality for a URI protocol. It routes I/O for a generic URI which may look like "protocol://*" or a canonical filepath "/foo/bar/baz".

class cvpods.utils.file.file_io.**PathManager**
Bases: object

A class for users to open generic paths or translate generic paths to file names.

static open(*path: str*, *mode: str = 'r'*) → IO[Any]
Open a stream to a URI, similar to the built-in *open*.

Parameters **path** (*str*) – A URI supported by this PathHandler

Returns *file* – a file-like object.

static copy(*src_path: str*, *dst_path: str*, *overwrite: bool = False*) → bool
Copies a source path to a destination path.

Parameters

- **src_path** (*str*) – A URI supported by this PathHandler
- **dst_path** (*str*) – A URI supported by this PathHandler
- **overwrite** (*bool*) – Bool flag for forcing overwrite of existing file

Returns *status (bool)* – True on success

static get_local_path(*path: str*) → str
Get a filepath which is compatible with native Python I/O such as *open* and *os.path*.

If URI points to a remote resource, this function may download and cache the resource to local disk.

Parameters **path** (*str*) – A URI supported by this PathHandler

Returns *local_path (str)* – a file path which exists on the local file system

static exists(*path: str*) → bool
Checks if there is a resource at the given URI.

Parameters **path** (*str*) – A URI supported by this PathHandler

Returns *bool* – true if the path exists

static isfile(*path: str*) → bool
Checks if there the resource at the given URI is a file.

Parameters **path** (*str*) – A URI supported by this PathHandler

Returns *bool* – true if the path is a file

static isdir(*path: str*) → bool
Checks if the resource at the given URI is a directory.

Parameters **path** (*str*) – A URI supported by this PathHandler

Returns *bool* – true if the path is a directory

static ls(*path: str*) → List[str]
List the contents of the directory at the provided URI.

Parameters **path** (*str*) – A URI supported by this PathHandler

Returns *List[str]* – list of contents in given path

static mkdirs(*path: str*) → None
Recursive directory creation function. Like mkdir(), but makes all intermediate-level directories needed to contain the leaf directory. Similar to the native *os.makedirs*.

Parameters **path** (*str*) – A URI supported by this PathHandler

static rm(*path: str*) → None
Remove the file (not directory) at the provided URI.

Parameters **path** (*str*) – A URI supported by this PathHandler

static stat(*path: str*)
get status of the file at the provided URI.

Parameters **path** (*str*) – A URI supported by this PathHandler

static upload(*local: str*, *remote: str*)
Upload the local file (not directory) to the specified remote URI.

Parameters

- **local** (*str*) – path of the local file to be uploaded.
- **remote** (*str*) – the remote s3uri.

static register_handler(*handler:* cvpods.utils.file.file_io.PathHandler) → None
Register a path handler associated with *handler._get_supported_prefixes* URI prefixes.

Parameters **handler** (PathHandler) –

1.10.8 cvpods.utils.imports module

cvpods.utils.imports.**dynamic_import**(*config_name*, *config_path*)
Dynamic import a project.

Parameters

- **config_name** (*str*) – module name
- **config_path** (*str*) – the dir that contains the .py with this module.

Examples::

```
>>> root = "/path/to/your/retinanet/"
>>> project = root + "retinanet.res50.fpn.coco.800size.1x.mrcnn_sigmoid"
>>> cfg = dynamic_import("config", project).config
>>> net = dynamic_import("net", project)
```

1.10.9 cvpods.utils.memory module

cvpods.utils.memory.**retry_if_cuda_oom**(*func*)
Makes a function retry itself after encountering pytorch's CUDA OOM error. It will first retry after calling *torch.cuda.empty_cache()*.

If that still fails, it will then retry by trying to convert inputs to CPUs. In this case, it expects the function to dispatch to CPU implementation. The return values may become CPU tensors as well and it's user's responsibility to convert it back to CUDA tensor if needed.

Parameters **func** – a stateless callable that takes tensor-like objects as arguments

Returns a callable which retries *func* if OOM is encountered.

Examples:

```
output = retry_if_cuda_oom(some_torch_function)(input1, input2)
# output may be on CPU even if inputs are on GPU
```

Note:

1. When converting inputs to CPU, it will only look at each argument and check if it has *.device* and *.to* for conversion. Nested structures of tensors are not supported.
2. Since the function might be called more than once, it has to be stateless.

1.10.10 cvpods.utils.visualizer.colormap module

An awesome colormap for really neat visualizations. Copied from Detectron, and removed gray colors.

cvpods.utils.visualizer.colormap.**colormap**(*rgb=False*, *maximum=255*)

Parameters

- **rgb** (*bool*) – whether to return RGB colors or BGR colors.
- **maximum** (*int*) – either 255 or 1

Returns *ndarray* – a float32 array of Nx3 colors, in range [0, 255] or [0, 1]

cvpods.utils.visualizer.colormap.**random_color**(*rgb=False*, *maximum=255*)

Parameters

- **rgb** (*bool*) – whether to return RGB colors or BGR colors.
- **maximum** (*int*) – either 255 or 1

Returns *ndarray* – a vector of 3 numbers

1.10.11 cvpods.utils.visualizer.video_visualizer module

class cvpods.utils.visualizer.video_visualizer.**VideoVisualizer**(*metadata*, *instance_mode=<ColorMode.IMAGE: 0>*)

Bases: object

__init__(*metadata*, *instance_mode=<ColorMode.IMAGE: 0>*)

Parameters **metadata** (*MetadataCatalog*) – image metadata.

draw_instance_predictions(*frame*, *predictions*)

Draw instance-level prediction results on an image.

Parameters

- **frame** (*ndarray*) – an RGB image of shape (H, W, C), in the range [0, 255].
- **predictions** (*Instances*) – the output of an instance detection/segmentation model. Following fields will be used to draw: "pred_boxes", "pred_classes", "scores", "pred_masks" (or "pred_masks_rle").

Returns *output (VisImage)* – image object with visualizations.

draw_sem_seg(*frame*, *sem_seg*, *area_threshold=None*)

Parameters

- **sem_seg** (*ndarray or Tensor*) – semantic segmentation of shape (H, W), each value is the integer label.
- **area_threshold** (*Optional[int]*) – only draw segmentations larger than the threshold

draw_panoptic_seg_predictions(*frame*, *panoptic_seg*, *segments_info*, *area_threshold=None*, *alpha=0.5*)

1.10.12 cvpods.utils.visualizer.visualizer module

class cvpods.utils.visualizer.visualizer.**ColorMode**
Bases: enum.Enum

Enum of different color modes to use for instance visualizations.

IMAGE
Picks a random color for every instance and overlay segmentations with low opacity.

SEGMENTATION
Let instances of the same category have similar colors, and overlay them with high opacity. This provides more attention on the quality of segmentation.

IMAGE_BW
same as IMAGE, but convert all areas without masks to gray-scale. Only available for drawing per-instance mask predictions.

IMAGE = 0

SEGMENTATION = 1

IMAGE_BW = 2

class cvpods.utils.visualizer.visualizer.**VisImage**(*img*, *scale=1.0*)
Bases: object

__init__(*img*, *scale=1.0*)

Parameters

- **img** (*ndarray*) – an RGB image of shape (H, W, 3).
- **scale** (*float*) – scale the input image

save(*filepath*)

Parameters **filepath** (*str*) – a string that contains the absolute path, including the file name, where the visualized image will be saved.

get_image()

Returns

ndarray –

the visualized image of shape (H, W, 3) (RGB) in uint8 type. The shape is scaled w.r.t the input image using the given *scale* argument.

class cvpods.utils.visualizer.visualizer.**Visualizer**(*img_rgb*, *metadata*, *scale=1.0*, *instance_mode=<ColorMode.IMAGE: 0>*)
Bases: object

__init__(*img_rgb*, *metadata*, *scale=1.0*, *instance_mode=<ColorMode.IMAGE: 0>*)

Parameters

- **img_rgb** – a numpy array of shape (H, W, C), where H and W correspond to the height and width of the image respectively. C is the number of color channels. The image is required to be in RGB format since that is a requirement of the Matplotlib library. The image is also expected to be in the range [0, 255].
- **metadata** (*MetadataCatalog*) – image metadata.

draw_instance_predictions(*predictions*)
Draw instance-level prediction results on an image.

Parameters **predictions** (Instances) – the output of an instance detection/segmentation model. Following fields will be used to draw: "pred_boxes", "pred_classes", "scores", "pred_masks" (or "pred_masks_rle").

Returns *output (VisImage)* – image object with visualizations.

draw_sem_seg(*sem_seg*, *area_threshold=None*, *alpha=0.8*)
Draw semantic segmentation predictions/labels.

Parameters

- **sem_seg** (*Tensor or ndarray*) – the segmentation of shape (H, W).
- **area_threshold** (*int*) – segments with less than *area_threshold* are not drawn.
- **alpha** (*float*) – the larger it is, the more opaque the segmentations are.

Returns *output (VisImage)* – image object with visualizations.

draw_panoptic_seg_predictions(*panoptic_seg*, *segments_info*, *area_threshold=None*, *alpha=0.7*)
Draw panoptic prediction results on an image.

Parameters

- **panoptic_seg** (*Tensor*) – of shape (height, width) where the values are ids for each segment.
- **segments_info** (*list[dict]*) – Describe each segment in *panoptic_seg*. Each dict contains keys "id", "category_id", "isthing".
- **area_threshold** (*int*) – stuff segments with less than *area_threshold* are not drawn.

Returns *output (VisImage)* – image object with visualizations.

draw_dataset_dict(*dic*)
Draw annotations/segmentaions in cvpods Dataset format.

Parameters **dic** (*dict*) – annotation/segmentation data of one image, in cvpods Dataset format.

Returns *output (VisImage)* – image object with visualizations.

overlay_instances(*, *boxes=None*, *labels=None*, *masks=None*, *keypoints=None*, *assigned_colors=None*, *alpha=0.5*)

Parameters

- **boxes** (Boxes, RotatedBoxes *or ndarray*) – either a Boxes, or an Nx4 numpy array of XYXY_ABS format for the N objects in a single image, or a RotatedBoxes, or an Nx5 numpy array of (x_center, y_center, width, height, angle_degrees) format for the N objects in a single image,
- **labels** (*list[str]*) – the text to be displayed for each instance.
- **masks** (*masks-like object*) – Supported types are:
  - *structures.masks.PolygonMasks*, *structures.masks.BitMasks*.
  - **list[list[ndarray]]: contains the segmentation masks for all objects in one image.** The first level of the list corresponds to individual instances. The second

level to all the polygon that compose the instance, and the third level to the polygon coordinates. The third level should have the format of [x0, y0, x1, y1, . . . , xn, yn] (n >= 3).

- list[ndarray]: each ndarray is a binary mask of shape (H, W).
- list[dict]: each dict is a COCO-style RLE.

- **keypoints** (*Keypoint or array like*) – an array-like object of shape (N, K, 3), where the N is the number of instances and K is the number of keypoints. The last dimension corresponds to (x, y, visibility or score).
- **assigned_colors** (*list[matplotlib.colors]*) – a list of colors, where each color corresponds to each mask or box in the image. Refer to 'matplotlib.colors' for full list of formats that the colors are accepted in.

Returns *output (VisImage)* – image object with visualizations.

overlay_rotated_instances(*boxes=None*, *labels=None*, *assigned_colors=None*)

Parameters

- **boxes** (*ndarray*) – an Nx5 numpy array of (x_center, y_center, width, height, angle_degrees) format for the N objects in a single image.
- **labels** (*list[str]*) – the text to be displayed for each instance.
- **assigned_colors** (*list[matplotlib.colors]*) – a list of colors, where each color corresponds to each mask or box in the image. Refer to 'matplotlib.colors' for full list of formats that the colors are accepted in.

Returns *output (VisImage)* – image object with visualizations.

draw_and_connect_keypoints(*keypoints*)

Draws keypoints of an instance and follows the rules for keypoint connections to draw lines between appropriate keypoints. This follows color heuristics for line color.

Parameters **keypoints** (*Tensor*) – a tensor of shape (K, 3), where K is the number of keypoints and the last dimension corresponds to (x, y, probability).

Returns *output (VisImage)* – image object with visualizations.

draw_text(*text*, *position*, ***, *font_size=None*, *color='g'*, *horizontal_alignment='center'*, *rotation=0*)

Parameters

- **text** (*str*) – class label
- **position** (*tuple*) – a tuple of the x and y coordinates to place text on image.
- **font_size** (*int, optional*) – font of the text. If not provided, a font size proportional to the image width is calculated and used.
- **color** – color of the text. Refer to *matplotlib.colors* for full list of formats that are accepted.
- **horizontal_alignment** (*str*) – see *matplotlib.text.Text*
- **rotation** – rotation angle in degrees CCW

Returns *output (VisImage)* – image object with text drawn.

draw_box(*box_coord*, *alpha=0.5*, *edge_color='g'*, *line_style='-'*)

Parameters

- **box_coord** (*tuple*) – a tuple containing x0, y0, x1, y1 coordinates, where x0 and y0 are the coordinates of the image's top left corner. x1 and y1 are the coordinates of the image's bottom right corner.
- **alpha** (*float*) – blending efficient. Smaller values lead to more transparent masks.
- **edge_color** – color of the outline of the box. Refer to *matplotlib.colors* for full list of formats that are accepted.
- **line_style** (*string*) – the string to use to create the outline of the boxes.

Returns *output (VisImage)* – image object with box drawn.

draw_rotated_box_with_label(*rotated_box*, *alpha=0.5*, *edge_color='g'*, *line_style='-'*, *label=None*)

Parameters

- **rotated_box** (*tuple*) – a tuple containing (cnt_x, cnt_y, w, h, angle), where cnt_x and cnt_y are the center coordinates of the box. w and h are the width and height of the box. angle represents how many degrees the box is rotated CCW with regard to the 0-degree box.
- **alpha** (*float*) – blending efficient. Smaller values lead to more transparent boxes.
- **edge_color** – color of the outline of the box. Refer to *matplotlib.colors* for full list of formats that are accepted.
- **line_style** (*string*) – the string to use to create the outline of the boxes.
- **label** (*string*) – label for rotated box. It will not be rendered when set to None.

Returns *output (VisImage)* – image object with box drawn.

draw_circle(*circle_coord*, *color*, *radius=3*)

Parameters

- **circle_coord** (*list(int) or tuple(int)*) – contains the x and y coordinates of the center of the circle.
- **color** – color of the polygon. Refer to *matplotlib.colors* for a full list of formats that are accepted.
- **radius** (*int*) – radius of the circle.

Returns *output (VisImage)* – image object with box drawn.

draw_line(*x_data*, *y_data*, *color*, *linestyle='-'*, *linewidth=None*, *alpha=1.0*)

Parameters

- **x_data** (*list[int]*) – a list containing x values of all the points being drawn. Length of list should match the length of y_data.
- **y_data** (*list[int]*) – a list containing y values of all the points being drawn. Length of list should match the length of x_data.
- **color** – color of the line. Refer to *matplotlib.colors* for a full list of formats that are accepted.
- **linestyle** – style of the line. Refer to *matplotlib.lines.Line2D* for a full list of formats that are accepted.

- **linewidth** (*float or None*) – width of the line. When it's None, a default value will be computed and used.
- **alpha** (*float*) – blending efficient. Smaller values lead to more transparent lines.

Returns *output (VisImage)* – image object with line drawn.

draw_binary_mask(*binary_mask*, *color=None*, ***, *edge_color=None*, *text=None*, *alpha=0.5*, *area_threshold=4096*)

Parameters

- **binary_mask** (*ndarray*) – numpy array of shape (H, W), where H is the image height and W is the image width. Each value in the array is either a 0 or 1 value of uint8 type.
- **color** – color of the mask. Refer to *matplotlib.colors* for a full list of formats that are accepted. If None, will pick a random color.
- **edge_color** – color of the polygon edges. Refer to *matplotlib.colors* for a full list of formats that are accepted.
- **text** (*str*) – if None, will be drawn in the object's center of mass.
- **alpha** (*float*) – blending efficient. Smaller values lead to more transparent masks.
- **area_threshold** (*float*) – a connected component small than this will not be shown.

Returns *output (VisImage)* – image object with mask drawn.

draw_polygon(*segment*, *color*, *edge_color=None*, *alpha=0.5*)

Parameters

- **segment** – numpy array of shape Nx2, containing all the points in the polygon.
- **color** – color of the polygon. Refer to *matplotlib.colors* for a full list of formats that are accepted.
- **edge_color** – color of the polygon edges. Refer to *matplotlib.colors* for a full list of formats that are accepted. If not provided, a darker shade of the polygon color will be used instead.
- **alpha** (*float*) – blending efficient. Smaller values lead to more transparent masks.

Returns *output (VisImage)* – image object with polygon drawn.

get_output()

Returns *output (VisImage)* – the image output containing the visualizations added to the image.

1.10.13 cvpods.utils.file.serialize module

class cvpods.utils.file.serialize.**PicklableWrapper**(*obj*)
Bases: object

Wrap an object to make it more picklable, note that it uses heavy weight serialization libraries that are slower than pickle. It's best to use it only on closures (which are usually not picklable).

This is a simplified version of https://github.com/joblib/joblib/blob/master/joblib/externals/loky/cloudpickle_wrapper.py

1.10.14 cvpods.utils.dump.events module

cvpods.utils.dump.events.**get_event_storage**()
Returns The *EventStorage* object that's currently being used. Throws an error if no :class`EventStorage` is currently enabled.

class cvpods.utils.dump.events.**EventWriter**
Bases: object

Base class for writers that obtain events from *EventStorage* and process them.

write()

close()

class cvpods.utils.dump.events.**JSONWriter**(*json_file*, *window_size=20*)
Bases: *cvpods.utils.dump.events.EventWriter*

Write scalars to a json file.

It saves scalars as one json per line (instead of a big json) for easy parsing.

Examples parsing such a json file:

```
$ cat metrics.json | jq -s '.[0:2]'
[
  {
    "data_time": 0.008433341979980469,
    "iteration": 20,
    "loss": 1.9228371381759644,
    "loss_box_reg": 0.050025828182697296,
    "loss_classifier": 0.5316952466964722,
    "loss_mask": 0.7236229181289673,
    "loss_rpn_box": 0.0856662318110466,
    "loss_rpn_cls": 0.48198649287223816,
    "lr": 0.007173333333333333,
    "time": 0.25401854515075684
  },
  {
    "data_time": 0.007216215133666992,
    "iteration": 40,
    "loss": 1.282649278640747,
    "loss_box_reg": 0.06222952902317047,
    "loss_classifier": 0.30682939291000366,
    "loss_mask": 0.6970193982124329,
    "loss_rpn_box": 0.038663312792778015,
    "loss_rpn_cls": 0.1471673548221588,
    "lr": 0.007706666666666667,
```

(continues on next page)

(continued from previous page)

```
    "time": 0.2490077018737793
  }
]

$ cat metrics.json | jq '.loss_mask'
0.7126231789588928
0.689423680305481
0.6776131987571716
...
```

__init__(*json_file*, *window_size=20*)

Parameters

- **json_file** (*str*) – path to the json file. New data will be appended if the file exists.
- **window_size** (*int*) – the window size of median smoothing for the scalars whose *smoothing_hint* are True.

write()

close()

class cvpods.utils.dump.events.**TensorboardXWriter**(*log_dir: str*, *window_size: int = 20*, ***kwargs*)

Bases: *cvpods.utils.dump.events.EventWriter*

Write all scalars to a tensorboard file.

__init__(*log_dir: str*, *window_size: int = 20*, ***kwargs*)

Parameters

- **log_dir** (*str*) – the directory to save the output events
- **window_size** (*int*) – the scalars will be median-smoothed by this window size
- **kwargs** – other arguments passed to *torch.utils.tensorboard.SummaryWriter(...)*

write()

close()

class cvpods.utils.dump.events.**CommonMetricPrinter**(*max_iter*, *window_size=20*, ***kwargs*)

Bases: *cvpods.utils.dump.events.EventWriter*

Print **common** metrics to the terminal, including iteration time, ETA, memory, all losses, and the learning rate.

To print something different, please implement a similar printer by yourself.

__init__(*max_iter*, *window_size=20*, ***kwargs*)

Parameters **max_iter** (*int*) – the maximum number of iterations to train. Used to compute ETA.

write()

class cvpods.utils.dump.events.**EventStorage**(*start_iter=0*, *window_size=20*)

Bases: object

The user-facing class that provides metric storage functionalities.

In the future we may add support for storing / logging other types of data if needed.

__init__ (*start_iter=0*, *window_size=20*)

Parameters **start_iter** (*int*) – the iteration number to start with

put_image (*img_name*, *img_tensor*)
Add an *img_tensor* to the *_vis_data* associated with *img_name*.

Parameters

- **img_name** (*str*) – The name of the image to put into tensorboard.
- **img_tensor** (*torch.Tensor or numpy.array*) – An *uint8* or *float* Tensor of shape *[channel, height, width]* where *channel* is 3. The image format should be RGB. The elements in img_tensor can either have values in [0, 1] (float32) or [0, 255] (uint8). The *img_tensor* will be visualized in tensorboard.

clear_images ()
Delete all the stored images for visualization. This should be called after images are written to tensorboard.

put_scalar (*name*, *value*, *smoothing_hint=True*)
Add a scalar *value* to the *HistoryBuffer* associated with *name*.

Parameters **smoothing_hint** (*bool*) – a 'hint' on whether this scalar is noisy and should be smoothed when logged. The hint will be accessible through *EventStorage.smoothing_hints()*. A writer may ignore the hint and apply custom smoothing rule.

It defaults to True because most scalars we save need to be smoothed to provide any useful signal.

put_scalars (*, *smoothing_hint=True*, ***kwargs*)
Put multiple scalars from keyword arguments.

Examples

storage.put_scalars(loss=my_loss, accuracy=my_accuracy, smoothing_hint=True)

history (*name*)

Returns *HistoryBuffer* – the scalar history for name

histories ()

Returns *dict[name -> HistoryBuffer]* – the HistoryBuffer for all scalars

latest ()

Returns *dict[name -> number]* – the scalars that's added in the current iteration.

latest_with_smoothing_hint ()
Similar to *latest()*, but the returned values are either the un-smoothed original latest value, or a median of the given window_size, depend on whether the smoothing_hint is True.

This provides a default behavior that other writers can use.

smoothing_hints ()

Returns

dict[name -> bool] –

the user-provided hint on whether the scalar is noisy and needs smoothing.

step()
User should call this function at the beginning of each iteration, to notify the storage of the start of a new iteration. The storage will then be able to associate the new data with the correct iteration number.

property vis_data

property iter

property iteration

name_scope(*name*)

Yields A context within which all the events added to this storage will be prefixed by the name scope.

1.10.15 cvpods.utils.dump.history_buffer module

class cvpods.utils.dump.history_buffer.**HistoryBuffer**(*max_length: int = 1000000*)
Bases: object

Track a series of scalar values and provide access to smoothed values over a window or the global average of the series.

__init__(*max_length: int = 1000000*)

Parameters **max_length** – maximal number of values that can be stored in the buffer. When the capacity of the buffer is exhausted, old values will be removed.

update(*value: float*, *iteration: float = None*)
Add a new scalar value produced at certain iteration. If the length of the buffer exceeds self._max_length, the oldest element will be removed from the buffer.

latest()
Return the latest scalar value added to the buffer.

median(*window_size: int*)
Return the median of the latest *window_size* values in the buffer.

avg(*window_size: int*)
Return the mean of the latest *window_size* values in the buffer.

global_avg()
Return the mean of all the elements in the buffer. Note that this includes those getting removed due to limited buffer storage.

values()

Returns *list[(number, iteration)]* – content of the current buffer.

1.10.16 cvpods.utils.dump.logger module

cvpods.utils.dump.logger.**setup_logger**(*output=None*, *distributed_rank=0*, ***, *color=True*, *name='cvpods'*, *abbrev_name=None*)
Initialize the cvpods logger and set its verbosity level to "INFO".
Parameters

- **output** (*str*) – a file name or a directory to save log. If None, will not save log file. If ends with ".txt" or ".log", assumed to be a file name. Otherwise, logs will be saved to *output/log.txt*.
- **name** (*str*) – the root module name of this logger

- **abbrev_name** (*str*) – an abbreviation of the module, to avoid long names in logs. Set to "" to not log the root module in logs. By default, will abbreviate "cvpods" to "c2" and leave other modules unchanged.

Returns *logging.Logger* – a logger

cvpods.utils.dump.logger.**log_first_n**(*lvl*, *msg*, *n=1*, ***, *name=None*, *key='caller'*)

Log only for the first n times.

Parameters

- **lvl** (*int*) – the logging level
- **msg** (*str*) –
- **n** (*int*) –
- **name** (*str*) – name of the logger to use. Will use the caller's module by default.
- **key** (*str or tuple[str]*) – the string(s) can be one of "caller" or "message", which defines how to identify duplicated logs. For example, if called with *n=1, key="caller"*, this function will only log the first call from the same caller, regardless of the message content. If called with *n=1, key="message"*, this function will log the same content only once, even if they are called from different places. If called with *n=1, key=("caller", "message")*, this function will not log only if the same caller has logged the same message before.

cvpods.utils.dump.logger.**log_every_n**(*lvl*, *msg*, *n=1*, ***, *name=None*)

Log once per n times.

Parameters

- **lvl** (*int*) – the logging level
- **msg** (*str*) –
- **n** (*int*) –
- **name** (*str*) – name of the logger to use. Will use the caller's module by default.

cvpods.utils.dump.logger.**log_every_n_seconds**(*lvl*, *msg*, *n=1*, ***, *name=None*)

Log no more than once per n seconds. :param lvl: the logging level :type lvl: int :param msg: :type msg: str :param n: :type n: int :param name: name of the logger to use. Will use the caller's module by default. :type name: str

cvpods.utils.dump.logger.**create_small_table**(*small_dict*)

Create a small table using the keys of small_dict as headers. This is only suitable for small dictionaries.

Parameters **small_dict** (*dict*) – a result dictionary of only a few items.

Returns *str* – the table as a string.

cvpods.utils.dump.logger.**create_table_with_header**(*header_dict*, *headers=['category', 'AP']*, *min_cols=6*)

create a table with given header.

Parameters

- **header_dict** (*dict*) –
- **headers** (*list*) –
- **min_cols** (*int*) –

Returns *str* – the table as a string

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X