ndsampler Documentation

Release 0.7.0

Jon Crall

Aug 11, 2022

CONTENTS:

CHAPTER

ONE

API REFERENCE

This page contains auto-generated API reference documentation[1].

1.1 ndsampler

mkinit ~/code/ndsampler/ndsampler/__init__.py –diff mkinit ~/code/ndsampler/ndsampler/__init__.py -w

1.1.1 Subpackages

ndsampler.utils

Submodules

ndsampler.utils.util_futures

Note: this module also exists in kwcoco.utils

Module Contents

Classes

SerialExecutor	Implements the concurrent.futures API around a single-threaded backend
Executor	Wrapper around a specific executor.

class ndsampler.utils.util_futures.**SerialExecutor**

Bases: object

Implements the concurrent.futures API around a single-threaded backend

[1] Created with sphinx-autoapi

Example

```
>>> with SerialExecutor() as executor:
>>>     futures = []
>>>     for i in range(100):
>>>         f = executor.submit(lambda x: x + 1, i)
>>>         futures.append(f)
>>>     for f in concurrent.futures.as_completed(futures):
>>>         assert f.result() > 0
>>>     for i, f in enumerate(futures):
>>>         assert i + 1 == f.result()
```

__enter__()

__exit__(*ex_type*, *ex_value*, *tb*)

submit(*func*, **args*, ***kw*)

shutdown()

class ndsampler.utils.util_futures.**Executor**(*mode='thread'*, *max_workers=0*)

Bases: object

Wrapper around a specific executor.

Abstracts Serial, Thread, and Process Executor via arguments.

Parameters

- **mode** (*str, default='thread'*) – either thread, serial, or process
- **max_workers** (*int, default=0*) – number of workers. If 0, serial is forced.

__enter__()

__exit__(*ex_type*, *ex_value*, *tb*)

submit(*func*, **args*, ***kw*)

shutdown()

ndsampler.utils.util_gdal

Module Contents

Classes

LazyGDalFrameFile

Functions

have_gdal()	
_fix_conda_gdal_hack()	
_doctest_check_cog(data, fpath)	
_numpy_to_gdal_dtype(numpy_dtype)	
_benchmark_cog_conversions()	CommandLine:
_imwrite_cloud_optimized_geotiff(fpath, data[, ...])	**Parameters** • **fpath** (*PathLike*) -- file path to save the COG to.
_dtype_equality(dtype1, dtype2)	Check for numpy dtype equality
_auto_compress([src_fpath, data, data_set])	Heuristic for automatically choosing compression type
_cli_convert_cloud_optimized_geotiff(src_fpath, dst_fpath)	For whatever reason using the CLI seems to simply be faster.
_api_convert_cloud_optimized_geotiff(src_fpath, dst_fpath)	Optimization of imwrite specifically for converting files that already
_api_convert_cloud_optimized_geotiff2(src_fpath, dst_fpath)	CommandLine:
_rectify_slice_dim(part, D)	
validate_nonzero_data(file)	Test to see if the image is all black.
batch_convert_to_cog(src_fpaths, dst_fpaths[, mode, ...])	Converts many input images to COGs and verifies that the outputs are
batch_validate_cog(dst_fpaths[, mode, max_workers])	Return cog infos
_validate_cog_worker(dst_fpath[, orig_fpath])	
_convert_to_cog_worker(src_fpath, dst_fpath, cog_config)	worker function

Attributes

profile
_GDAL_DTYPE_LUT

ndsampler.utils.util_gdal.**profile**

ndsampler.utils.util_gdal.**have_gdal**()

ndsampler.utils.util_gdal.**_fix_conda_gdal_hack**()

ndsampler.utils.util_gdal.**_doctest_check_cog**(*data*, *fpath*)

ndsampler.utils.util_gdal.**_numpy_to_gdal_dtype**(*numpy_dtype*)

ndsampler.utils.util_gdal.**_benchmark_cog_conversions**()

CommandLine:
xdoctest -m ~/code/ndsampler/ndsampler/utils/util_gdal.py _benchmark_cog_conversions

ndsampler.utils.util_gdal.**_imwrite_cloud_optimized_geotiff**(*fpath*, *data*, *compress='auto'*, *blocksize=256*)

Parameters

- **fpath** (*PathLike*) – file path to save the COG to.
- **data** (*ndarray[ndim=3]*) – Raw HWC image data to save. Dimensions should be height, width, channels.
- **compress** (*bool, default='auto'*) – Can be JPEG (lossy) or LZW (lossless), or DEFLATE (lossless). Can also be 'auto', which will try to hueristically choose a sensible choice.
- **blocksize** (*int, default=256*) – size of tiled blocks

References

https://geoexamples.com/other/2019/02/08/cog-tutorial.html#create-a-cog-using-gdal-python http://osgeo-org.1560.x6.nabble.com/gdal-dev-Creating-Cloud-Optimized-GeoTIFFs-td5320101.html https://gdal.org/drivers/raster/cog.html https://github.com/harshurampur/Geotiff-conversion https://github.com/sshuair/cogeotiff https://github.com/cogeotiff/rio-cogeo

Notes

conda install gdal

OR

sudo apt install gdal-dev pip install gdal —with special flags, forgot which though, sry

CommandLine:
xdoctest -m ndsampler.utils.util_gdal _imwrite_cloud_optimized_geotiff

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from ndsampler.utils.util_gdal import *  # NOQA
>>> from ndsampler.utils.util_gdal import _imwrite_cloud_optimized_geotiff
>>> from ndsampler.utils.util_gdal import _doctest_check_cog
```

```
>>> data = np.random.randint(0, 255, (800, 800, 3), dtype=np.uint8)
>>> fpath = '/tmp/foo.cog.tiff'
>>> compress = 'JPEG'
>>> _imwrite_cloud_optimized_geotiff(fpath, data, compress='JPEG')
>>> assert _doctest_check_cog(data, fpath)
>>> _imwrite_cloud_optimized_geotiff(fpath, data, compress='LZW')
>>> assert _doctest_check_cog(data, fpath)
```

```
>>> data = (np.random.rand(100, 100, 4) * 255).astype(np.uint8)
>>> _imwrite_cloud_optimized_geotiff(fpath, data, compress='JPEG')
>>> assert _doctest_check_cog(data, fpath)
>>> _imwrite_cloud_optimized_geotiff(fpath, data, compress='LZW')
>>> assert _doctest_check_cog(data, fpath)
>>> _imwrite_cloud_optimized_geotiff(fpath, data, compress='DEFLATE')
>>> assert _doctest_check_cog(data, fpath)
```

```
>>> data = (np.random.rand(100, 100, 5) * 255).astype(np.uint8)
>>> _imwrite_cloud_optimized_geotiff(fpath, data, compress='LZW')
>>> assert _doctest_check_cog(data, fpath)
```

Ignore:

```
>>> import xdev
>>> data = np.random.randint(0, 255, (8000, 8000, 3), dtype=np.uint8)
>>> print(xdev.byte_str(data.size * data.dtype.itemsize))
>>> fpath = fpath1 = ub.expandpath('~/ssd/foo.tiff')
>>> fpath2 = ub.expandpath('~/raid/foo.tiff')
>>> import ubelt as ub
>>> ti = ub.Timerit(1, bestof=1, verbose=3)
>>> #
>>> for timer in ti.reset('SSD'):
>>>     ub.delete(fpath1)
>>>     with timer:
>>>         _imwrite_cloud_optimized_geotiff(fpath1, data)
>>> for timer in ti.reset('HDD'):
>>>     ub.delete(fpath2)
>>>     with timer:
>>>         _imwrite_cloud_optimized_geotiff(fpath2, data)
```

ndsampler.utils.util_gdal.**_dtype_equality**(*dtype1*, *dtype2*)

Check for numpy dtype equality

References

https://stackoverflow.com/questions/26921836/correct-way-to-test-for-numpy-dtype

Example

dtype1 = np.empty(0, dtype=np.uint8).dtype dtype2 = np.uint8 _dtype_equality(dtype1, dtype2)

ndsampler.utils.util_gdal.**_auto_compress**(*src_fpath=None*, *data=None*, *data_set=None*)

Heuristic for automatically choosing compression type

Parameters

- **src_fpath** (*str*) – path to source image if known
- **data** (*ndarray*) – data pixels if known
- **data_set** (*gdal.Dataset*) – gdal dataset if known

Returns
gdal compression code

Return type
str

Example

```
>>> assert _auto_compress(src_fpath='foo.jpg') == 'JPEG'
>>> assert _auto_compress(src_fpath='foo.png') == 'LZW'
>>> assert _auto_compress(data=np.random.rand(3, 2)) == 'RAW'
>>> assert _auto_compress(data=np.random.rand(3, 2, 3).astype(np.uint8)) == 'JPEG'
>>> assert _auto_compress(data=np.random.rand(3, 2, 4).astype(np.uint8)) == 'RAW'
>>> assert _auto_compress(data=np.random.rand(3, 2, 1).astype(np.uint8)) == 'RAW'
```

ndsampler.utils.util_gdal.**_cli_convert_cloud_optimized_geotiff**(*src_fpath*, *dst_fpath*, *compress='auto'*, *blocksize=256*)

For whatever reason using the CLI seems to simply be faster.

Parameters

- **src_fpath** (*PathLike*) – file path to convert
- **dst_fpath** (*PathLike*) – file path to save the COG to.
- **blocksize** (*int, default=256*) – size of tiled blocks
- **compress** (*bool, default='auto'*) – Can be JPEG (lossy) or LZW (lossless), or DEFLATE (lossless). Can also be 'auto', which will try to hueristically choose a sensible choice.

Ignore:

```
>>> import xdev
>>> import kwimage
>>> data = np.random.randint(0, 255, (4000, 4000, 3), dtype=np.uint8)
>>> print(xdev.byte_str(data.size * data.dtype.itemsize))
>>> src_fpath = ub.expandpath('~/raid/src.tiff')
>>> kwimage.imwrite(src_fpath, data)
>>> dst_fpath = ub.expandpath('~/raid/dst.tiff')
>>> ti = ub.Timerit(1, bestof=1, verbose=3)
>>> for timer in ti.reset('SSD-api'):
>>>     _cli_convert_cloud_optimized_geotiff(src_fpath, dst_fpath)
```

ndsampler.utils.util_gdal.**_api_convert_cloud_optimized_geotiff**(*src_fpath*, *dst_fpath*, *compress='JPEG'*, *blocksize=256*)

Optimization of imwrite specifically for converting files that already exist on disk. Skipping the initial load of data can be very helpful.

CommandLine:
xdoctest -m ~/code/ndsampler/ndsampler/utils/util_gdal.py _api_convert_cloud_optimized_geotiff –bench

ndsampler.utils.util_gdal.**_api_convert_cloud_optimized_geotiff2**(*src_fpath*, *dst_fpath*, *compress='JPEG'*, *blocksize=256*)

CommandLine:

xdoctest -m ~/code/ndsampler/ndsampler/utils/util_gdal.py _api_convert_cloud_optimized_geotiff –bench

ndsampler.utils.util_gdal.**_GDAL_DTYPE_LUT**

class ndsampler.utils.util_gdal.**LazyGDalFrameFile**(*cog_fpath*)

Bases: ubelt.NiceRepr

Todo:

- [ ] Move to its own backend module
- **[ ] When used with COCO, allow the image metadata to populate the** height, width, and channels if possible.

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from ndsampler.utils.util_gdal import *  # NOQA
>>> self = LazyGDalFrameFile.demo()
>>> cog_fpath = self.cog_fpath
>>> print('self = {!r}'.format(self))
>>> self[0:3, 0:3]
>>> self[:, :, 0]
>>> self[0]
>>> self[0, 3]
```

```
>>> # import kwplot
>>> # kwplot.imshow(self[:])
```

_ds()

classmethod demo(*key='astro'*, *dsize=None*)

Ignore:

```
>>> from ndsampler.utils.util_gdal import *  # NOQA
>>> self = LazyGDalFrameFile.demo(dsize=(6600, 4400))
```

property ndim

property shape

property dtype

__nice__()

__getitem__(*index*)

References

https://gis.stackexchange.com/questions/162095/gdal-driver-create-typeerror

Ignore:

```
>>> from ndsampler.utils.util_gdal import *  # NOQA
>>> self = LazyGDalFrameFile.demo(dsize=(6600, 4400))
>>> index = [slice(2100, 2508, None), slice(4916, 5324, None), None]
>>> img_part = self[index]
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img_part)
```

__array__()

Allow this object to be passed to np.asarray

References

https://numpy.org/doc/stable/user/basics.dispatch.html

validate(*orig_fpath=None*, *orig_data=None*)

Check for any corruption issues

Parameters

- **orig_fpath** (*str*) – if specified and the data seems to be all zero, we check if the pixels are close to the pixels in this other image. If not the warning becomes an error.
- **orig_data** (*ndarray*) – alternative to orig_fpath if the data is in memory.

Returns

info about errors, warnings, and details

Return type

Dict

ndsampler.utils.util_gdal.**_rectify_slice_dim**(*part*, *D*)

ndsampler.utils.util_gdal.**validate_nonzero_data**(*file*)

Test to see if the image is all black.

May fail on all-black images

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> from ndsampler.utils.util_gdal import LazyGDalFrameFile
>>> import kwimage
>>> gpath = kwimage.grab_test_image_fpath()
>>> file = LazyGDalFrameFile(gpath)
>>> validate_nonzero_data(file)
```

ndsampler.utils.util_gdal.**batch_convert_to_cog**(*src_fpaths*, *dst_fpaths*, *mode='process'*, *max_workers=0*, *cog_config=None*)

Converts many input images to COGs and verifies that the outputs are correct

Parameters

- **src_fpaths** (*List[str]*) – source image filepaths
- **dst_fpaths** (*List[str]*) – corresponding destination image filepaths
- **mode** (*str, default='process'*) – either process, thread, or serial
- **max_workers** (*int, default=0*) – number of processes / threads to use
- **cog_config** (*dict*) – config options for COG files (e.g. compress, blocksize, overviews, etc).

ndsampler.utils.util_gdal.**batch_validate_cog**(*dst_fpaths*, *mode='thread'*, *max_workers=0*)

Return cog infos

Parameters

- **dst_fpaths** (*List[str]*) – paths to validate
- **mode** (*str, default='process'*) – either process, thread, or serial
- **max_workers** (*int, default=0*) – number of processes / threads to use

ndsampler.utils.util_gdal.**_validate_cog_worker**(*dst_fpath*, *orig_fpath=None*)

ndsampler.utils.util_gdal.**_convert_to_cog_worker**(*src_fpath*, *dst_fpath*, *cog_config*)

worker function

ndsampler.utils.util_lru

Module Contents

Classes

LRUDict	Pure python implementation for lru cache fallback

Functions

_benchmarks()	Test the speed of LRU implementations and ensure the API is the same.

class ndsampler.utils.util_lru.**LRUDict**(*max_size*)

Bases: ubelt.NiceRepr

Pure python implementation for lru cache fallback

References

https://github.com/amitdev/lru-dict pip install lru-dict http://www.kunxi.org/blog/2014/05/lru-cache-in-python/

Parameters
max_size (*int*) – (default = 5)

Returns
cache_obj

Return type
LRUDict

CommandLine:
xdoctest -m /home/joncrall/code/ndsampler/ndsampler/utils/util_lru.py LRUDict

Example

```
>>> from ndsampler.utils.util_lru import *  # NOQA
>>> max_size = 5
>>> self = LRUDict(max_size)
>>> for count in range(0, 5):
...     self[count] = count
>>> print(self)
>>> self[0]
>>> for count in range(5, 8):
...     self[count] = count
>>> print(self)
>>> del self[5]
>>> assert 4 in self
>>> # xdoctest: +IGNORE_WANT
>>> print('self = {}'.format(ub.repr2(self, nl=1)))
self = <LRUDict({4: 4, 0: 0, 6: 6, 7: 7}) at 0x7f4c78af95e0>
```

__contains__(*item*)

__delitem__(*key*)

__nice__()

update(*other*)

__iter__()

items()

keys()

values()

iteritems()

iterkeys()

itervalues()

clear()

__len__()

__getitem__(*key*)

__setitem__(*key*, *value*)

has_key(*item*)

set_size(*max_size*)

change the capacity of the LRU cache

Example

```
>>> self = LRUDict(10)
>>> self.update(dict(zip(range(10), range(10))))
>>> assert len(self) == 10
>>> self.set_size(2)
>>> assert len(self) == 2
>>> self.set_size(10)
>>> assert len(self) == 2
```

classmethod new(*max_size*, *impl='auto'*)

Creates an LRU dictionary instance, but uses the efficient c-backend if that is available.

Parameters

- **max_size** (*int*)
- **impl** (*str, default='auto'*) – which implementation to use

Example:

ndsampler.utils.util_lru.**_benchmarks**()

Test the speed of LRU implementations and ensure the API is the same.

CommandLine:

xdoctest -m ndsampler.utils.util_lru _benchmarks

Results:

— impl=c — Timed c-integer-test for: 10 loops, best of 3

time per loop: best=198.068 ms, mean=205.546 ± 5.1 ms

Timed c-miss-case for: 10 loops, best of 3

time per loop: best=405.937 µs, mean=410.180 ± 4.9 µs

Timed c-hit-case for: 10 loops, best of 3

time per loop: best=257.571 µs, mean=266.696 ± 8.3 µs

Timed c-clear-test for: 10 loops, best of 3

time per loop: best=341.119 µs, mean=350.805 ± 7.2 µs

— impl=py — Timed py-integer-test for: 10 loops, best of 3

time per loop: best=896.410 ms, mean=898.566 ± 1.8 ms

Timed py-miss-case for: 10 loops, best of 3
time per loop: best=2.041 ms, mean=2.116 ± 0.1 ms

Timed py-hit-case for: 10 loops, best of 3
time per loop: best=1.095 ms, mean=1.119 ± 0.0 ms

Timed py-clear-test for: 10 loops, best of 3
time per loop: best=1.402 ms, mean=1.451 ± 0.0 ms

Conclusion:
As expected, the c backend is more performant.

ndsampler.utils.util_misc

Module Contents

Classes

HashIdentifiable	A class is hash-identifiable if its invariants can be tied to a specific

class ndsampler.utils.util_misc.**HashIdentifiable**(***kwargs*)

Bases: object

A class is hash-identifiable if its invariants can be tied to a specific list of hashable dependencies.

The inheriting class must either:

- implement *_depends*
- implement *_make_hashid*
- define *_hashid*

Example

class Base:

def __init__(self):
commenting the next line removes cooperative inheritence super().__init__() self.base = 1

class Derived(Base, HashIdentifiable):

def __init__(self):
super().__init__() self.defived = 1

self = Derived() dir(self)

abstract _depends()

_make_hashid()

property hashid

ndsampler.utils.util_shape

Module Contents

Functions

nestshape(data)	Examine nested shape of the data

ndsampler.utils.util_shape.**nestshape**(*data*)

Examine nested shape of the data

Example

```
>>> data = [np.arange(10), np.arange(13)]
>>> nestshape(data)
[(10,), (13,)]
```

ndsampler.utils.util_sklearn

Extensions to sklearn constructs

Module Contents

Classes

StratifiedGroupKFold	Stratified K-Folds cross-validator with Grouping

class ndsampler.utils.util_sklearn.**StratifiedGroupKFold**(*n_splits=3*, *shuffle=False*, *random_state=None*)

Bases: sklearn.model_selection._split._BaseKFold

Stratified K-Folds cross-validator with Grouping

Provides train/test indices to split data in train/test sets.

This cross-validation object is a variation of GroupKFold that returns stratified folds. The folds are made by preserving the percentage of samples for each class.

This is an old interface and should likely be refactored and modernized.

Parameters

n_splits (*int, default=3*) – Number of folds. Must be at least 2.

_make_test_folds(*X*, *y=None*, *groups=None*)

Parameters

- **X** (*ndarray*) – data
- **y** (*ndarray*) – labels

- **groups** (*ndarray*) – groupids for items. Items with the same groupid must be placed in the same group.

Returns
test_folds

Return type
list

Example

```
>>> import kwarray
>>> rng = kwarray.ensure_rng(0)
>>> groups = [1, 1, 3, 4, 2, 2, 7, 8, 8]
>>> y      = [1, 1, 1, 1, 2, 2, 2, 3, 3]
>>> X = np.empty((len(y), 0))
>>> self = StratifiedGroupKFold(random_state=rng, shuffle=True)
>>> skf_list = list(self.split(X=X, y=y, groups=groups))
...
>>> import ubelt as ub
>>> print(ub.repr2(skf_list, nl=1, with_dtype=False))
[
    (np.array([2, 3, 4, 5, 6]), np.array([0, 1, 7, 8])),
    (np.array([0, 1, 2, 7, 8]), np.array([3, 4, 5, 6])),
    (np.array([0, 1, 3, 4, 5, 6, 7, 8]), np.array([2])),
]
```

_iter_test_masks(*X*, *y=None*, *groups=None*)

Generates boolean masks corresponding to test sets.

By default, delegates to _iter_test_indices(X, y, groups)

split(*X*, *y*, *groups=None*)

Generate indices to split data into training and test set.

ndsampler.utils.validate_cog

Module Contents

Functions

Usage()	
validate(ds[, check_tiled])	Check if a file is a (Geo)TIFF with cloud optimized compatible structure.
main()	Return 0 in case of success, 1 for failure.

ndsampler.utils.validate_cog.**Usage**()

exception ndsampler.utils.validate_cog.**ValidateCloudOptimizedGeoTIFFException**

Bases: Exception

Common base class for all non-exit exceptions.

ndsampler.utils.validate_cog.**validate**(*ds*, *check_tiled=True*)

Check if a file is a (Geo)TIFF with cloud optimized compatible structure.

Parameters

- **ds** – GDAL Dataset for the file to inspect.
- **check_tiled** – Set to False to ignore missing tiling.

Returns

Tuple[List, List, Dict] - warnings, errors, details - A tuple, whose first element is an array of error messages (empty if there is no error), and the second element, a dictionary with the structure of the GeoTIFF file.

Raises

ValidateCloudOptimizedGeoTIFFException – Unable to open the file or the file is not a Tiff.

ndsampler.utils.validate_cog.**main**()

Return 0 in case of success, 1 for failure.

1.1.2 Submodules

ndsampler._internal

Module Contents

Functions

_boolean_environ(key)

Attributes

NDSAMPLER_DISABLE_WARNINGS

NDSAMPLER_DISABLE_OPTIONAL_WARNINGS

ndsampler._internal.**_boolean_environ**(*key*)

ndsampler._internal.**NDSAMPLER_DISABLE_WARNINGS**

ndsampler._internal.**NDSAMPLER_DISABLE_OPTIONAL_WARNINGS**

ndsampler.abstract_frames

Fast access to subregions of images.

This implements the core convert-and-cache-as-cog logic, which enables us to read from subregions of images quickly.

Todo:

- [X] Implement npy memmap backend
- **[X] Implement gdal COG.TIFF backend**
  - [X] Use as COG if input file is a COG
  - [X] Convert to COG if needed

Module Contents

Classes

Frames	Abstract implementation of Frames.
SimpleFrames	Basic concrete implementation of frames objects for images where there is a
AlignableImageData	Class for sampling channels / frames that are aligned with each other

Attributes

profile

ndsampler.abstract_frames.**profile**

class ndsampler.abstract_frames.**Frames**(*hashid_mode='PATH'*, *workdir=None*, *backend=None*)

Bases: object

Abstract implementation of Frames.

While this is an abstract class, it contains most of the `Frames` functionality. The inheriting class needs to overload the constructor and `_lookup_gpath`, which maps an image-id to its path on disk.

Parameters

- **hashid_mode** (*str, default='PATH'*) – The method used to compute a unique identifier for every image. to can be PATH, PIXELS, or GIVEN. TODO: Add DVC as a method (where it uses the name of the symlink)?
- **workdir** (*PathLike*) – This is the directory where *Frames* can store cached results. This SHOULD be specified.
- **backend** (*str | Dict*) – Determine the backend to use for fast subimage region lookups. This can either be a string 'cog' or 'npy'. This can also be a config dictionary for fine-grained backend control. For this case, 'type': specified cog or npy, and only COG has additional options which are:

{
'type': 'cog', 'config': { 'compress': <'LZW' | 'JPEG | 'DEFLATE' | 'ZSTD' | 'auto'>, }

}

Example

```
>>> from ndsampler.abstract_frames import *
>>> self = SimpleFrames.demo(backend='npy')
>>> file = self.load_image(1)
>>> print('file = {!r}'.format(file))
>>> assert self.load_image(1).shape == (512, 512, 3)
>>> assert self.load_region(1, (slice(-20), slice(-10))).shape == (492, 502, 3)
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> self = SimpleFrames.demo(backend='cog')
>>> assert self.load_image(1).shape == (512, 512, 3)
>>> assert self.load_region(1, (slice(-20), slice(-10))).shape == (492, 502, 3)
```

Benchmark:

```
>>> from ndsampler.abstract_frames import *  # NOQA
>>> import ubelt as ub
>>> #
>>> ti = ub.Timerit(100, bestof=3, verbose=2)
>>> #
>>> self = SimpleFrames.demo(backend='cog')
>>> for timer in ti.reset('cog-small-subregion'):
>>>     self.load_image(1)[10:42, 10:42]
>>> #
>>> self = SimpleFrames.demo(backend='npy')
>>> for timer in ti.reset('npy-small-subregion'):
>>>     self.load_image(1)[10:42, 10:42]
>>> print('----')
>>> #
>>> self = SimpleFrames.demo(backend='cog')
>>> for timer in ti.reset('cog-large-subregion'):
>>>     self.load_image(1)[3:-3, 3:-3]
>>> #
>>> self = SimpleFrames.demo(backend='npy')
>>> for timer in ti.reset('npy-large-subregion'):
>>>     self.load_image(1)[3:-3, 3:-3]
>>> print('----')
>>> #
>>> self = SimpleFrames.demo(backend='cog')
>>> for timer in ti.reset('cog-loadimage'):
>>>     self.load_image(1)
>>> #
>>> self = SimpleFrames.demo(backend='npy')
>>> for timer in ti.reset('npy-loadimage'):
>>>     self.load_image(1)
```

DEFAULT_NPY_CONFIG

DEFAULT_COG_CONFIG

__getstate__()

__setstate__(*state*)

_update_backend(*backend*)

change the backend and update internals accordingly

classmethod _coerce_backend_config(*backend=None*)

Coerce a backend argument into a valid configuration dictionary.

Returns

a dictionary with two items: 'type', which is a string and
and 'config', which is a dictionary of parameters for the specific type.

Return type
Dict

property cache_dpath

Returns the path where cached frame representations will be stored.

This will be None if there is no backend.

abstract _build_pathinfo(*image_id*)

A user specified function that maps an image id to paths to relevant resources on disk. These resources are also indexed by channel.

SeeAlso:
_populate_chan_info for helping populate cache info in each channel.

Parameters
image_id – the image id (usually an integer)

Returns

with the following structure:

{
<NotFinalized> 'channels': {
<channel_spec>: {'path': <abspath>, ...}, ...
}
}

Return type
Dict

_lookup_pathinfo(*image_id*)

_populate_chan_info(*chan*, *root=''*)

Helper to construct a path dictionary in the _build_pathinfo method based on the current hashing and caching settings.

static _build_file_hashid(*root*, *suffix*, *hashid_mode*)

Build a hashid for a specific file given as a path root and suffix.

property image_ids

__len__()

__getitem__(*index*)

load_region(*image_id*, *region=None*, *channels=ub.NoParam*, *width=None*, *height=None*)

Ammortized O(1) image subregion loading (assuming constant region size)

Parameters

- **image_id** (*int*) – image identifier
- **region** (*Tuple[slice, ...]*) – space-time region within an image
- **channels** (*str*) – NotImplemented
- **width** (*int*) – if the width of the entire image is know specify it
- **height** (*int*) – if the height of the entire image is know specify it

_load_alignable(*image_id*, *cache=True*)

load_image(*image_id*, *channels=ub.NoParam*, *cache=True*, *noreturn=False*)

Load the image data for a particular image id

Parameters

- **image_id** (*int*) – the id of the image to load
- **cache** (*bool, default=True*) – ensure and return the efficient backend cached representation.
- **channels** – NotImplemented
- **noreturn** (*bool, default=False*) – if True, nothing is returned. This is useful if you simply want to ensure the cached representation.

CAREFUL: THIS NEEDS TO MAINTAIN A STABLE API. OTHER PROJECTS DEPEND ON IT.

Returns

an indexable array like representation, possibly
memmapped.

Return type
ArrayLike

load_frame(*image_id*)

TODO: FINISHME or rename to lazy frame?

Returns a frame object that lazy loads on slice

prepare(*gids=None*, *workers=0*, *use_stamp=True*)

Precompute the cached frame conversions

Parameters

- **gids** (*List[int] | None*) – specific image ids to prepare. If None prepare all images.
- **workers** (*int, default=0*) – number of parallel threads for this io-bound task

Example

```
>>> from ndsampler.abstract_frames import *
>>> workdir = ub.ensure_app_cache_dir('ndsampler/tests/test_cog_precomp')
>>> print('workdir = {!r}'.format(workdir))
>>> ub.delete(workdir)
>>> ub.ensuredir(workdir)
>>> self = SimpleFrames.demo(backend='npy', workdir=workdir)
>>> print('self = {!r}'.format(self))
>>> print('self.cache_dpath = {!r}'.format(self.cache_dpath))
>>> #_ = ub.cmd('tree ' + workdir, verbose=3)
>>> self.prepare()
>>> self.prepare()
>>> #_ = ub.cmd('tree ' + workdir, verbose=3)
>>> _ = ub.cmd('ls ' + self.cache_dpath, verbose=3)
```

Example

```
>>> from ndsampler.abstract_frames import *
>>> import ndsampler
>>> workdir = ub.get_app_cache_dir('ndsampler/tests/test_cog_precomp2')
>>> ub.delete(workdir)
>>> # TEST NPY
>>> #
>>> sampler = ndsampler.CocoSampler.demo(workdir=workdir, backend='npy')
>>> self = sampler.frames
>>> ub.delete(self.cache_dpath)  # reset
>>> self.prepare()  # serial, miss
>>> self.prepare()  # serial, hit
>>> ub.delete(self.cache_dpath)  # reset
>>> self.prepare(workers=3)  # parallel, miss
>>> self.prepare(workers=3)  # parallel, hit
>>> #
>>> ## TEST COG
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> sampler = ndsampler.CocoSampler.demo(workdir=workdir, backend='cog')
>>> self = sampler.frames
>>> ub.delete(self.cache_dpath)  # reset
>>> self.prepare()  # serial, miss
>>> self.prepare()  # serial, hit
>>> ub.delete(self.cache_dpath)  # reset
>>> self.prepare(workers=3)  # parallel, miss
>>> self.prepare(workers=3)  # parallel, hit
```

class ndsampler.abstract_frames.**SimpleFrames**(*id_to_path*, *workdir=None*, *backend=None*)

Bases: *Frames*

Basic concrete implementation of frames objects for images where there is a strict one-file-to-one-image mapping (i.e. no auxiliary images).

Parameters

id_to_path (*Dict*) – mapping from image-id to image path

Example

```
>>> from ndsampler.abstract_frames import *
>>> self = SimpleFrames.demo(backend='npy')
>>> pathinfo = self._build_pathinfo(1)
>>> print('pathinfo = {}'.format(ub.repr2(pathinfo, nl=3)))
```

```
>>> assert self.load_image(1).shape == (512, 512, 3)
>>> assert self.load_region(1, (slice(-20), slice(-10))).shape == (492, 502, 3)
```

_lookup_gpath(*image_id*)

image_ids()

classmethod demo(***kw*)

Get a smple frames object

_build_pathinfo(*image_id*)

A user specified function that maps an image id to paths to relevant resources on disk. These resources are also indexed by channel.

SeeAlso:

`_populate_chan_info` for helping populate cache info in each channel.

Parameters

image_id – the image id (usually an integer)

Returns

with the following structure:

{
<NotFinalized> 'channels': {
<channel_spec>: {'path': <abspath>, ...}, ...
}
}

Return type

Dict

class ndsampler.abstract_frames.**AlignableImageData**(*pathinfo*, *cache_backend*)

Bases: `object`

Class for sampling channels / frames that are aligned with each other

---

Todo:

- [ ] This is more general than the older way of accessing image data

however, there is a lot more logic that hasn't been profiled, so we may be able to find meaningful optimizations.

- [ ] Make sure adding this didnt significantly hurt performance
- [ ] DEPRECATE THIS IN FAVOR OF NEW KWCOCO DELAYED LOGIC

Example

```
>>> from ndsampler.abstract_frames import *
>>> frames = SimpleFrames.demo(backend='npy')
>>> pathinfo = frames._build_pathinfo(1)
>>> cache_backend = frames._backend
>>> print('pathinfo = {}'.format(ub.repr2(pathinfo, nl=3)))
>>> self = AlignableImageData(pathinfo, cache_backend)
>>> img_region = None
>>> prefused = self._load_prefused_region(img_region)
>>> print('prefused = {!r}'.format(prefused))
>>> img_region = (slice(0, 10), slice(0, 10))
>>> prefused = self._load_prefused_region(img_region)
>>> print('prefused = {!r}'.format(prefused))
```

_load_native_channel(*chan_name*, *cache=True*)

Load a specific auxiliary channel, optionally caching it

_load_delayed_channel(*chan_name*, *cache=True*)

_coerce_channels(*channels=ub.NoParam*)

_load_prefused_region(*img_region*, *channels=ub.NoParam*)

Loads crops from multiple channels in their native coordinate system packaged with transformation info on how to align them.

_load_fused_region(*img_region*, *channels=ub.NoParam*)

Loads crops from multiple channels in aligned base coordinates.

load_region(*img_region*, *channels=ub.NoParam*, *fused=True*)

Parameters

img_region (*Tuple[slice, ...]*) – slice into the base image (will be warped into the auxiliary image's frames)

__getitem__(*img_region*)

ndsampler.abstract_sampler

Module Contents

Classes

AbstractSampler	API for Samplers, not all methods need to be implemented depending on the

class ndsampler.abstract_sampler.**AbstractSampler**

Bases: object

API for Samplers, not all methods need to be implemented depending on the use case (for example, load_sample may not be defined if positive / negative cases are generated on the fly).

property class_ids

abstract lookup_class_name(*class_id*)

abstract lookup_class_id(*class_name*)

abstract load_sample(*tr*, *pad=None*, *window_dims=None*, *visible_thresh=0.1*)

property n_positives

abstract load_item(*index*, *pad=None*, *window_dims=None*)

abstract load_positive(*index=None*, *pad=None*, *window_dims=None*, *rng=None*)

abstract load_negative(*index=None*, *pad=None*, *window_dims=None*, *rng=None*)

abstract load_image(*image_id*)

abstract image_ids()

abstract preselect(***kwargs*)

Setup a pool of training examples before the epoch begins

ndsampler.category_tree

Extends the *CategoryTree* class in the `kwcoco.category_tree` module with torch methods for computing hierarchical losses / decisions.

Notes from YOLO-9000:

- perform multiple softmax operations over co-hyponyms
- we compute the softmax over all sysnsets that are hyponyms of the same concept

synsets - sets of synonyms (word or phrase that means exactly or nearly the same as another)

hyponymn - a word of more specific meaning than a general or
superordinate term applicable to it. For example, spoon is a hyponym of cutlery.

Module Contents

Classes

CategoryTree	Wrapper that maintains flat or hierarchical category information.

class ndsampler.category_tree.**CategoryTree**(*graph=None*, *checks=True*)

Bases: `kwcoco.CategoryTree`, `Mixin_CategoryTree_Torch`

Wrapper that maintains flat or hierarchical category information.

Helps compute softmaxes and probabilities for tree-based categories where a directed edge (A, B) represents that A is a superclass of B.

Note: There are three basic properties that this object maintains:

```
node:
    Alphanumeric string names that should be generally descriptive.
    Using spaces and special characters in these names is
    discouraged, but can be done.  This is the COCO category "name"
    attribute.  For categories this may be denoted as (name, node,
    cname, catname).

id:
    The integer id of a category should ideally remain consistent.
    These are often given by a dataset (e.g. a COCO dataset).  This
    is the COCO category "id" attribute. For categories this is
    often denoted as (id, cid).

index:
    Contigous zero-based indices that indexes the list of
    categories.  These should be used for the fastest access in
    backend computation tasks. Typically corresponds to the
    ordering of the channels in the final linear layer in an
    associated model.  For categories this is often denoted as
    (index, cidx, idx, or cx).
```

Variables

- **idx_to_node** (*List[str]*) – a list of class names. Implicitly maps from index to category name.
- **id_to_node** (*Dict[int, str]*) – maps integer ids to category names
- **node_to_id** (*Dict[str, int]*) – maps category names to ids
- **node_to_idx** (*Dict[str, int]*) – maps category names to indexes
- **graph** (*networkx.Graph*) – a Graph that stores any hierarchy information. For standard mutually exclusive classes, this graph is edgeless. Nodes in this graph can maintain category attributes / properties.
- **idx_groups** (*List[List[int]]*) – groups of category indices that share the same parent category.

Example

```
>>> from kwcoco.category_tree import *
>>> graph = nx.from_dict_of_lists({
>>>     'background': [],
>>>     'foreground': ['animal'],
>>>     'animal': ['mammal', 'fish', 'insect', 'reptile'],
>>>     'mammal': ['dog', 'cat', 'human', 'zebra'],
>>>     'zebra': ['grevys', 'plains'],
>>>     'grevys': ['fred'],
>>>     'dog': ['boxer', 'beagle', 'golden'],
>>>     'cat': ['maine coon', 'persian', 'sphynx'],
>>>     'reptile': ['bearded dragon', 't-rex'],
>>> }, nx.DiGraph)
```

(continues on next page)

(continued from previous page)

```
>>> self = CategoryTree(graph)
>>> print(self)
<CategoryTree(nNodes=22, maxDepth=6, maxBreadth=4...)>
```

Example

```
>>> # The coerce classmethod is the easiest way to create an instance
>>> import kwcoco
>>> kwcoco.CategoryTree.coerce(['a', 'b', 'c'])
<CategoryTree...nNodes=3, nodes=...'a', 'b', 'c'...
>>> kwcoco.CategoryTree.coerce(4)
<CategoryTree...nNodes=4, nodes=...'class_1', 'class_2', 'class_3', ...
>>> kwcoco.CategoryTree.coerce(4)
```

ndsampler.coco_dataset

This module has moded to the kwcoco module

mkinit kwcoco

ndsampler.coco_frames

Module Contents

Classes

CocoFrames	wrapper around coco-style dataset to allow for getitem syntax

Attributes

profile

ndsampler.coco_frames.**profile**

class ndsampler.coco_frames.**CocoFrames**(*dset*, *hashid_mode='PATH'*, *workdir=None*, *verbose=0*, *backend='auto'*)

Bases: *ndsampler.abstract_frames.Frames*, *ndsampler.utils.util_misc.HashIdentifiable*

wrapper around coco-style dataset to allow for getitem syntax

CommandLine:

xdoctest -m ndsampler.coco_frames CocoFrames

Example

```
>>> from ndsampler.coco_frames import *
>>> import ndsampler
>>> import kwcoco
>>> import ubelt as ub
>>> workdir = ub.ensure_app_cache_dir('ndsampler')
>>> dset = kwcoco.CocoDataset.demo(workdir=workdir)
>>> dset._ensure_imgsize()
>>> self = CocoFrames(dset, workdir=workdir)
>>> assert self.load_image(1).shape == (512, 512, 3)
>>> assert self.load_image(1)[:-20, :-10].shape == (492, 502, 3)
>>> assert self.load_region(1, (slice(-20), slice(-10))).shape == (492, 502, 3)
```

Example

```
>>> from ndsampler import coco_sampler
>>> self = coco_sampler.CocoSampler.demo().frames
>>> assert self.load_image(1).shape == (600, 600, 3)
>>> assert self.load_image(1)[:-20, :-10].shape == (580, 590, 3)
```

property image_ids

_make_hashid()

load_region(*image_id*, *region=None*, *channels=ub.NoParam*)

Ammortized O(1) image subregion loading (assuming constant region size)

Parameters

- **image_id** (*int*) – image identifier
- **region** (*Tuple[slice, ...]*) – space-time region within an image
- **channels** (*str*) – NotImplemented
- **width** (*int*) – if the width of the entire image is know specify it
- **height** (*int*) – if the height of the entire image is know specify it

_build_pathinfo(*image_id*)

Returns

See Parent Method Docs

Example

```
>>> import ndsampler
>>> sampler1 = ndsampler.CocoSampler.demo('vidshapes5-aux')
>>> sampler2 = ndsampler.CocoSampler.demo('vidshapes5-multispectral')
>>> self = sampler1.frames
>>> pathinfo = self._build_pathinfo(1)
>>> print('pathinfo = {}'.format(ub.repr2(pathinfo, nl=3)))
```

```
>>> self = sampler2.frames
>>> pathinfo = self._build_pathinfo(1)
>>> print('pathinfo = {}'.format(ub.repr2(pathinfo, nl=3)))
```

ndsampler.coco_regions

Maintains information about groundtruth targets. Positives are specified explicitly, and negatives are mined.

The Targets class maintains the "Positive Population".

A Positive is a bounding box that belongs to an image or video with a class label and potentially other attributes. Negatives are similar except they are boxes that do not significantly intersect positives. A pool of positives can also be selected from the population such that only a subset of data is used per epoch.

Cases to Handle:

- **[ ] Annotations are significantly smaller than images**
  - Annotations are typically very far apart
  - Annotations can be clustered tightly together
  - Annotations are at massively different scales
- [ ] Annotations are about the same size as the images

Module Contents

Classes

Targets	Abstract API
CocoRegions	Converts Coco-Style datasets into a table for efficient online work

Functions

tabular_coco_targets(dset)	Transforms COCO box annotations into a tabular form
select_positive_regions(targets[, window_dims, ...])	Reduce positive example redundency by selecting disparate positive samples
new_video_sample_grid(dset[, window_dims, ...])	Create a space time-grid to sample with
new_image_sample_grid(dset, window_dims[, ...])	Create a space time-grid to sample with

Attributes

profile

ndsampler.coco_regions.**profile**

exception ndsampler.coco_regions.**MissingNegativePool**

Bases: AssertionError

Assertion failed.

class ndsampler.coco_regions.**Targets**

Bases: object

Abstract API

get_negative(*index=None*, *rng=None*)

get_positive(*index=None*, *rng=None*)

abstract overlapping_aids(*gid*, *box*)

preselect(*n_pos=None*, *n_neg=None*, *neg_to_pos_ratio=None*, *window_dims=None*, *rng=None*, *verbose=0*)

Shuffle selection of positive and negative samples

Todo: [X] Basic, window around positive annotation algorithm [ ] Sliding window algorithm from bioharn

class ndsampler.coco_regions.**CocoRegions**(*dset*, *workdir=None*, *verbose=1*)

Bases: *Targets*, *ndsampler.utils.util_misc.HashIdentifiable*, ubelt.NiceRepr

Converts Coco-Style datasets into a table for efficient on-line work

Perhaps rename this class to regions, and then have targets be an attribute of regions.

Parameters

- **dset** (*ndsampler.CocoAPI*) – a dataset in coco format
- **workdir** (*PathLike*) – a temporary directory where we can cache stuff
- **verbose** (*int*) – verbosity level

Example

```
>>> from ndsampler.coco_regions import *
>>> self = CocoRegions.demo()
>>> pos_tr = self.get_positive(rng=0)
>>> neg_tr = self.get_negative(rng=0)
>>> print(ub.repr2(pos_tr, precision=2))
>>> print(ub.repr2(neg_tr, precision=2))
```

property catgraph

property n_negatives

property n_positives

property n_samples

property class_ids

property image_ids

property n_annots

property n_images

property n_categories

lookup_class_name(*class_id*)

lookup_class_id(*class_name*)

__nice__()

classmethod demo()

_make_hashid()

property isect_index

Lazy access to a disk-cached intersection index for this dataset

_lazy_isect_index(*verbose=None*)

property targets

All viable positive annotations targets in a flat table.

The main idea is that this is the population of all positives that we could sample from. Often times we will simply use all of them.

This function takes a subset of annotations in the coco dataset that can be considered "viable" positives. We may subsample these futher, but this serves to collect the annotations that could feasibly be used by the network. Essentailly we remove annotations without bounding boxes. I'm not sure I 100% like the way this works though. Shouldn't filtering be done before we even get here? Perhaps but perhaps not. This design needs a bit more thought.

property neg_anchors

overlapping_aids(*gid*, *region*, *visible_thresh=0.0*)

Finds the other annotations in this image that overlap a region

Parameters

- **gid** (*int*) – image id
- **region** (*kwimage.Boxes*) – bounding box
- **visible_thresh** (*float*) – does not return annotations with visibility less than this threshold.

Returns
annotation ids

Return type
List[int]

get_segmentations(*aids*)

Returns the segmentations corresponding to a set of annotation ids

Example

```
>>> from ndsampler.coco_regions import *
>>> from ndsampler import coco_sampler
>>> self = coco_sampler.CocoSampler.demo().regions
>>> aids = [1, 2]
```

get_negative(*index=None*, *rng=None*)

Get localization information for a negative region

Parameters

- **index** (*int or None*) – indexes into the current negative pool or if None returns a random negative
- **rng** (*RandomState*) – used only if index is None

Returns

tr: target info dictionary

Return type

Dict

CommandLine:

xdoctest -m ndsampler.coco_regions CocoRegions.get_negative

Example

```
>>> from ndsampler.coco_regions import *
>>> from ndsampler import coco_sampler
>>> rng = kwarray.ensure_rng(0)
>>> self = coco_sampler.CocoSampler.demo().regions
>>> tr = self.get_negative(rng=rng)
>>> # xdoctest: +IGNORE_WANT
>>> assert 'category_id' in tr
>>> assert 'aid' in tr
>>> assert 'cx' in tr
>>> print(ub.repr2(tr, precision=2))
{
    'aid': -1,
    'category_id': 0,
    'cx': 190.71,
    'cy': 95.83,
    'gid': 1,
    'height': 140.00,
    'img_height': 600,
    'img_width': 600,
    'width': 68.00,
}
```

get_positive(*index=None*, *rng=None*)

Get localization information for a positive region

Parameters

- **index** (*int or None*) – indexes into the current positive pool or if None returns a random negative
- **rng** (*RandomState*) – used only if index is None

Returns
tr: target info dictionary

Return type
Dict

Example

```
>>> from ndsampler import coco_sampler
>>> rng = kwarray.ensure_rng(0)
>>> self = coco_sampler.CocoSampler.demo().regions
>>> tr = self.get_positive(0, rng=rng)
>>> print(ub.repr2(tr, precision=2))
```

get_item(*index*, *rng=None*)

Loads from positives and then negatives.

_random_negatives(*num*, *exact=False*, *neg_anchors=None*, *window_size=None*, *rng=None*, *thresh=0.0*)

Samples multiple negatives at once for efficiency

Parameters

- **num** (*int*) – number of negatives to sample
- **exact** (*bool*) – if True, we will try to find exactly *num* negatives, otherwise the number returned is approximate.
- **neg_anchors** () – prior normalized aspect ratios for negative boxes. Mutually exclusive with *window_size*.
- **window_size** (*Tuple*) – absolute box size (width, height) used to sample negative regions. If not specified the relative anchor strategy will be used to randomly choose potentially non-square regions relative to the image size.
- **thresh** (*float*) – overlap area threshold as a percentage of the negative box size. When thresh=0.0, that means negatives cannot overlap any positive, when threh=1.0, there are no constrains on negative placement.

Returns
targets - contains negative target information

Return type
DataFrameArray

Example

```
>>> from ndsampler.coco_regions import *
>>> from ndsampler import coco_sampler
>>> self = coco_sampler.CocoSampler.demo().regions
>>> num = 100
>>> rng = kwarray.ensure_rng(0)
>>> targets = self._random_negatives(num, rng=rng)
>>> assert len(targets) <= num
>>> targets = self._random_negatives(num, exact=True)
>>> assert len(targets) == num
```

new_sample_grid(*task*, *window_dims*, *window_overlap=0*, ***kwargs*)

New experimental method to replace preselect positives / negatives

Parameters

- **task** (*str*) – can be video_detection image_detection # video_classification # image_classification
- ****kwargs** – passed to *new_video_sample_grid* or *new_image_sample_grid*

Example

```
>>> from ndsampler.coco_regions import *
>>> from ndsampler import coco_sampler
>>> self = coco_sampler.CocoSampler.demo('vidshapes1').regions
>>> self.dset.conform()
>>> sample_grid = self.new_sample_grid('video_detection', window_dims=(2, 100,␣
↪100))
```

_preselect_positives(*num=None*, *window_dims=None*, *rng=None*, *verbose=None*)

” preload a bunch of positives

Example

```
>>> from ndsampler.coco_regions import *
>>> from ndsampler import coco_sampler
>>> self = coco_sampler.CocoSampler.demo().regions
>>> window_dims = (64, 64)
>>> self._preselect_positives(window_dims=window_dims, verbose=4)
```

_preselect_negatives(*num*, *window_dims=None*, *thresh=0.3*, *rng=None*, *verbose=None*)

Preselect a set of random regions to be used as negative examples.

Parameters

- **num** (*int*) – number of desired negatives to preselect. In some cases achieving this number may not be possible.
- **window_dims** (*Tuple*) – absolute dimensions (height, width) used to sample negative regions. If not specified the relative anchor strategy will be used to randomly choose potentially non-square regions relative to the image size.

- **thresh** (*float*) – overlap area threshold as a percentage of the negative box size. When thresh=0.0, that means negatives cannot overlap any positive, when threh=1.0, there are no constrains on negative placement.
- **rng** (*int | RandomState*) – random seed / state
- **verbose** (*int*) – verbosity level

Returns
number of negatives actually chosen

Return type
int

Example

```
>>> from ndsampler.coco_regions import *
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo().regions
>>> num = 100
>>> self._preselect_negatives(num, window_dims=(30, 30))
```

_cacher(*fname*, *extra_deps=None*, *disable=False*, *verbose=None*)

Create a cacher for a known lazy computation using a common hashid.

If *self.workdir* or *self.hashid* is None, then caches are disabled by default. Caches can be explicitly disabled by setting the appropriate value in the *self._enabled_caches* dictionary.

Parameters

- **fname** (*str*) – name of the property we are caching
- **extra_deps** (*OrderedDict*) – extra data to contribute to the hashid
- **disable** (*bool*) – explicitly disable cache if True, otherwise do normal checks to see if enabled.
- **verbose** (*bool, default=None*) – if specified overrides *self.verbose*.

Returns

cacher - if enabled this cacher will minimally depend
on the *self.hashid*, but may also depend on extra info.

Return type
ub.Cacher

ndsampler.coco_regions.**tabular_coco_targets**(*dset*)

Transforms COCO box annotations into a tabular form

_ = xdev.profile_now(tabular_coco_targets)(dset)

ndsampler.coco_regions.**select_positive_regions**(*targets*, *window_dims=(300, 300)*, *thresh=0.0*, *rng=None*, *verbose=0*)

Reduce positive example redundency by selecting disparate positive samples

Example

```
>>> from ndsampler.coco_regions import *
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> targets = tabular_coco_targets(dset)
>>> window_dims = (300, 300)
>>> selected = select_positive_regions(targets, window_dims)
>>> print(len(selected))
>>> print(len(dset.anns))
```

ndsampler.coco_regions.**new_video_sample_grid**(*dset*, *window_dims=None*, *window_overlap=0.0*, *space_dims=None*, *time_dim=None*, *classes_of_interest=None*, *ignore_coverage_thresh=0.6*, *negative_classes={'ignore', 'background'}*, *use_annots=True*, *legacy=True*, *verbose=1*)

Create a space time-grid to sample with

Returns

sample_grid

contains "targets", and if use_annots=True then also
contains "positives_indexes" and "negatives_indexes" indicating which annotations contain positive/negative samples.

The "positives" and "negatives" lists are deprecated and will be removed.

Return type
Dict

Example

```
>>> from ndsampler.coco_regions import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral', num_frames=5)
>>> dset.conform()
>>> window_dims = (2, 224, 224)
>>> sample_grid = new_video_sample_grid(dset, window_dims)
>>> print('sample_grid = {}'.format(ub.repr2(sample_grid, nl=2)))
>>> # Now try to load a sample
>>> tr = sample_grid['positives'][0]
>>> import ndsampler
>>> sampler = ndsampler.CocoSampler(dset)
>>> tr_ = sampler._infer_target_attributes(tr)
>>> print('tr_ = {}'.format(ub.repr2(tr_, nl=1)))
>>> sample = sampler.load_sample(tr)
>>> assert sample['im'].shape == (2, 224, 224, 5)
```

Example

```
>>> from ndsampler.coco_regions import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral', num_frames=5)
>>> dset.conform()
>>> window_dims = (2, 224, 224)
>>> sample_grid = new_video_sample_grid(dset, window_dims, use_annots=False)
```

Ignore:

import timerit ti = timerit.Timerit(10, bestof=3, verbose=2) for timer in ti.reset('vid use_annots=True'):

with timer:

new_video_sample_grid(dset, window_dims, use_annots=True, verbose=0)

for timer in ti.reset('vid use_annots=False'):

with timer:

new_video_sample_grid(dset, window_dims, use_annots=False, verbose=0)

import timerit ti = timerit.Timerit(10, bestof=3, verbose=2) for timer in ti.reset('img use_annots=True'):

with timer:

new_image_sample_grid(dset, window_dims[1:], use_annots=True, verbose=0)

for timer in ti.reset('img use_annots=False'):

with timer:

new_image_sample_grid(dset, window_dims[1:], use_annots=False, verbose=0)

Ignore:

import xdev globals().update(xdev.get_func_kwargs(new_video_sample_grid))

ndsampler.coco_regions.**new_image_sample_grid**(*dset*, *window_dims*, *window_overlap=0.0*, *classes_of_interest=None*, *ignore_coverage_thresh=0.6*, *negative_classes={'ignore', 'background'}*, *use_annots=True*, *legacy=True*, *verbose=1*)

Create a space time-grid to sample with

Returns

sample_grid

contains "targets", and if use_annots=True then also
contains "positives_indexes" and "negatives_indexes" indicating which annotations contain positive/negative samples.

The "positives" and "negatives" lists are deprecated and will be removed.

Return type
Dict

Example

```
>>> from ndsampler.coco_regions import *  # NOQA
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> dset = kwcoco.CocoDataset.demo('vidshapes8-multispectral')
>>> window_dims = (224, 224)
>>> sample_grid1 = new_image_sample_grid(dset, window_dims, use_annots=False)
>>> sample_grid = new_image_sample_grid(dset, window_dims)
>>> # Now try to load a sample
>>> idx = sample_grid['positives_indexes'][0]
>>> tr = sample_grid['targets'][idx]
>>> import ndsampler
>>> sampler = ndsampler.CocoSampler(dset)
>>> tr['channels'] = '<all>'
>>> tr_ = sampler._infer_target_attributes(tr)
>>> print('tr_ = {}'.format(ub.repr2(tr_, nl=1)))
>>> sample = sampler.load_sample(tr)
>>> assert sample['im'].shape == (224, 224, 5)
```

Ignore:
import xdev globals().update(xdev.get_func_kwargs(new_image_sample_grid))

ndsampler.coco_sampler

The CocoSampler is the ndsampler interface for efficiently sampling windowed data from a `kwcoco.CocoDataset`.

CommandLine:
xdoctest -m ndsampler.coco_sampler __doc__ –show

Example

```
>>> # Imagine you have some images
>>> import kwimage
>>> image_paths = [
>>>     kwimage.grab_test_image_fpath('astro'),
>>>     kwimage.grab_test_image_fpath('carl'),
>>>     kwimage.grab_test_image_fpath('airport'),
>>> ]  # xdoctest: +IGNORE_WANT
['~/.cache/kwimage/demodata/KXhKM72.png',
 '~/.cache/kwimage/demodata/flTHWFD.png',
 '~/.cache/kwimage/demodata/Airport.jpg']
>>> # And you want to randomly load subregions of them in O(1) time
>>> import ndsampler
>>> import kwcoco
>>> # First make a COCO dataset that refers to your images
>>> dataset = {
>>>     'images': [{'id': i, 'file_name': fpath} for i, fpath in enumerate(image_paths)],
>>>     'annotations': [],
>>>     'categories': [],
>>> }
```

(continues on next page)

(continued from previous page)

```
>>> coco_dset = kwcoco.CocoDataset(dataset)
>>> # (and possibly annotations)
>>> category_id = coco_dset.ensure_category('face')
>>> image_id = 0
>>> coco_dset.add_annotation(image_id=image_id, category_id=category_id, bbox=kwimage.
↪Boxes([[140, 10, 180, 180]], 'xywh'))
>>> print(coco_dset)
<CocoDataset(tag=None, n_anns=1, n_imgs=3, ... n_cats=1)>
>>> # Now pass the dataset to a sampler and tell it where it can store temporary files
>>> workdir = ub.ensure_app_cache_dir('ndsampler/demo')
>>> sampler = ndsampler.CocoSampler(coco_dset, workdir=workdir)
>>> # Now you can load arbirary samples by specifing a target dictionary
>>> # with an image_id (gid) center location (cx, cy) and width, height.
>>> target = {'gid': 0, 'cx': 220, 'cy': 100, 'width': 300, 'height': 300}
>>> sample = sampler.load_sample(target)
>>> # The sample contains the image data, any visible annotations, a reference
>>> # to the original target, and params of the transform used to sample this
>>> # patch
...
>>> print(sorted(sample.keys()))
['annots', 'classes', 'im', 'kp_classes', 'params', 'target', 'tr']
>>> im = sample['im']
>>> print(f'im.shape={im.shape}')
im.shape=(300, 300, 3)
>>> dets = sample['annots']['frame_dets'][0]
>>> print(f'dets={dets}')
>>> print('dets.data = {}'.format(ub.repr2(dets.data, nl=1, sv=1)))
dets=<Detections(1)>
dets.data = {
    'aids': [1],
    'boxes': <Boxes(xywh, array([[ 70.,  60., 180., 180.]]))>,
    'cids': [1],
    'keypoints': <PointsList(n=1)>,
    'segmentations': <SegmentationList(n=1)>,
}
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(im)
>>> dets.draw(labels=False)
>>> kwplot.show_if_requested()
>>> # The load sample function is at the core of what ndsampler does
>>> # There are other helper functions like load_positive / load_negative
>>> # which deal with annotations. See those for more details.
>>> # For random negative sampling see coco_regions.
```

Module Contents

Classes

CocoSampler	Samples patches of positives and negative detection windows from a COCO

Functions

_center_extent_to_slice(center, window_dims)	Transforms a center and window dimensions into a start/stop slice
_ensure_iterablen(scalar, n)	
_coerce_pad(pad, ndims)	

Attributes

profile

ndsampler.coco_sampler.**profile**

class ndsampler.coco_sampler.**CocoSampler**(*dset*, *workdir=None*, *autoinit=True*, *backend=None*, *verbose=0*)

Bases: *ndsampler.abstract_sampler.AbstractSampler*, *ndsampler.utils.util_misc.HashIdentifiable*, ubelt.NiceRepr

Samples patches of positives and negative detection windows from a COCO dataset. Can be used for training FCN or RPN based classifiers / detectors.

Does data loading, padding, etc…

Parameters

- **dset** (*kwcoco.CocoDataset*) – a coco-formatted dataset
- **backend** (*str | Dict*) – either 'cog' or 'npy', or a dict with *{'type': str, 'config': Dict}*. See AbstractFrames for more details. Defaults to None, which does not do anything fancy.

Example

#print >>> from ndsampler.coco_sampler import * >>> self = CocoSampler.demo('photos') … >>> print(sorted(self.class_ids)) [0, 1, 2, 3, 4, 5, 6, 7, 8] >>> print(self.n_positives) 4

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo('photos')
>>> p_sample = self.load_positive()
>>> n_sample = self.load_negative()
>>> self = ndsampler.CocoSampler.demo('shapes')
>>> p_sample2 = self.load_positive()
>>> n_sample2 = self.load_negative()
>>> for sample in [p_sample, n_sample, p_sample2, n_sample2]:
>>>     assert 'annots' in sample
>>>     assert 'im' in sample
>>>     assert 'rel_boxes' in sample['annots']
>>>     assert 'rel_ssegs' in sample['annots']
>>>     assert 'rel_kpts' in sample['annots']
>>>     assert 'cids' in sample['annots']
>>>     assert 'aids' in sample['annots']
```

classmethod demo(*key='shapes'*, *workdir=None*, *backend=None*, ***kw*)

Create a toy coco sampler for testing and demo puposes

SeeAlso:

- kwcoco.CocoDataset.demo

_init()

property classes

property catgraph

DEPRICATED, use self.classes instead

_depends()

lookup_class_name(*class_id*)

lookup_class_id(*class_name*)

property n_positives

property n_annots

property n_samples

__len__()

property n_images

property n_categories

property class_ids

property image_ids

preselect(***kwargs*)

Setup a pool of training examples before the epoch begins

new_sample_grid(*task*, *window_dims*, *window_overlap=0*)

load_image_with_annots(*image_id*, *cache=True*)

Parameters

- **image_id** (*int*) – the coco image id
- **cache** (*bool, default=True*) – if True returns the fast subregion-indexable file reference. Otherwise, eagerly loads the entire image.

Returns
img: the coco image dict augmented with imdata anns: the coco annotations in this image

Return type
Tuple[Dict, List[Dict]]

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> img, anns = self.load_image_with_annots(1)
>>> dets = kwimage.Detections.from_coco_annots(anns, dset=self.dset)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'][:], doclf=1)
>>> dets.draw()
>>> kwplot.show_if_requested()
```

load_annotations(*image_id*)

Loads the annotations within an image

Parameters
image_id (*int*) – the coco image id

Returns
list of coco annotation dictionaries

Return type
List[Dict]

load_image(*image_id*, *cache=True*)

Loads the annotations within an image

Parameters

- **image_id** (*int*) – the coco image id
- **cache** (*bool, default=True*) – if True returns the fast subregion-indexable file reference. Otherwise, eagerly loads the entire image.

Returns
either ndarray data or a indexable reference

Return type
ArrayLike

load_item(*index*, *with_annots=True*, *target=None*, *rng=None*, ***kw*)

Loads item from either positive or negative regions pool.

Lower indexes will return positive regions and higher indexes will return negative regions.

The main paradigm of the sampler is that sampler.regions maintains a pool of target regions, you can influence what that pool is at any point by calling sampler.regions.preselect (usually either at the start of learning, or maybe after every epoch, etc..), and you use load_item to load the index-th item from that preselected pool. Depending on how you preselected the pool, the returned item might correspond to a positive or negative region.

Parameters

- **index** (*int*) – index of target region
- **with_annots** (*bool | str, default=True*) – if True, also extracts information about any annotation that overlaps the region of interest (subject to visibility_thresh). Can also be a List[str] that specifies which specific subinfo should be extracted. Valid strings in this list are: boxes, keypoints, and segmenation.
- **target** (*Dict*) – Extra target arguments that update the positive target, like window_dims, pad, etc.... See `load_sample()` for details on allowed keywords.
- **rng** (*None | int | RandomState*) – a seed or seeded random number generator.
- ****kw** – other arguments that can be passed to `CocoSampler.load_sample()`

Returns

sample: dict containing keys
im (ndarray): image data target (dict): contains the same input items as the input

target but additionally specifies inferred information like rel_cx and rel_cy, which gives the center of the target w.r.t the returned **padded** sample.

annots (dict): Dict of aids, cids, and rel/abs boxes

Return type
Dict

load_positive(*index=None*, *with_annots=True*, *target=None*, *rng=None*, ***kw*)

Load an item from the the positive pool of regions.

Parameters

- **index** (*int*) – index of positive target
- **with_annots** (*bool | str, default=True*) – if True, also extracts information about any annotation that overlaps the region of interest (subject to visibility_thresh). Can also be a List[str] that specifies which specific subinfo should be extracted. Valid strings in this list are: boxes, keypoints, and segmentation.
- **target** (*Dict*) – Extra target arguments that update the positive target, like window_dims, pad, etc.... See `load_sample()` for details on allowed keywords.
- **rng** (*None | int | RandomState*) – a seed or seeded random number generator.
- ****kw** – other arguments that can be passed to `CocoSampler.load_sample()`

Returns

sample: dict containing keys
im (ndarray): image data tr (dict): contains the same input items as tr but additionally

specifies rel_cx and rel_cy, which gives the center of the target w.r.t the returned **padded** sample.

annots (dict): Dict of aids, cids, and rel/abs boxes

Return type
Dict

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> sample = self.load_positive(pad=(10, 10), tr=dict(window_dims=(3, 3)))
>>> assert sample['im'].shape[0] == 23
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(sample['im'], doclf=1)
>>> kwplot.show_if_requested()
```

load_negative(*index=None*, *with_annots=True*, *target=None*, *rng=None*, ***kw*)

Load an item from the the negative pool of regions.

Parameters

- **index** (*int*) – if specified loads a specific negative from the presampled pool, otherwise the next negative in the pool is returned.
- **with_annots** (*bool | str, default=True*) – if True, also extracts information about any annotation that overlaps the region of interest (subject to visibility_thresh). Can also be a List[str] that specifies which specific subinfo should be extracted. Valid strings in this list are: boxes, keypoints, and segmentation.
- **target** (*Dict*) – Extra target arguments that update the positive target, like window_dims, pad, etc.... See *load_sample()* for details on allowed keywords.
- **rng** (*None | int | RandomState*) – a seed or seeded random number generator.

Returns

sample: dict containing keys
im (ndarray): image data tr (dict): contains the same input items as tr but additionally

specifies rel_cx and rel_cy, which gives the center of the target w.r.t the returned **padded** sample.

annots (dict): Dict of aids, cids, and rel/abs boxes

Return type
Dict

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> rng = None
>>> sample = self.load_negative(rng=rng, pad=(0, 0))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> import kwimage
>>> kwplot.autompl()
```

(continues on next page)

(continued from previous page)

```
>>> abs_sample_box = sample['params']['sample_tlbr']
>>> tf_rel_from_abs = kwimage.Affine.coerce(sample['params']['tf_rel_to_abs']).
↪inv()
>>> wh, ww = sample['target']['window_dims']
>>> abs_window_box = kwimage.Boxes([[sample['target']['cx'], sample['target'][
↪'cy'], ww, wh]], 'cxywh')
>>> rel_window_box = abs_window_box.warp(tf_rel_from_abs)
>>> rel_sample_box = abs_sample_box.warp(tf_rel_from_abs)
>>> kwplot.imshow(sample['im'], fnum=1, doclf=True)
>>> rel_sample_box.draw(color='kw_green', lw=10)
>>> rel_window_box.draw(color='kw_blue', lw=8)
>>> kwplot.show_if_requested()
```

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> rng = None
>>> sample = self.load_negative(rng=rng, pad=(10, 20), target=dict(window_
↪dims=(64, 64)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> import kwimage
>>> kwplot.autompl()
>>> abs_sample_box = sample['params']['sample_tlbr']
>>> tf_rel_from_abs = kwimage.Affine.coerce(sample['params']['tf_rel_to_abs']).
↪inv()
>>> wh, ww = sample['target']['window_dims']
>>> abs_window_box = kwimage.Boxes([[sample['target']['cx'], sample['target'][
↪'cy'], ww, wh]], 'cxywh')
>>> rel_window_box = abs_window_box.warp(tf_rel_from_abs)
>>> rel_sample_box = abs_sample_box.warp(tf_rel_from_abs)
>>> kwplot.imshow(sample['im'], fnum=1, doclf=True)
>>> rel_sample_box.draw(color='kw_green', lw=10)
>>> rel_window_box.draw(color='kw_blue', lw=8)
>>> kwplot.show_if_requested()
```

load_sample(*target=None*, *with_annots=True*, *visible_thresh=0.0*, ***kwargs*)

Loads the volume data associated with the bbox and frame of a target

Parameters

- **target** (*dict*) – target dictionary (often abbreviated as tr) indicating an nd source object (e.g. image or video) and the coordinate region to sample from. Unspecified coordinate regions default to the extent of the source object.

  For 2D image source objects, target must contain or be able to infer the key *gid (int)*, to specify an image id.

  For 3D video source objects, target must contain the key *vidid (int)*, to specify a video id (NEW in 0.6.1) or *gids List[int]*, as a list of images in a video (NEW in 0.6.2)

  In general, coordinate regions can specified by the key *slices*, a numpy-like "fancy

index" over each of the n dimensions. Usually this is a tuple of slices, e.g. (y1:y2, x1:x2) for images and (t1:t2, y1:y2, x1:x2) for videos.

You may also specify: *space_slice* as (y1:y2, x1:x2) for both 2D images and 3D videos and *time_slice* as t1:t2 for 3D videos.

Spatial regions can be specified with keys:

- 'cx' and 'cy' as the center of the region in pixels.
- 'width' and 'height' are in pixels.
- 'window_dims' is a height, width tuple or can be a

special string key 'square', which overrides width and height to both be the maximum of the two.

Temporal regions are specifiable by *slices*, *time_slice* or an explicit list of *gids*.

The *aid* key can be specified to indicate a specific annotation to load. This uses the annotation information to infer 'gid', 'cx', 'cy', 'width', and 'height' if they are not present. (NEW in 0.5.10)

The *channels* key can be specified as a channel code or
`kwcoco.ChannelSpec` object. (NEW in 0.6.1)

as_xarray (bool, default=False):
if True, return the image data as an xarray object

interpolation (str, default='auto'):
type of resample interpolation

antialias (str, default='auto'):
antialias sample or not

nodata: override function level nodata

use_native_scale (bool): If True, the "im" field is returned
as a jagged list of data that are as close to native resolution as possible while still maintaining alignment up to a scale factor. Currently only available for video sampling.

scale (float | Tuple[float, float]):
if specified, add an extra scale factor to the data.

pad (tuple): (height, width) extra context to add to window dims.
This helps prevent augmentation from producing boundary effects

padkw (dict): kwargs for *numpy.pad*.
Defaults to {'mode': 'constant'}.

dtype (type | None):
Cast the loaded data to this type. If unspecified returns the data as-is.

nodata (int | None, default=None):
If specified, for integer data with nodata values, this is passed to kwcoco delayed image finalize. The data is converted to float32 and nodata values are replaced with nan. These nan values are handled correctly in subsequent warping operations.

- **with_annots** (*bool | str, default=True*) – if True, also extracts information about any annotation that overlaps the region of interest (subject to visibility_thresh). Can also be a List[str] that specifies which specific subinfo should be extracted. Valid strings in this list are: boxes, keypoints, and segmentation.

- **visible_thresh** (*float*) – does not return annotations with visibility less than this threshold.
- ****kwargs** – handles deprecated arguments which are now specified in the target dictionary itself.

Returns

sample: dict containing keys
im (ndarray | DataArray): image / video data target (dict): contains the same input items as the input

target but additionally specifies inferred information like rel_cx and rel_cy, which gives the center of the target w.r.t the returned **padded** sample.

annots (dict): containing items:

frame_dets (List[kwimage.Detections]): a list of detection
objects containing the requested annotation info for each frame.

aids (list): annotation ids DEPRECATED cids (list): category ids DEPRECATED rel_ssegs (ndarray): segmentations relative to the sample DEPRECATED rel_kpts (ndarray): keypoints relative to the sample DEPRECATED

Return type
Dict

CommandLine:
xdoctest -m ndsampler.coco_sampler CocoSampler.load_sample:2 –show

xdoctest -m ndsampler.coco_sampler CocoSampler.load_sample:1 –show xdoctest -m ndsampler.coco_sampler CocoSampler.load_sample:3 –show

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> # The target (target) lets you specify an arbitrary window
>>> target = {'gid': 1, 'cx': 5, 'cy': 2, 'width': 6, 'height': 6}
>>> sample = self.load_sample(target)
...
>>> print('sample.shape = {!r}'.format(sample['im'].shape))
sample.shape = (6, 6, 3)
```

Example

```
>>> # Access direct annotation information
>>> import ndsampler
>>> sampler = ndsampler.CocoSampler.demo()
>>> # Sample a region that contains at least one annotation
>>> target = {'gid': 1, 'cx': 5, 'cy': 2, 'width': 600, 'height': 600}
>>> sample = sampler.load_sample(target)
>>> annotation_ids = sample['annots']['aids']
>>> aid = annotation_ids[0]
```

(continues on next page)

(continued from previous page)

```
>>> # Method1: Access ann dict directly via the coco index
>>> ann = sampler.dset.anns[aid]
>>> # Method2: Access ann objects via annots method
>>> dets = sampler.dset.annots(annotation_ids).detections
>>> print('dets.data = {}'.format(ub.repr2(dets.data, nl=1)))
```

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> target = self.regions.get_positive(0)
>>> target['window_dims'] = 'square'
>>> target['pad'] = (25, 25)
>>> sample = self.load_sample(target)
>>> print('im.shape = {!r}'.format(sample['im'].shape))
im.shape = (135, 135, 3)
>>> target['window_dims'] = None
>>> target['pad'] = (0, 0)
>>> sample = self.load_sample(target)
>>> print('im.shape = {!r}'.format(sample['im'].shape))
im.shape = (52, 85, 3)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(sample['im'])
>>> kwplot.show_if_requested()
```

Example

```
>>> # sample an out of bounds target
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> target = self.regions.get_positive(0)
>>> target['window_dims'] = (364, 364)
>>> sample = self.load_sample(target)
>>> annots = sample['annots']
>>> assert len(annots['aids']) > 0
>>> #assert len(annots['rel_cxywh']) == len(annots['aids'])
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> abs_frame = self.frames.load_image(sample['target']['gid'])[:]
>>> tf_rel_to_abs = sample['params']['tf_rel_to_abs']
>>> abs_boxes = annots['rel_boxes'].warp(tf_rel_to_abs)
>>> abs_ssegs = annots['rel_ssegs'].warp(tf_rel_to_abs)
>>> abs_kpts = annots['rel_kpts'].warp(tf_rel_to_abs)
>>> # Draw box in original image context
>>> kwplot.imshow(abs_frame, pnum=(1, 2, 1), fnum=1)
>>> abs_boxes.translate([-.5, -.5]).draw()
```

(continues on next page)

(continued from previous page)

```
>>> abs_kpts.draw(color='green', radius=10)
>>> abs_ssegs.draw(color='red', alpha=.5)
>>> # Draw box in relative sample context
>>> kwplot.imshow(sample['im'], pnum=(1, 2, 2), fnum=1)
>>> annots['rel_boxes'].translate([-.5, -.5]).draw()
>>> annots['rel_ssegs'].draw(color='red', alpha=.6)
>>> annots['rel_kpts'].draw(color='green', alpha=.4, radius=10)
>>> kwplot.show_if_requested()
```

Example

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('photos')
>>> target = self.regions.get_positive(1)
>>> target['window_dims'] = (300, 150)
>>> target['pad'] = None
>>> sample = self.load_sample(target)
>>> assert sample['im'].shape[0:2] == target['window_dims']
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(sample['im'], colorspace='rgb')
>>> kwplot.show_if_requested()
```

Example

```
>>> # Multispectral video sample example
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('vidshapes1-multispectral', num_frames=5)
>>> sample_grid = self.new_sample_grid('video_detection', (3, 128, 128))
>>> target = sample_grid['positives'][0]
>>> target['channels'] = 'B1|B8'
>>> target['as_xarray'] = False
>>> sample = self.load_sample(target)
>>> print(ub.repr2(sample['target'], nl=1))
>>> print(sample['im'].shape)
>>> assert sample['im'].shape == (3, 128, 128, 2)
>>> target['channels'] = '<all>'
>>> sample = self.load_sample(target)
>>> assert sample['im'].shape == (3, 128, 128, 5)
```

Example

```
>>> # Multispectral-multisensor jagged video sample example
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('vidshapes1-msi-multisensor', num_frames=5)
>>> sample_grid = self.new_sample_grid('video_detection', (3, 128, 128))
>>> target = sample_grid['positives'][0]
>>> target['channels'] = 'B1|B8'
>>> target['as_xarray'] = False
>>> sample1 = self.load_sample(target)
>>> target['scale'] = 2
>>> sample2 = self.load_sample(target)
>>> target['use_native_scale'] = True
>>> sample3 = self.load_sample(target)
>>> ####
>>> assert sample1['im'].shape == (3, 128, 128, 2)
>>> assert sample2['im'].shape == (3, 256, 256, 2)
>>> box1 = sample1['annots']['frame_dets'][0].boxes
>>> box2 = sample2['annots']['frame_dets'][0].boxes
>>> box3 = sample3['annots']['frame_dets'][0].boxes
>>> assert np.allclose((box2.width / box1.width), 2)
>>> # Jagged annotations are still in video space
>>> assert np.allclose((box3.width / box1.width), 2)
>>> jagged_shape = [[p.shape for p in f] for f in sample3['im']]
>>> jagged_align = [[a for a in m['align']] for m in sample3['params']['jagged_
↪meta']]
```

_infer_target_attributes(*target*, ***kwargs*)

Infer unpopulated target attribues

Example

```
>>> # sample using only an annotation id
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> target = {'aid': 1, 'as_xarray': True}
>>> target_ = self._infer_target_attributes(target)
>>> print('target_ = {}'.format(ub.repr2(target_, nl=1)))
>>> assert target_['gid'] == 1
>>> assert all(k in target_ for k in ['cx', 'cy', 'width', 'height'])
```

```
>>> self = CocoSampler.demo('vidshapes8-multispectral')
>>> target = {'aid': 1, 'as_xarray': True}
>>> target_ = self._infer_target_attributes(target)
>>> assert target_['gid'] == 1
>>> assert all(k in target_ for k in ['cx', 'cy', 'width', 'height'])
```

```
>>> target = {'vidid': 1, 'as_xarray': True}
>>> target_ = self._infer_target_attributes(target)
>>> print('target_ = {}'.format(ub.repr2(target_, nl=1)))
>>> assert 'gids' in target_
```

```
>>> target = {'gids': [1, 2], 'as_xarray': True}
>>> target_ = self._infer_target_attributes(target)
>>> print('target_ = {}'.format(ub.repr2(target_, nl=1)))
```

_load_slice(*target*)

Example

```
>>> # sample an out of bounds target
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> target = self.regions.get_positive(0)
>>> target = self._infer_target_attributes(target)
>>> target['as_xarray'] = True
>>> sample = self._load_slice(target)
>>> print('sample = {!r}'.format(ub.map_vals(type, sample)))
```

```
>>> # sample an out of bounds target
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('vidshapes2')
>>> target = self._infer_target_attributes({'vidid': 1})
>>> target = self._infer_target_attributes(target)
>>> target['as_xarray'] = True
>>> sample = self._load_slice(target)
>>> print('sample = {!r}'.format(ub.map_vals(type, sample)))
```

```
>>> target = self._infer_target_attributes({'gids': [1, 2, 3]})
>>> target['as_xarray'] = True
>>> sample = self._load_slice(target)
>>> print('sample = {!r}'.format(ub.map_vals(type, sample)))
```

CommandLine:

xdoctest -m ndsampler.coco_sampler CocoSampler._load_slice –profile

Ignore:

from ndsampler.coco_sampler import * # NOQA from ndsampler.coco_sampler import _center_extent_to_slice, _ensure_iterablen import ndsampler import xdev globals().update(xdev.get_func_kwargs(ndsampler.CocoSampler._load_slice))

Example

```
>>> # Multispectral video sample example
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('vidshapes1-multispectral', num_frames=5)
>>> sample_grid = self.new_sample_grid('video_detection', (3, 128, 128))
>>> target = sample_grid['positives'][0]
>>> target = self._infer_target_attributes(target)
>>> target['channels'] = 'B1|B8'
>>> target['as_xarray'] = False
>>> sample = self.load_sample(target)
```

(continues on next page)

(continued from previous page)

```
>>> print(ub.repr2(sample['target'], nl=1))
>>> print(sample['im'].shape)
>>> assert sample['im'].shape == (3, 128, 128, 2)
>>> target['channels'] = '<all>'
>>> sample = self.load_sample(target)
>>> assert sample['im'].shape == (3, 128, 128, 5)
```

Example

```
>>> # Multispectral video sample example
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('vidshapes1-multisensor-msi', num_frames=5)
>>> sample_grid = self.new_sample_grid('video_detection', (3, 128, 128))
>>> target = sample_grid['positives'][0]
>>> target = self._infer_target_attributes(target)
>>> target['channels'] = 'B1|B8'
>>> target['as_xarray'] = False
>>> target['space_slice'] = (slice(-64, 64), slice(-64, 64))
>>> sample = self.load_sample(target)
>>> print(ub.repr2(sample['target'], nl=1))
>>> print(sample['im'].shape)
>>> assert sample['im'].shape == (3, 128, 128, 2)
>>> target['channels'] = '<all>'
>>> sample = self.load_sample(target)
>>> assert sample['im'].shape[2] > 5  # probably 16
```

```
>>> # Test jagged native scale sampling
>>> target['use_native_scale'] = True
>>> target['as_xarray'] = True
>>> target['channels'] = 'B1|B8|r|g|b|disparity|gauss'
>>> sample = self.load_sample(target)
>>> jagged_meta = sample['params']['jagged_meta']
>>> frames = sample['im']
>>> jagged_shape = [[p.shape for p in f] for f in frames]
>>> jagged_chans = [[p.coords['c'].values.tolist() for p in f] for f in frames]
>>> jagged_chans2 = [m['chans'] for m in jagged_meta]
>>> jagged_align = [[a.concise() for a in m['align']] for m in jagged_meta]
>>> # all frames should have the same number of channels
>>> assert len(frames) == 3
>>> assert all(sum(p.shape[2] for p in f) == 7 for f in frames)
>>> frames[0] == 3
>>> print('jagged_chans = {}'.format(ub.repr2(jagged_chans, nl=1)))
>>> print('jagged_shape = {}'.format(ub.repr2(jagged_shape, nl=1)))
>>> print('jagged_chans2 = {}'.format(ub.repr2(jagged_chans2, nl=1)))
>>> print('jagged_align = {}'.format(ub.repr2(jagged_align, nl=1)))
```

```
>>> # Test realigned native scale sampling
>>> target['use_native_scale'] = True
>>> target['realign_native'] = 'largest'
>>> target['as_xarray'] = True
```

(continues on next page)

(continued from previous page)

```
>>> gid = None
>>> for coco_img in self.dset.images().coco_images:
>>>     if coco_img.channels & 'r|g|b':
>>>         gid = coco_img.img['id']
>>>         break
>>> assert gid is not None, 'need specific image'
>>> target['gids'] = [gid]
>>> # Test channels that are good early fused groups
>>> target['channels'] = 'r|g|b'
>>> sample1 = self.load_sample(target)
>>> target['channels'] = 'B8|B11'
>>> sample2 = self.load_sample(target)
>>> target['channels'] = 'r|g|b|B11'
>>> sample3 = self.load_sample(target)
>>> shape1 = sample1['im'].shape[1:3]
>>> shape2 = sample2['im'].shape[1:3]
>>> shape3 = sample3['im'].shape[1:3]
>>> print(f'shape1={shape1}')
>>> print(f'shape2={shape2}')
>>> print(f'shape3={shape3}')
>>> assert shape1 != shape2
>>> assert shape2 == shape3
```

_load_slice_3d(*target*)

Breakout the 2d vs 3d logic so they can evolve somewhat independently.

TODO: the 2D logic needs to be updated to be more consistent with 3d logic

Or at least the differences between them are more clear.

_load_slice_2d(*target*)

Breakout the 2d vs 3d logic so they can evolve somewhat independently.

TODO: the 2D logic needs to be updated to be more consistent with 3d logic

Or at least the differences between them are more clear.

_populate_overlap(*sample*, *visible_thresh=0.1*, *with_annots=True*)

Add information about annotations overlapping the sample.

with_annots can be a + separated string or list of the the special keys:
'segmentation' and 'keypoints'.

Example

```
>>> # sample an out of bounds target
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> target = self.regions.get_item(0)
>>> target = self._infer_target_attributes(target)
>>> sample = self._load_slice(target)
>>> sample = self._populate_overlap(sample)
>>> print('sample = {}'.format(ub.repr2(ub.util_dict.dict_diff(sample, ['im']),
↪ nl=-1)))
```

ndsampler.coco_sampler.**_center_extent_to_slice**(*center*, *window_dims*)

Transforms a center and window dimensions into a start/stop slice

Parameters

- **center** (*Tuple[float]*) – center location (cy, cx)
- **window_dims** (*Tuple[int]*) – window size (height, width)

Returns

the slice corresponding to the centered window

Return type

Tuple[slice, ...]

Example

```
>>> center = (2, 5)
>>> window_dims = (6, 6)
>>> slices = _center_extent_to_slice(center, window_dims)
>>> assert slices == (slice(-1, 5), slice(2, 8))
```

Example:

```
>>> center = (2, 5)
>>> window_dims = (64, 64)
>>> slices = _center_extent_to_slice(center, window_dims)
>>> assert slices == (slice(-30, 34, None), slice(-27, 37, None))
```

Example

```
>>> # Test floating point error case
>>> center = (500.5, 974.9999999999999)
>>> window_dims  = (100, 100)
>>> slices = _center_extent_to_slice(center, window_dims)
>>> assert slices == (slice(450, 550, None), slice(924, 1024, None))
```

ndsampler.coco_sampler.**_ensure_iterablen**(*scalar*, *n*)

ndsampler.coco_sampler.**_coerce_pad**(*pad*, *ndims*)

ndsampler.coerce_data

Moved to netharn

Module Contents

Functions

coerce_datasets(config[, build_hashid, verbose])	Coerce train / val / test datasets from standard netharn config keys
_print_catfreq_columns(subsets)	
_catfreq_columns_str(subsets)	
_split_train_vali_test(coco_dset[, factor])	**Parameters** **factor** (*int*) -- number of pieces to divide images into

ndsampler.coerce_data.**coerce_datasets**(*config*, *build_hashid=False*, *verbose=1*)

Coerce train / val / test datasets from standard netharn config keys

Todo:

- Does this belong in netharn?

This only looks at the following keys in config:

- datasets
- train_dataset
- vali_dataset
- test_dataset

Example

```
>>> import kwcoco
>>> import ndsampler.coerce_data
>>> config = {'datasets': 'special:shapes'}
>>> print('config = {!r}'.format(config))
>>> dsets = ndsampler.coerce_data.coerce_datasets(config)
>>> print('dsets = {!r}'.format(dsets))
```

```
>>> config = {'datasets': 'special:shapes256'}
>>> ndsampler.coerce_data.coerce_datasets(config)
```

```
>>> config = {
>>>     'datasets': kwcoco.CocoDataset.demo('shapes'),
>>> }
>>> coerce_datasets(config)
>>> coerce_datasets({
>>>     'datasets': kwcoco.CocoDataset.demo('shapes'),
```

(continues on next page)

(continued from previous page)

```
>>>     'test_dataset': kwcoco.CocoDataset.demo('photos'),
>>> })
>>> coerce_datasets({
>>>     'datasets': kwcoco.CocoDataset.demo('shapes'),
>>>     'test_dataset': kwcoco.CocoDataset.demo('photos'),
>>> })
```

ndsampler.coerce_data.**_print_catfreq_columns**(*subsets*)

ndsampler.coerce_data.**_catfreq_columns_str**(*subsets*)

ndsampler.coerce_data.**_split_train_vali_test**(*coco_dset*, *factor=3*)

Parameters

factor (*int*) – number of pieces to divide images into

CommandLine:

xdoctest -m /home/joncrall/code/ndsampler/ndsampler/coerce_data.py _split_train_vali_test

Example

```
>>> from ndsampler.coerce_data import _split_train_vali_test
>>> import kwcoco
>>> coco_dset = kwcoco.CocoDataset.demo('shapes8')
>>> split_gids = _split_train_vali_test(coco_dset)
>>> print('split_gids = {}'.format(ub.repr2(split_gids, nl=1)))
```

ndsampler.frame_cache

Tools for caching intermediate frame representations.

Module Contents

Functions

_cog_cache_write(gpath, cache_gpath[, config])	CommandLine:
_npy_cache_write(gpath, cache_gpath[, config])	
_locked_cache_write(_write_func, gpath, cache_gpath[, ...])	Ensures that mem_gpath exists in a multiprocessing-safe way
_lookup_dvc_hash(path)	proof of concept
_ensure_image_npy(gpath, cache_gpath)	Ensures that cache_gpath exists in a multiprocessing-safe way
_ensure_image_cog(gpath, cache_gpath, config[, ...])	Returns a special array-like object with a COG GeoTIFF backend

Attributes

DEBUG_COG_ATOMIC_WRITE
DEBUG_FILE_LOCK_CACHE_WRITE
RUN_COG_CORRUPTION_CHECKS
DEBUG_LOAD_COG

ndsampler.frame_cache.**DEBUG_COG_ATOMIC_WRITE = 0**

ndsampler.frame_cache.**DEBUG_FILE_LOCK_CACHE_WRITE = 0**

ndsampler.frame_cache.**RUN_COG_CORRUPTION_CHECKS = True**

ndsampler.frame_cache.**DEBUG_LOAD_COG**

exception ndsampler.frame_cache.**CorruptCOG**

Bases: Exception

Common base class for all non-exit exceptions.

ndsampler.frame_cache.**_cog_cache_write**(*gpath*, *cache_gpath*, *config=None*)

CommandLine:

xdoctest -m ndsampler.abstract_frames _cog_cache_write

Example

```
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> import ndsampler
>>> from ndsampler.abstract_frames import *
>>> import kwcoco
>>> workdir = ub.ensure_app_cache_dir('ndsampler')
>>> dset = kwcoco.CocoDataset.demo()
>>> imgs = dset.images()
>>> id_to_name = imgs.lookup('file_name', keepid=True)
>>> id_to_path = {gid: join(dset.img_root, name)
>>>               for gid, name in id_to_name.items()}
>>> self = SimpleFrames(id_to_path, workdir=workdir)
>>> image_id = ub.peek(id_to_name)
>>> #gpath = self._lookup_gpath(image_id)
```

EXIT # >>> hashid = self._lookup_hashid(image_id) # >>> cog_gname = '{}_{}.cog.tiff'.format(image_id, hashid) # >>> cache_gpath = cog_gpath = join(self.cache_dpath, cog_gname) # >>> _cog_cache_write(gpath, cache_gpath, {})

ndsampler.frame_cache.**_npy_cache_write**(*gpath*, *cache_gpath*, *config=None*)

ndsampler.frame_cache.**_locked_cache_write**(*_write_func*, *gpath*, *cache_gpath*, *config=None*)

Ensures that mem_gpath exists in a multiprocessing-safe way

ndsampler.frame_cache.**_lookup_dvc_hash**(*path*)

proof of concept

Ignore:

path = '/home/joncrall/data/dvc-repos/viame_dvc/public/Benthic/US_NE_2017_CFF_HABCAM/annotations_flatfish.kwco
_lookup_dvc_hash(path) path = '/home/joncrall/data/dvc-repos/viame_dvc/public/Benthic/US_NE_2017_CFF_HABCAM/
_lookup_dvc_hash(path)

ndsampler.frame_cache.**_ensure_image_npy**(*gpath*, *cache_gpath*)

Ensures that cache_gpath exists in a multiprocessing-safe way

Returns a memmapped reference to the entire image

ndsampler.frame_cache.**_ensure_image_cog**(*gpath*, *cache_gpath*, *config*, *hack_use_cli=True*)

Returns a special array-like object with a COG GeoTIFF backend

ndsampler.isect_indexer

Module Contents

Classes

FrameIntersectionIndex	Build spatial tree for each frame so we can quickly determine if a random

Attributes

profile

ndsampler.isect_indexer.**profile**

class ndsampler.isect_indexer.**FrameIntersectionIndex**

Bases: ubelt.NiceRepr

Build spatial tree for each frame so we can quickly determine if a random negative is too close to a positive. For each frame/image we built a qtree.

Example

```
>>> from ndsampler.isect_indexer import *
>>> import kwcoco
>>> import ubelt as ub
>>> dset = kwcoco.CocoDataset.demo()
>>> dset._ensure_imgsize()
>>> dset.remove_annotations([ann for ann in dset.anns.values()
>>>                          if 'bbox' not in ann])
>>> # Build intersection index aroung coco dataset
>>> self = FrameIntersectionIndex.from_coco(dset)
>>> gid = 1
```

(continues on next page)

(continued from previous page)

```
>>> box = kwimage.Boxes([0, 10, 100, 100], 'xywh')
>>> isect_aids, ious = self.ious(gid, box)
>>> print(ub.repr2(ious.tolist(), nl=0, precision=4))
[0.0507]
```

__nice__()

classmethod from_coco(*dset*, *verbose=0*)

Parameters
dset (*kwcoco.CocoDataset*) – positive annotation data

Returns
FrameIntersectionIndex

classmethod demo(**args*, ***kwargs*)

Create a demo intersection index.

Parameters

- ***args** – see kwcoco.CocoDataset.demo
- ****kwargs** – see kwcoco.CocoDataset.demo

Returns
FrameIntersectionIndex

static _build_index(*dset*, *verbose=0*)

overlapping_aids(*gid*, *box*)

Find all annotation-ids within an image that have some overlap with a bounding box.

Parameters

- **gid** (*int*) – an image id
- **box** (*kwimage.Boxes*) – the specified region

Returns
list of annotation ids

Return type
List[int]

Example

```
>>> self = FrameIntersectionIndex.demo('shapes128')
>>> for gid, qtree in self.qtrees.items():
>>>     box = kwimage.Boxes([0, 0, qtree.width, qtree.height], 'xywh')
>>>     self.overlapping_aids(gid, box)
```

ious(*gid*, *box*)

Find overlaping annotations in a specific image and their intersection over union with a a query box.

Parameters

- **gid** (*int*) – an image id
- **box** (*kwimage.Boxes*) – the specified region

Returns
isect_aids: list of annotation ids ious: jaccard score for each returned annotation id

Return type
Tuple[List[int], ndarray]

iooas(*gid*, *box*)

Intersection over other's area

Parameters

- **gid** (*int*) – an image id
- **box** (*kwimage.Boxes*) – the specified region

Like iou, but non-symetric, returned number is a percentage of the other's (groundtruth) area. This means we dont care how big the (negative) *box* is.

random_negatives(*num*, *anchors=None*, *window_size=None*, *gids=None*, *thresh=0.0*, *exact=True*, *rng=None*, *patience=None*)

Finds random boxes that don't have a large overlap with positive instances.

Parameters

- **num** (*int*) – number of negative boxes to generate (actual number of boxes returned may be less unless *exact=True*)
- **anchors** (*ndarray*) – prior normalized aspect ratios for negative boxes. Mutually exclusive with *window_size*.
- **window_size** (*ndarray*) – absolute (W, H) sizes to use for negative boxes. Mutually exclusive with *anchors*.
- **gids** (*List[int]*) – image-ids to generate negatives for, if not specified generates for all images.
- **thresh** (*float*) – overlap area threshold as a percentage of the negative box size. When thresh=0.0, that means negatives cannot overlap any positive, when threh=1.0, there are no constrains on negative placement.
- **exact** (*bool*) – if True, ensure that we generate exactly *num* boxes
- **rng** (*RandomState*) – random number generator

Example

```
>>> from ndsampler.isect_indexer import *
>>> import ndsampler
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> self = FrameIntersectionIndex.from_coco(dset)
>>> anchors = np.array([[.35, .15], [.2, .2], [.1, .1]])
>>> #num = 25
>>> num = 5
>>> rng = kwarray.ensure_rng(None)
>>> neg_gids, neg_boxes = self.random_negatives(
>>>     num, anchors, gids=[1], rng=rng, thresh=0.01, exact=1)
>>> # xdoc: +REQUIRES(--show)
>>> gid = sorted(set(neg_gids))[0]
```

(continues on next page)

(continued from previous page)

```
>>> boxes = neg_boxes.compress(neg_gids == gid)
>>> import kwplot
>>> kwplot.autompl()
>>> img = kwimage.imread(dset.imgs[gid]['file_name'])
>>> kwplot.imshow(img, doclf=True, fnum=1, colorspace='bgr')
>>> support = self._support(gid)
>>> kwplot.draw_boxes(support, color='blue')
>>> kwplot.draw_boxes(boxes, color='orange')
```

Example

```
>>> from ndsampler.isect_indexer import *
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> self = FrameIntersectionIndex.from_coco(dset)
>>> #num = 25
>>> num = 5
>>> rng = kwarray.ensure_rng(None)
>>> window_size = (50, 50)
>>> neg_gids, neg_boxes = self.random_negatives(
>>>     num, window_size=window_size, gids=[1], rng=rng,
>>>     thresh=0.01, exact=1)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> gid = sorted(set(neg_gids))[0]
>>> boxes = neg_boxes.compress(neg_gids == gid)
>>> img = kwimage.imread(dset.imgs[gid]['file_name'])
>>> kwplot.imshow(img, doclf=True, fnum=1, colorspace='bgr')
>>> support = self._support(gid)
>>> support.draw(color='blue')
>>> boxes.draw(color='orange')
```

_debug_index()

_support(*gid*)

ndsampler.toydata

Module Contents

Classes

DynamicToySampler	Generates positive and negative samples on the fly.

class ndsampler.toydata.**DynamicToySampler**(*n_positives=100000.0*, *seed=None*, *gsize=(416, 416)*, *categories=None*)

Bases: *ndsampler.abstract_sampler.AbstractSampler*

Generates positive and negative samples on the fly.

Note: Its probably more robust to generate a static fixed-size dataset with 'demodata_toy_dset' or *kwcoco.CocoDataset.demo*. However, if you need a sampler that dynamically generates toydata, this is for you.

Ignore:

```
>>> from ndsampler.toydata import *
>>> self = DynamicToySampler()
>>> window_dims = (96, 96)
```

img, anns = self.load_positive(window_dims=window_dims) kwplot.autompl() kwplot.imshow(img['imdata'])

img, anns = self.load_negative(window_dims=window_dims) kwplot.autompl() kwplot.imshow(img['imdata'])

CommandLine:

xdoctest -m ndsampler.toydata DynamicToySampler –show

Example

```
>>> # Test that this sampler works with the dataset
>>> from ndsampler.toydata import *
>>> self = DynamicToySampler(1e3)
>>> imgs = [self.load_positive()['im'] for _ in range(9)]
>>> # xdoctest: +REQUIRES(--show)
>>> stacked = kwimage.stack_images_grid(imgs, overlap=-10)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(stacked)
>>> kwplot.show_if_requested()
```

load_item(*index*, *pad=None*, *window_dims=None*)

Loads from positives and then negatives.

__len__()

_depends()

property class_ids

property n_positives

property n_annots

property n_images

image_ids()

lookup_class_name(*class_id*)

lookup_class_id(*class_name*)

_lookup_kpnames(*class_id*)

property n_categories

preselect(*n_pos=None*, *n_neg=None*, *neg_to_pos_ratio=None*, *window_dims=None*, *rng=None*, *verbose=0*)

Setup a pool of training examples before the epoch begins

load_image(*image_id=None*, *rng=None*)

load_image_with_annots(*image_id=None*, *rng=None*)

Returns a random image and its annotations

abstract load_sample(*tr*, *pad=None*, *window_dims=None*)

_load_toy_sample(*window_dims*, *pad*, *rng*, *centerobj*, *n_annots*)

load_positive(*index=None*, *pad=None*, *window_dims=None*, *rng=None*)

Note: window_dims is height / width

Example

```
>>> from ndsampler.toydata import *
>>> self = DynamicToySampler(1e2)
>>> sample = self.load_positive()
>>> annots = sample['annots']
>>> assert len(annots['aids']) > 0
>>> assert len(annots['rel_cxywh']) == len(annots['aids'])
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> # Draw box in relative sample context
>>> kwplot.imshow(sample['im'], pnum=(1, 1, 1), fnum=1)
>>> annots['rel_boxes'].translate([-.5, -.5]).draw()
>>> annots['rel_ssegs'].draw(color='red', alpha=.6)
>>> annots['rel_kpts'].draw(color='green', alpha=.8, radius=4)
```

load_negative(*index=None*, *pad=None*, *window_dims=None*, *rng=None*)

1.1.3 Package Contents

Classes

HashIdentifiable	A class is hash-identifiable if its invariants can be tied to a specific
Frames	Abstract implementation of Frames.
SimpleFrames	Basic concrete implementation of frames objects for images where there is a
AbstractSampler	API for Samplers, not all methods need to be implemented depending on the
CategoryTree	Wrapper that maintains flat or hierarchical category information.
CocoFrames	wrapper around coco-style dataset to allow for getitem syntax
CocoRegions	Converts Coco-Style datasets into a table for efficient online work
Targets	Abstract API
CocoSampler	Samples patches of positives and negative detection windows from a COCO
FrameIntersectionIndex	Build spatial tree for each frame so we can quickly determine if a random
DynamicToySampler	Generates positive and negative samples on the fly.

Functions

select_positive_regions(targets[, window_dims, ...])	Reduce positive example redundency by selecting disparate positive samples
tabular_coco_targets(dset)	Transforms COCO box annotations into a tabular form

class ndsampler.**HashIdentifiable**(***kwargs*)

Bases: object

A class is hash-identifiable if its invariants can be tied to a specific list of hashable dependencies.

The inheriting class must either:

- implement *_depends*
- implement *_make_hashid*
- define *_hashid*

Example

class Base:

def __init__(self):
commenting the next line removes cooperative inheritence super().__init__() self.base = 1

class Derived(Base, HashIdentifiable):

def __init__(self):
super().__init__() self.defived = 1

self = Derived() dir(self)

abstract _depends()

_make_hashid()

property hashid

class ndsampler.**Frames**(*hashid_mode='PATH'*, *workdir=None*, *backend=None*)

Bases: object

Abstract implementation of Frames.

While this is an abstract class, it contains most of the `Frames` functionality. The inheriting class needs to overload the constructor and `_lookup_gpath`, which maps an image-id to its path on disk.

Parameters

- **hashid_mode** (*str, default='PATH'*) – The method used to compute a unique identifier for every image. to can be PATH, PIXELS, or GIVEN. TODO: Add DVC as a method (where it uses the name of the symlink)?
- **workdir** (*PathLike*) – This is the directory where *Frames* can store cached results. This SHOULD be specified.
- **backend** (*str | Dict*) – Determine the backend to use for fast subimage region lookups. This can either be a string 'cog' or 'npy'. This can also be a config dictionary for fine-grained backend control. For this case, 'type': specified cog or npy, and only COG has additional options which are:

  {
  'type': 'cog', 'config': { 'compress': <'LZW' | 'JPEG | 'DEFLATE' | 'ZSTD' | 'auto'>, }

  }

Example

```
>>> from ndsampler.abstract_frames import *
>>> self = SimpleFrames.demo(backend='npy')
>>> file = self.load_image(1)
>>> print('file = {!r}'.format(file))
>>> assert self.load_image(1).shape == (512, 512, 3)
>>> assert self.load_region(1, (slice(-20), slice(-10))).shape == (492, 502, 3)
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> self = SimpleFrames.demo(backend='cog')
>>> assert self.load_image(1).shape == (512, 512, 3)
>>> assert self.load_region(1, (slice(-20), slice(-10))).shape == (492, 502, 3)
```

Benchmark:

```
>>> from ndsampler.abstract_frames import *  # NOQA
>>> import ubelt as ub
>>> #
>>> ti = ub.Timerit(100, bestof=3, verbose=2)
>>> #
>>> self = SimpleFrames.demo(backend='cog')
>>> for timer in ti.reset('cog-small-subregion'):
```

(continues on next page)

(continued from previous page)

```
>>>     self.load_image(1)[10:42, 10:42]
>>> #
>>> self = SimpleFrames.demo(backend='npy')
>>> for timer in ti.reset('npy-small-subregion'):
>>>     self.load_image(1)[10:42, 10:42]
>>> print('----')
>>> #
>>> self = SimpleFrames.demo(backend='cog')
>>> for timer in ti.reset('cog-large-subregion'):
>>>     self.load_image(1)[3:-3, 3:-3]
>>> #
>>> self = SimpleFrames.demo(backend='npy')
>>> for timer in ti.reset('npy-large-subregion'):
>>>     self.load_image(1)[3:-3, 3:-3]
>>> print('----')
>>> #
>>> self = SimpleFrames.demo(backend='cog')
>>> for timer in ti.reset('cog-loadimage'):
>>>     self.load_image(1)
>>> #
>>> self = SimpleFrames.demo(backend='npy')
>>> for timer in ti.reset('npy-loadimage'):
>>>     self.load_image(1)
```

DEFAULT_NPY_CONFIG

DEFAULT_COG_CONFIG

__getstate__()

__setstate__(*state*)

_update_backend(*backend*)

change the backend and update internals accordingly

classmethod _coerce_backend_config(*backend=None*)

Coerce a backend argument into a valid configuration dictionary.

Returns

a dictionary with two items: 'type', which is a string and
and 'config', which is a dictionary of parameters for the specific type.

Return type
Dict

property cache_dpath

Returns the path where cached frame representations will be stored.

This will be None if there is no backend.

abstract _build_pathinfo(*image_id*)

A user specified function that maps an image id to paths to relevant resources on disk. These resources are also indexed by channel.

SeeAlso:
`_populate_chan_info` for helping populate cache info in each channel.

Parameters
image_id – the image id (usually an integer)

Returns

with the following structure:

{
<NotFinalized> 'channels': {
<channel_spec>: {'path': <abspath>, ...}, ...
}
}

Return type
Dict

`_lookup_pathinfo`(*image_id*)

`_populate_chan_info`(*chan*, *root=''*)

Helper to construct a path dictionary in the `_build_pathinfo` method based on the current hashing and caching settings.

`static _build_file_hashid`(*root*, *suffix*, *hashid_mode*)

Build a hashid for a specific file given as a path root and suffix.

`property image_ids`

`__len__`()

`__getitem__`(*index*)

`load_region`(*image_id*, *region=None*, *channels=ub.NoParam*, *width=None*, *height=None*)

Ammortized O(1) image subregion loading (assuming constant region size)

Parameters

- **image_id** (*int*) – image identifier
- **region** (*Tuple[slice, ...]*) – space-time region within an image
- **channels** (*str*) – NotImplemented
- **width** (*int*) – if the width of the entire image is know specify it
- **height** (*int*) – if the height of the entire image is know specify it

`_load_alignable`(*image_id*, *cache=True*)

`load_image`(*image_id*, *channels=ub.NoParam*, *cache=True*, *noreturn=False*)

Load the image data for a particular image id

Parameters

- **image_id** (*int*) – the id of the image to load
- **cache** (*bool, default=True*) – ensure and return the efficient backend cached representation.
- **channels** – NotImplemented
- **noreturn** (*bool, default=False*) – if True, nothing is returned. This is useful if you simply want to ensure the cached representation.

CAREFUL: THIS NEEDS TO MAINTAIN A STABLE API. OTHER PROJECTS DEPEND ON IT.

Returns

an indexable array like representation, possibly
memmapped.

Return type
ArrayLike

load_frame(*image_id*)

TODO: FINISHME or rename to lazy frame?

Returns a frame object that lazy loads on slice

prepare(*gids=None*, *workers=0*, *use_stamp=True*)

Precompute the cached frame conversions

Parameters

- **gids** (*List[int] | None*) – specific image ids to prepare. If None prepare all images.
- **workers** (*int, default=0*) – number of parallel threads for this io-bound task

Example

```
>>> from ndsampler.abstract_frames import *
>>> workdir = ub.ensure_app_cache_dir('ndsampler/tests/test_cog_precomp')
>>> print('workdir = {!r}'.format(workdir))
>>> ub.delete(workdir)
>>> ub.ensuredir(workdir)
>>> self = SimpleFrames.demo(backend='npy', workdir=workdir)
>>> print('self = {!r}'.format(self))
>>> print('self.cache_dpath = {!r}'.format(self.cache_dpath))
>>> #_ = ub.cmd('tree ' + workdir, verbose=3)
>>> self.prepare()
>>> self.prepare()
>>> #_ = ub.cmd('tree ' + workdir, verbose=3)
>>> _ = ub.cmd('ls ' + self.cache_dpath, verbose=3)
```

Example

```
>>> from ndsampler.abstract_frames import *
>>> import ndsampler
>>> workdir = ub.get_app_cache_dir('ndsampler/tests/test_cog_precomp2')
>>> ub.delete(workdir)
>>> # TEST NPY
>>> #
>>> sampler = ndsampler.CocoSampler.demo(workdir=workdir, backend='npy')
>>> self = sampler.frames
>>> ub.delete(self.cache_dpath)  # reset
>>> self.prepare()  # serial, miss
>>> self.prepare()  # serial, hit
>>> ub.delete(self.cache_dpath)  # reset
>>> self.prepare(workers=3)  # parallel, miss
```

(continues on next page)

(continued from previous page)

```
>>> self.prepare(workers=3)  # parallel, hit
>>> #
>>> ## TEST COG
>>> # xdoctest: +REQUIRES(module:osgeo)
>>> sampler = ndsampler.CocoSampler.demo(workdir=workdir, backend='cog')
>>> self = sampler.frames
>>> ub.delete(self.cache_dpath)  # reset
>>> self.prepare()  # serial, miss
>>> self.prepare()  # serial, hit
>>> ub.delete(self.cache_dpath)  # reset
>>> self.prepare(workers=3)  # parallel, miss
>>> self.prepare(workers=3)  # parallel, hit
```

class ndsampler.**SimpleFrames**(*id_to_path*, *workdir=None*, *backend=None*)

Bases: *Frames*

Basic concrete implementation of frames objects for images where there is a strict one-file-to-one-image mapping (i.e. no auxiliary images).

Parameters

id_to_path (*Dict*) – mapping from image-id to image path

Example

```
>>> from ndsampler.abstract_frames import *
>>> self = SimpleFrames.demo(backend='npy')
>>> pathinfo = self._build_pathinfo(1)
>>> print('pathinfo = {}'.format(ub.repr2(pathinfo, nl=3)))
```

```
>>> assert self.load_image(1).shape == (512, 512, 3)
>>> assert self.load_region(1, (slice(-20), slice(-10))).shape == (492, 502, 3)
```

_lookup_gpath(*image_id*)

image_ids()

classmethod demo(**kw*)

Get a smple frames object

_build_pathinfo(*image_id*)

A user specified function that maps an image id to paths to relevant resources on disk. These resources are also indexed by channel.

SeeAlso:

`_populate_chan_info` for helping populate cache info in each channel.

Parameters

image_id – the image id (usually an integer)

Returns

with the following structure:

{

<NotFinalized> ‘channels’: {

<channel_spec>: {'path': <abspath>, ...}, ...

}

}

Return type
Dict

class ndsampler.**AbstractSampler**

Bases: object

API for Samplers, not all methods need to be implemented depending on the use case (for example, load_sample may not be defined if positive / negative cases are generated on the fly).

property class_ids

abstract lookup_class_name(*class_id*)

abstract lookup_class_id(*class_name*)

abstract load_sample(*tr*, *pad=None*, *window_dims=None*, *visible_thresh=0.1*)

property n_positives

abstract load_item(*index*, *pad=None*, *window_dims=None*)

abstract load_positive(*index=None*, *pad=None*, *window_dims=None*, *rng=None*)

abstract load_negative(*index=None*, *pad=None*, *window_dims=None*, *rng=None*)

abstract load_image(*image_id*)

abstract image_ids()

abstract preselect(***kwargs*)

Setup a pool of training examples before the epoch begins

class ndsampler.**CategoryTree**(*graph=None*, *checks=True*)

Bases: kwcoco.CategoryTree, Mixin_CategoryTree_Torch

Wrapper that maintains flat or hierarchical category information.

Helps compute softmaxes and probabilities for tree-based categories where a directed edge (A, B) represents that A is a superclass of B.

Note: There are three basic properties that this object maintains:

```
node:
    Alphanumeric string names that should be generally descriptive.
    Using spaces and special characters in these names is
    discouraged, but can be done.  This is the COCO category "name"
    attribute.  For categories this may be denoted as (name, node,
    cname, catname).

id:
    The integer id of a category should ideally remain consistent.
    These are often given by a dataset (e.g. a COCO dataset).  This
    is the COCO category "id" attribute. For categories this is
```

(continues on next page)

(continued from previous page)

```
    often denoted as (id, cid).

index:
    Contigous zero-based indices that indexes the list of
    categories.  These should be used for the fastest access in
    backend computation tasks. Typically corresponds to the
    ordering of the channels in the final linear layer in an
    associated model.  For categories this is often denoted as
    (index, cidx, idx, or cx).
```

Variables

- **idx_to_node** (*List[str]*) – a list of class names. Implicitly maps from index to category name.
- **id_to_node** (*Dict[int, str]*) – maps integer ids to category names
- **node_to_id** (*Dict[str, int]*) – maps category names to ids
- **node_to_idx** (*Dict[str, int]*) – maps category names to indexes
- **graph** (*networkx.Graph*) – a Graph that stores any hierarchy information. For standard mutually exclusive classes, this graph is edgeless. Nodes in this graph can maintain category attributes / properties.
- **idx_groups** (*List[List[int]]*) – groups of category indices that share the same parent category.

Example

```
>>> from kwcoco.category_tree import *
>>> graph = nx.from_dict_of_lists({
>>>     'background': [],
>>>     'foreground': ['animal'],
>>>     'animal': ['mammal', 'fish', 'insect', 'reptile'],
>>>     'mammal': ['dog', 'cat', 'human', 'zebra'],
>>>     'zebra': ['grevys', 'plains'],
>>>     'grevys': ['fred'],
>>>     'dog': ['boxer', 'beagle', 'golden'],
>>>     'cat': ['maine coon', 'persian', 'sphynx'],
>>>     'reptile': ['bearded dragon', 't-rex'],
>>> }, nx.DiGraph)
>>> self = CategoryTree(graph)
>>> print(self)
<CategoryTree(nNodes=22, maxDepth=6, maxBreadth=4...)>
```

Example

```
>>> # The coerce classmethod is the easiest way to create an instance
>>> import kwcoco
>>> kwcoco.CategoryTree.coerce(['a', 'b', 'c'])
<CategoryTree...nNodes=3, nodes=...'a', 'b', 'c'...
>>> kwcoco.CategoryTree.coerce(4)
<CategoryTree...nNodes=4, nodes=...'class_1', 'class_2', 'class_3', ...
>>> kwcoco.CategoryTree.coerce(4)
```

class ndsampler.**CocoFrames**(*dset*, *hashid_mode='PATH'*, *workdir=None*, *verbose=0*, *backend='auto'*)

Bases: *ndsampler.abstract_frames.Frames*, *ndsampler.utils.util_misc.HashIdentifiable*

wrapper around coco-style dataset to allow for getitem syntax

CommandLine:

xdoctest -m ndsampler.coco_frames CocoFrames

Example

```
>>> from ndsampler.coco_frames import *
>>> import ndsampler
>>> import kwcoco
>>> import ubelt as ub
>>> workdir = ub.ensure_app_cache_dir('ndsampler')
>>> dset = kwcoco.CocoDataset.demo(workdir=workdir)
>>> dset._ensure_imgsize()
>>> self = CocoFrames(dset, workdir=workdir)
>>> assert self.load_image(1).shape == (512, 512, 3)
>>> assert self.load_image(1)[:-20, :-10].shape == (492, 502, 3)
>>> assert self.load_region(1, (slice(-20), slice(-10))).shape == (492, 502, 3)
```

Example

```
>>> from ndsampler import coco_sampler
>>> self = coco_sampler.CocoSampler.demo().frames
>>> assert self.load_image(1).shape == (600, 600, 3)
>>> assert self.load_image(1)[:-20, :-10].shape == (580, 590, 3)
```

property image_ids

_make_hashid()

load_region(*image_id*, *region=None*, *channels=ub.NoParam*)

Ammortized O(1) image subregion loading (assuming constant region size)

Parameters

- **image_id** (*int*) – image identifier
- **region** (*Tuple[slice, ...]*) – space-time region within an image
- **channels** (*str*) – NotImplemented
- **width** (*int*) – if the width of the entire image is know specify it

- **height** (*int*) – if the height of the entire image is know specify it

_build_pathinfo(*image_id*)

Returns
See Parent Method Docs

Example

```
>>> import ndsampler
>>> sampler1 = ndsampler.CocoSampler.demo('vidshapes5-aux')
>>> sampler2 = ndsampler.CocoSampler.demo('vidshapes5-multispectral')
>>> self = sampler1.frames
>>> pathinfo = self._build_pathinfo(1)
>>> print('pathinfo = {}'.format(ub.repr2(pathinfo, nl=3)))
```

```
>>> self = sampler2.frames
>>> pathinfo = self._build_pathinfo(1)
>>> print('pathinfo = {}'.format(ub.repr2(pathinfo, nl=3)))
```

class ndsampler.**CocoRegions**(*dset*, *workdir=None*, *verbose=1*)

Bases: *Targets*, *ndsampler.utils.util_misc.HashIdentifiable*, ubelt.NiceRepr

Converts Coco-Style datasets into a table for efficient on-line work

Perhaps rename this class to regions, and then have targets be an attribute of regions.

Parameters

- **dset** (*ndsampler.CocoAPI*) – a dataset in coco format
- **workdir** (*PathLike*) – a temporary directory where we can cache stuff
- **verbose** (*int*) – verbosity level

Example

```
>>> from ndsampler.coco_regions import *
>>> self = CocoRegions.demo()
>>> pos_tr = self.get_positive(rng=0)
>>> neg_tr = self.get_negative(rng=0)
>>> print(ub.repr2(pos_tr, precision=2))
>>> print(ub.repr2(neg_tr, precision=2))
```

property catgraph

property n_negatives

property n_positives

property n_samples

property class_ids

property image_ids

property n_annots

property n_images

property n_categories

lookup_class_name(*class_id*)

lookup_class_id(*class_name*)

__nice__()

classmethod demo()

_make_hashid()

property isect_index

Lazy access to a disk-cached intersection index for this dataset

_lazy_isect_index(*verbose=None*)

property targets

All viable positive annotations targets in a flat table.

The main idea is that this is the population of all positives that we could sample from. Often times we will simply use all of them.

This function takes a subset of annotations in the coco dataset that can be considered "viable" positives. We may subsample these futher, but this serves to collect the annotations that could feasibly be used by the network. Essentailly we remove annotations without bounding boxes. I'm not sure I 100% like the way this works though. Shouldn't filtering be done before we even get here? Perhaps but perhaps not. This design needs a bit more thought.

property neg_anchors

overlapping_aids(*gid*, *region*, *visible_thresh=0.0*)

Finds the other annotations in this image that overlap a region

Parameters

- **gid** (*int*) – image id
- **region** (*kwimage.Boxes*) – bounding box
- **visible_thresh** (*float*) – does not return annotations with visibility less than this threshold.

Returns
annotation ids

Return type
List[int]

get_segmentations(*aids*)

Returns the segmentations corresponding to a set of annotation ids

Example

```
>>> from ndsampler.coco_regions import *
>>> from ndsampler import coco_sampler
>>> self = coco_sampler.CocoSampler.demo().regions
>>> aids = [1, 2]
```

get_negative(*index=None*, *rng=None*)

Get localization information for a negative region

Parameters

- **index** (*int or None*) – indexes into the current negative pool or if None returns a random negative
- **rng** (*RandomState*) – used only if index is None

Returns
tr: target info dictionary

Return type
Dict

CommandLine:
xdoctest -m ndsampler.coco_regions CocoRegions.get_negative

Example

```
>>> from ndsampler.coco_regions import *
>>> from ndsampler import coco_sampler
>>> rng = kwarray.ensure_rng(0)
>>> self = coco_sampler.CocoSampler.demo().regions
>>> tr = self.get_negative(rng=rng)
>>> # xdoctest: +IGNORE_WANT
>>> assert 'category_id' in tr
>>> assert 'aid' in tr
>>> assert 'cx' in tr
>>> print(ub.repr2(tr, precision=2))
{
    'aid': -1,
    'category_id': 0,
    'cx': 190.71,
    'cy': 95.83,
    'gid': 1,
    'height': 140.00,
    'img_height': 600,
    'img_width': 600,
    'width': 68.00,
}
```

get_positive(*index=None*, *rng=None*)

Get localization information for a positive region

Parameters

- **index** (*int or None*) – indexes into the current positive pool or if None returns a random negative
- **rng** (*RandomState*) – used only if index is None

Returns
tr: target info dictionary

Return type
Dict

Example

```
>>> from ndsampler import coco_sampler
>>> rng = kwarray.ensure_rng(0)
>>> self = coco_sampler.CocoSampler.demo().regions
>>> tr = self.get_positive(0, rng=rng)
>>> print(ub.repr2(tr, precision=2))
```

get_item(*index*, *rng=None*)

Loads from positives and then negatives.

_random_negatives(*num*, *exact=False*, *neg_anchors=None*, *window_size=None*, *rng=None*, *thresh=0.0*)

Samples multiple negatives at once for efficiency

Parameters

- **num** (*int*) – number of negatives to sample
- **exact** (*bool*) – if True, we will try to find exactly *num* negatives, otherwise the number returned is approximate.
- **neg_anchors** () – prior normalized aspect ratios for negative boxes. Mutually exclusive with *window_size*.
- **window_size** (*Tuple*) – absolute box size (width, height) used to sample negative regions. If not specified the relative anchor strategy will be used to randomly choose potentially non-square regions relative to the image size.
- **thresh** (*float*) – overlap area threshold as a percentage of the negative box size. When thresh=0.0, that means negatives cannot overlap any positive, when threh=1.0, there are no constrains on negative placement.

Returns
targets - contains negative target information

Return type
DataFrameArray

Example

```
>>> from ndsampler.coco_regions import *
>>> from ndsampler import coco_sampler
>>> self = coco_sampler.CocoSampler.demo().regions
>>> num = 100
>>> rng = kwarray.ensure_rng(0)
>>> targets = self._random_negatives(num, rng=rng)
>>> assert len(targets) <= num
>>> targets = self._random_negatives(num, exact=True)
>>> assert len(targets) == num
```

new_sample_grid(*task*, *window_dims*, *window_overlap=0*, ***kwargs*)

New experimental method to replace preselect positives / negatives

Parameters

- **task** (*str*) – can be video_detection image_detection # video_classification # image_classification
- ****kwargs** – passed to *new_video_sample_grid* or *new_image_sample_grid*

Example

```
>>> from ndsampler.coco_regions import *
>>> from ndsampler import coco_sampler
>>> self = coco_sampler.CocoSampler.demo('vidshapes1').regions
>>> self.dset.conform()
>>> sample_grid = self.new_sample_grid('video_detection', window_dims=(2, 100,␣
↪100))
```

_preselect_positives(*num=None*, *window_dims=None*, *rng=None*, *verbose=None*)

” preload a bunch of positives

Example

```
>>> from ndsampler.coco_regions import *
>>> from ndsampler import coco_sampler
>>> self = coco_sampler.CocoSampler.demo().regions
>>> window_dims = (64, 64)
>>> self._preselect_positives(window_dims=window_dims, verbose=4)
```

_preselect_negatives(*num*, *window_dims=None*, *thresh=0.3*, *rng=None*, *verbose=None*)

Preselect a set of random regions to be used as negative examples.

Parameters

- **num** (*int*) – number of desired negatives to preselect. In some cases achieving this number may not be possible.
- **window_dims** (*Tuple*) – absolute dimensions (height, width) used to sample negative regions. If not specified the relative anchor strategy will be used to randomly choose potentially non-square regions relative to the image size.

- **thresh** (*float*) – overlap area threshold as a percentage of the negative box size. When thresh=0.0, that means negatives cannot overlap any positive, when threh=1.0, there are no constrains on negative placement.
- **rng** (*int | RandomState*) – random seed / state
- **verbose** (*int*) – verbosity level

Returns
number of negatives actually chosen

Return type
int

Example

```
>>> from ndsampler.coco_regions import *
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo().regions
>>> num = 100
>>> self._preselect_negatives(num, window_dims=(30, 30))
```

_cacher(*fname*, *extra_deps=None*, *disable=False*, *verbose=None*)

Create a cacher for a known lazy computation using a common hashid.

If *self.workdir* or *self.hashid* is None, then caches are disabled by default. Caches can be explicitly disabled by setting the appropriate value in the *self._enabled_caches* dictionary.

Parameters

- **fname** (*str*) – name of the property we are caching
- **extra_deps** (*OrderedDict*) – extra data to contribute to the hashid
- **disable** (*bool*) – explicitly disable cache if True, otherwise do normal checks to see if enabled.
- **verbose** (*bool, default=None*) – if specified overrides *self.verbose*.

Returns

cacher - if enabled this cacher will minimally depend
on the *self.hashid*, but may also depend on extra info.

Return type
ub.Cacher

exception ndsampler.**MissingNegativePool**

Bases: AssertionError

Assertion failed.

class ndsampler.**Targets**

Bases: object

Abstract API

get_negative(*index=None*, *rng=None*)

get_positive(*index=None*, *rng=None*)

abstract overlapping_aids(*gid*, *box*)

preselect(*n_pos=None*, *n_neg=None*, *neg_to_pos_ratio=None*, *window_dims=None*, *rng=None*, *verbose=0*)

Shuffle selection of positive and negative samples

Todo: [X] Basic, window around positive annotation algorithm [ ] Sliding window algorithm from bioharn

---

ndsampler.**select_positive_regions**(*targets*, *window_dims=(300, 300)*, *thresh=0.0*, *rng=None*, *verbose=0*)

Reduce positive example redundency by selecting disparate positive samples

Example

```
>>> from ndsampler.coco_regions import *
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> targets = tabular_coco_targets(dset)
>>> window_dims = (300, 300)
>>> selected = select_positive_regions(targets, window_dims)
>>> print(len(selected))
>>> print(len(dset.anns))
```

ndsampler.**tabular_coco_targets**(*dset*)

Transforms COCO box annotations into a tabular form

_ = xdev.profile_now(tabular_coco_targets)(dset)

class ndsampler.**CocoSampler**(*dset*, *workdir=None*, *autoinit=True*, *backend=None*, *verbose=0*)

Bases: *ndsampler.abstract_sampler.AbstractSampler*, *ndsampler.utils.util_misc.HashIdentifiable*, ubelt.NiceRepr

Samples patches of positives and negative detection windows from a COCO dataset. Can be used for training FCN or RPN based classifiers / detectors.

Does data loading, padding, etc…

Parameters

- **dset** (*kwcoco.CocoDataset*) – a coco-formatted dataset
- **backend** (*str | Dict*) – either ‘cog’ or ‘npy’, or a dict with *{‘type’: str, ‘config’: Dict}*. See AbstractFrames for more details. Defaults to None, which does not do anything fancy.

Example

#print >>> from ndsampler.coco_sampler import * >>> self = CocoSampler.demo(‘photos’) … >>> print(sorted(self.class_ids)) [0, 1, 2, 3, 4, 5, 6, 7, 8] >>> print(self.n_positives) 4

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo('photos')
>>> p_sample = self.load_positive()
>>> n_sample = self.load_negative()
>>> self = ndsampler.CocoSampler.demo('shapes')
>>> p_sample2 = self.load_positive()
>>> n_sample2 = self.load_negative()
>>> for sample in [p_sample, n_sample, p_sample2, n_sample2]:
>>>     assert 'annots' in sample
>>>     assert 'im' in sample
>>>     assert 'rel_boxes' in sample['annots']
>>>     assert 'rel_ssegs' in sample['annots']
>>>     assert 'rel_kpts' in sample['annots']
>>>     assert 'cids' in sample['annots']
>>>     assert 'aids' in sample['annots']
```

classmethod demo(*key='shapes'*, *workdir=None*, *backend=None*, ***kw*)

Create a toy coco sampler for testing and demo puposes

SeeAlso:

- kwcoco.CocoDataset.demo

_init()

property classes

property catgraph

DEPRICATED, use self.classes instead

_depends()

lookup_class_name(*class_id*)

lookup_class_id(*class_name*)

property n_positives

property n_annots

property n_samples

__len__()

property n_images

property n_categories

property class_ids

property image_ids

preselect(***kwargs*)

Setup a pool of training examples before the epoch begins

new_sample_grid(*task*, *window_dims*, *window_overlap=0*)

load_image_with_annots(*image_id*, *cache=True*)

Parameters

- **image_id** (*int*) – the coco image id
- **cache** (*bool, default=True*) – if True returns the fast subregion-indexable file reference. Otherwise, eagerly loads the entire image.

Returns
img: the coco image dict augmented with imdata anns: the coco annotations in this image

Return type
Tuple[Dict, List[Dict]]

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> img, anns = self.load_image_with_annots(1)
>>> dets = kwimage.Detections.from_coco_annots(anns, dset=self.dset)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'][:], doclf=1)
>>> dets.draw()
>>> kwplot.show_if_requested()
```

load_annotations(*image_id*)

Loads the annotations within an image

Parameters
image_id (*int*) – the coco image id

Returns
list of coco annotation dictionaries

Return type
List[Dict]

load_image(*image_id*, *cache=True*)

Loads the annotations within an image

Parameters

- **image_id** (*int*) – the coco image id
- **cache** (*bool, default=True*) – if True returns the fast subregion-indexable file reference. Otherwise, eagerly loads the entire image.

Returns
either ndarray data or a indexable reference

Return type
ArrayLike

load_item(*index*, *with_annots=True*, *target=None*, *rng=None*, ***kw*)

Loads item from either positive or negative regions pool.

Lower indexes will return positive regions and higher indexes will return negative regions.

The main paradigm of the sampler is that sampler.regions maintains a pool of target regions, you can influence what that pool is at any point by calling sampler.regions.preselect (usually either at the start of learning, or maybe after every epoch, etc..), and you use load_item to load the index-th item from that preselected pool. Depending on how you preselected the pool, the returned item might correspond to a positive or negative region.

Parameters

- **index** (*int*) – index of target region
- **with_annots** (*bool | str, default=True*) – if True, also extracts information about any annotation that overlaps the region of interest (subject to visibility_thresh). Can also be a List[str] that specifies which specific subinfo should be extracted. Valid strings in this list are: boxes, keypoints, and segmenation.
- **target** (*Dict*) – Extra target arguments that update the positive target, like window_dims, pad, etc.... See `load_sample()` for details on allowed keywords.
- **rng** (*None | int | RandomState*) – a seed or seeded random number generator.
- ****kw** – other arguments that can be passed to `CocoSampler.load_sample()`

Returns

sample: dict containing keys
im (ndarray): image data target (dict): contains the same input items as the input

target but additionally specifies inferred information like rel_cx and rel_cy, which gives the center of the target w.r.t the returned **padded** sample.

annots (dict): Dict of aids, cids, and rel/abs boxes

Return type
Dict

`load_positive`(*index=None*, *with_annots=True*, *target=None*, *rng=None*, ***kw*)

Load an item from the the positive pool of regions.

Parameters

- **index** (*int*) – index of positive target
- **with_annots** (*bool | str, default=True*) – if True, also extracts information about any annotation that overlaps the region of interest (subject to visibility_thresh). Can also be a List[str] that specifies which specific subinfo should be extracted. Valid strings in this list are: boxes, keypoints, and segmentation.
- **target** (*Dict*) – Extra target arguments that update the positive target, like window_dims, pad, etc.... See `load_sample()` for details on allowed keywords.
- **rng** (*None | int | RandomState*) – a seed or seeded random number generator.
- ****kw** – other arguments that can be passed to `CocoSampler.load_sample()`

Returns

sample: dict containing keys
im (ndarray): image data tr (dict): contains the same input items as tr but additionally

specifies rel_cx and rel_cy, which gives the center of the target w.r.t the returned **padded** sample.

annots (dict): Dict of aids, cids, and rel/abs boxes

Return type
Dict

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> sample = self.load_positive(pad=(10, 10), tr=dict(window_dims=(3, 3)))
>>> assert sample['im'].shape[0] == 23
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(sample['im'], doclf=1)
>>> kwplot.show_if_requested()
```

load_negative(*index=None*, *with_annots=True*, *target=None*, *rng=None*, ***kw*)

Load an item from the the negative pool of regions.

Parameters

- **index** (*int*) – if specified loads a specific negative from the presampled pool, otherwise the next negative in the pool is returned.
- **with_annots** (*bool | str, default=True*) – if True, also extracts information about any annotation that overlaps the region of interest (subject to visibility_thresh). Can also be a List[str] that specifies which specific subinfo should be extracted. Valid strings in this list are: boxes, keypoints, and segmentation.
- **target** (*Dict*) – Extra target arguments that update the positive target, like window_dims, pad, etc.... See *load_sample()* for details on allowed keywords.
- **rng** (*None | int | RandomState*) – a seed or seeded random number generator.

Returns

sample: dict containing keys
im (ndarray): image data tr (dict): contains the same input items as tr but additionally

specifies rel_cx and rel_cy, which gives the center of the target w.r.t the returned **padded** sample.

annots (dict): Dict of aids, cids, and rel/abs boxes

Return type
Dict

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> rng = None
>>> sample = self.load_negative(rng=rng, pad=(0, 0))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> import kwimage
>>> kwplot.autompl()
```

(continues on next page)

(continued from previous page)

```
>>> abs_sample_box = sample['params']['sample_tlbr']
>>> tf_rel_from_abs = kwimage.Affine.coerce(sample['params']['tf_rel_to_abs']).
↪inv()
>>> wh, ww = sample['target']['window_dims']
>>> abs_window_box = kwimage.Boxes([[sample['target']['cx'], sample['target'][
↪'cy'], ww, wh]], 'cxywh')
>>> rel_window_box = abs_window_box.warp(tf_rel_from_abs)
>>> rel_sample_box = abs_sample_box.warp(tf_rel_from_abs)
>>> kwplot.imshow(sample['im'], fnum=1, doclf=True)
>>> rel_sample_box.draw(color='kw_green', lw=10)
>>> rel_window_box.draw(color='kw_blue', lw=8)
>>> kwplot.show_if_requested()
```

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> rng = None
>>> sample = self.load_negative(rng=rng, pad=(10, 20), target=dict(window_
↪dims=(64, 64)))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> import kwimage
>>> kwplot.autompl()
>>> abs_sample_box = sample['params']['sample_tlbr']
>>> tf_rel_from_abs = kwimage.Affine.coerce(sample['params']['tf_rel_to_abs']).
↪inv()
>>> wh, ww = sample['target']['window_dims']
>>> abs_window_box = kwimage.Boxes([[sample['target']['cx'], sample['target'][
↪'cy'], ww, wh]], 'cxywh')
>>> rel_window_box = abs_window_box.warp(tf_rel_from_abs)
>>> rel_sample_box = abs_sample_box.warp(tf_rel_from_abs)
>>> kwplot.imshow(sample['im'], fnum=1, doclf=True)
>>> rel_sample_box.draw(color='kw_green', lw=10)
>>> rel_window_box.draw(color='kw_blue', lw=8)
>>> kwplot.show_if_requested()
```

load_sample(*target=None*, *with_annots=True*, *visible_thresh=0.0*, ***kwargs*)

Loads the volume data associated with the bbox and frame of a target

Parameters

- **target** (*dict*) – target dictionary (often abbreviated as tr) indicating an nd source object (e.g. image or video) and the coordinate region to sample from. Unspecified coordinate regions default to the extent of the source object.

  For 2D image source objects, target must contain or be able to infer the key *gid (int)*, to specify an image id.

  For 3D video source objects, target must contain the key *vidid (int)*, to specify a video id (NEW in 0.6.1) or *gids List[int]*, as a list of images in a video (NEW in 0.6.2)

  In general, coordinate regions can specified by the key *slices*, a numpy-like "fancy

index" over each of the n dimensions. Usually this is a tuple of slices, e.g. (y1:y2, x1:x2) for images and (t1:t2, y1:y2, x1:x2) for videos.

You may also specify: *space_slice* as (y1:y2, x1:x2) for both 2D images and 3D videos and *time_slice* as t1:t2 for 3D videos.

Spatial regions can be specified with keys:

- 'cx' and 'cy' as the center of the region in pixels.
- 'width' and 'height' are in pixels.
- 'window_dims' is a height, width tuple or can be a

special string key 'square', which overrides width and height to both be the maximum of the two.

Temporal regions are specifiable by *slices*, *time_slice* or an explicit list of *gids*.

The *aid* key can be specified to indicate a specific annotation to load. This uses the annotation information to infer 'gid', 'cx', 'cy', 'width', and 'height' if they are not present. (NEW in 0.5.10)

The *channels* key can be specified as a channel code or
`kwcoco.ChannelSpec` object. (NEW in 0.6.1)

as_xarray (bool, default=False):
if True, return the image data as an xarray object

interpolation (str, default='auto'):
type of resample interpolation

antialias (str, default='auto'):
antialias sample or not

nodata: override function level nodata

use_native_scale (bool): If True, the "im" field is returned
as a jagged list of data that are as close to native resolution as possible while still maintaining alignment up to a scale factor. Currently only available for video sampling.

scale (float | Tuple[float, float]):
if specified, add an extra scale factor to the data.

pad (tuple): (height, width) extra context to add to window dims.
This helps prevent augmentation from producing boundary effects

padkw (dict): kwargs for *numpy.pad*.
Defaults to {'mode': 'constant'}.

dtype (type | None):
Cast the loaded data to this type. If unspecified returns the data as-is.

nodata (int | None, default=None):
If specified, for integer data with nodata values, this is passed to kwcoco delayed image finalize. The data is converted to float32 and nodata values are replaced with nan. These nan values are handled correctly in subsequent warping operations.

- **with_annots** (*bool | str, default=True*) – if True, also extracts information about any annotation that overlaps the region of interest (subject to visibility_thresh). Can also be a List[str] that specifies which specific subinfo should be extracted. Valid strings in this list are: boxes, keypoints, and segmentation.

- **visible_thresh** (*float*) – does not return annotations with visibility less than this threshold.
- ****kwargs** – handles deprecated arguments which are now specified in the target dictionary itself.

Returns

sample: dict containing keys
im (ndarray | DataArray): image / video data target (dict): contains the same input items as the input

target but additionally specifies inferred information like rel_cx and rel_cy, which gives the center of the target w.r.t the returned **padded** sample.

annots (dict): containing items:

frame_dets (List[kwimage.Detections]): a list of detection
objects containing the requested annotation info for each frame.

aids (list): annotation ids DEPRECATED cids (list): category ids DEPRECATED rel_ssegs (ndarray): segmentations relative to the sample DEPRECATED rel_kpts (ndarray): keypoints relative to the sample DEPRECATED

Return type
Dict

CommandLine:
xdoctest -m ndsampler.coco_sampler CocoSampler.load_sample:2 –show

xdoctest -m ndsampler.coco_sampler CocoSampler.load_sample:1 –show xdoctest -m ndsampler.coco_sampler CocoSampler.load_sample:3 –show

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> # The target (target) lets you specify an arbitrary window
>>> target = {'gid': 1, 'cx': 5, 'cy': 2, 'width': 6, 'height': 6}
>>> sample = self.load_sample(target)
...
>>> print('sample.shape = {!r}'.format(sample['im'].shape))
sample.shape = (6, 6, 3)
```

Example

```
>>> # Access direct annotation information
>>> import ndsampler
>>> sampler = ndsampler.CocoSampler.demo()
>>> # Sample a region that contains at least one annotation
>>> target = {'gid': 1, 'cx': 5, 'cy': 2, 'width': 600, 'height': 600}
>>> sample = sampler.load_sample(target)
>>> annotation_ids = sample['annots']['aids']
>>> aid = annotation_ids[0]
```

(continues on next page)

(continued from previous page)

```
>>> # Method1: Access ann dict directly via the coco index
>>> ann = sampler.dset.anns[aid]
>>> # Method2: Access ann objects via annots method
>>> dets = sampler.dset.annots(annotation_ids).detections
>>> print('dets.data = {}'.format(ub.repr2(dets.data, nl=1)))
```

Example

```
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> target = self.regions.get_positive(0)
>>> target['window_dims'] = 'square'
>>> target['pad'] = (25, 25)
>>> sample = self.load_sample(target)
>>> print('im.shape = {!r}'.format(sample['im'].shape))
im.shape = (135, 135, 3)
>>> target['window_dims'] = None
>>> target['pad'] = (0, 0)
>>> sample = self.load_sample(target)
>>> print('im.shape = {!r}'.format(sample['im'].shape))
im.shape = (52, 85, 3)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(sample['im'])
>>> kwplot.show_if_requested()
```

Example

```
>>> # sample an out of bounds target
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> target = self.regions.get_positive(0)
>>> target['window_dims'] = (364, 364)
>>> sample = self.load_sample(target)
>>> annots = sample['annots']
>>> assert len(annots['aids']) > 0
>>> #assert len(annots['rel_cxywh']) == len(annots['aids'])
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> abs_frame = self.frames.load_image(sample['target']['gid'])[:]
>>> tf_rel_to_abs = sample['params']['tf_rel_to_abs']
>>> abs_boxes = annots['rel_boxes'].warp(tf_rel_to_abs)
>>> abs_ssegs = annots['rel_ssegs'].warp(tf_rel_to_abs)
>>> abs_kpts = annots['rel_kpts'].warp(tf_rel_to_abs)
>>> # Draw box in original image context
>>> kwplot.imshow(abs_frame, pnum=(1, 2, 1), fnum=1)
>>> abs_boxes.translate([-.5, -.5]).draw()
```

(continues on next page)

(continued from previous page)

```
>>> abs_kpts.draw(color='green', radius=10)
>>> abs_ssegs.draw(color='red', alpha=.5)
>>> # Draw box in relative sample context
>>> kwplot.imshow(sample['im'], pnum=(1, 2, 2), fnum=1)
>>> annots['rel_boxes'].translate([-.5, -.5]).draw()
>>> annots['rel_ssegs'].draw(color='red', alpha=.6)
>>> annots['rel_kpts'].draw(color='green', alpha=.4, radius=10)
>>> kwplot.show_if_requested()
```

Example

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('photos')
>>> target = self.regions.get_positive(1)
>>> target['window_dims'] = (300, 150)
>>> target['pad'] = None
>>> sample = self.load_sample(target)
>>> assert sample['im'].shape[0:2] == target['window_dims']
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(sample['im'], colorspace='rgb')
>>> kwplot.show_if_requested()
```

Example

```
>>> # Multispectral video sample example
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('vidshapes1-multispectral', num_frames=5)
>>> sample_grid = self.new_sample_grid('video_detection', (3, 128, 128))
>>> target = sample_grid['positives'][0]
>>> target['channels'] = 'B1|B8'
>>> target['as_xarray'] = False
>>> sample = self.load_sample(target)
>>> print(ub.repr2(sample['target'], nl=1))
>>> print(sample['im'].shape)
>>> assert sample['im'].shape == (3, 128, 128, 2)
>>> target['channels'] = '<all>'
>>> sample = self.load_sample(target)
>>> assert sample['im'].shape == (3, 128, 128, 5)
```

Example

```
>>> # Multispectral-multisensor jagged video sample example
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('vidshapes1-msi-multisensor', num_frames=5)
>>> sample_grid = self.new_sample_grid('video_detection', (3, 128, 128))
>>> target = sample_grid['positives'][0]
>>> target['channels'] = 'B1|B8'
>>> target['as_xarray'] = False
>>> sample1 = self.load_sample(target)
>>> target['scale'] = 2
>>> sample2 = self.load_sample(target)
>>> target['use_native_scale'] = True
>>> sample3 = self.load_sample(target)
>>> ####
>>> assert sample1['im'].shape == (3, 128, 128, 2)
>>> assert sample2['im'].shape == (3, 256, 256, 2)
>>> box1 = sample1['annots']['frame_dets'][0].boxes
>>> box2 = sample2['annots']['frame_dets'][0].boxes
>>> box3 = sample3['annots']['frame_dets'][0].boxes
>>> assert np.allclose((box2.width / box1.width), 2)
>>> # Jagged annotations are still in video space
>>> assert np.allclose((box3.width / box1.width), 2)
>>> jagged_shape = [[p.shape for p in f] for f in sample3['im']]
>>> jagged_align = [[a for a in m['align']] for m in sample3['params']['jagged_
↪meta']]
```

_infer_target_attributes(*target*, ***kwargs*)

Infer unpopulated target attribues

Example

```
>>> # sample using only an annotation id
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> target = {'aid': 1, 'as_xarray': True}
>>> target_ = self._infer_target_attributes(target)
>>> print('target_ = {}'.format(ub.repr2(target_, nl=1)))
>>> assert target_['gid'] == 1
>>> assert all(k in target_ for k in ['cx', 'cy', 'width', 'height'])
```

```
>>> self = CocoSampler.demo('vidshapes8-multispectral')
>>> target = {'aid': 1, 'as_xarray': True}
>>> target_ = self._infer_target_attributes(target)
>>> assert target_['gid'] == 1
>>> assert all(k in target_ for k in ['cx', 'cy', 'width', 'height'])
```

```
>>> target = {'vidid': 1, 'as_xarray': True}
>>> target_ = self._infer_target_attributes(target)
>>> print('target_ = {}'.format(ub.repr2(target_, nl=1)))
>>> assert 'gids' in target_
```

```
>>> target = {'gids': [1, 2], 'as_xarray': True}
>>> target_ = self._infer_target_attributes(target)
>>> print('target_ = {}'.format(ub.repr2(target_, nl=1)))
```

_load_slice(*target*)

Example

```
>>> # sample an out of bounds target
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> target = self.regions.get_positive(0)
>>> target = self._infer_target_attributes(target)
>>> target['as_xarray'] = True
>>> sample = self._load_slice(target)
>>> print('sample = {!r}'.format(ub.map_vals(type, sample)))
```

```
>>> # sample an out of bounds target
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('vidshapes2')
>>> target = self._infer_target_attributes({'vidid': 1})
>>> target = self._infer_target_attributes(target)
>>> target['as_xarray'] = True
>>> sample = self._load_slice(target)
>>> print('sample = {!r}'.format(ub.map_vals(type, sample)))
```

```
>>> target = self._infer_target_attributes({'gids': [1, 2, 3]})
>>> target['as_xarray'] = True
>>> sample = self._load_slice(target)
>>> print('sample = {!r}'.format(ub.map_vals(type, sample)))
```

CommandLine:

xdoctest -m ndsampler.coco_sampler CocoSampler._load_slice –profile

Ignore:

from ndsampler.coco_sampler import * # NOQA from ndsampler.coco_sampler import _center_extent_to_slice, _ensure_iterablen import ndsampler import xdev globals().update(xdev.get_func_kwargs(ndsampler.CocoSampler._load_slice))

Example

```
>>> # Multispectral video sample example
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('vidshapes1-multispectral', num_frames=5)
>>> sample_grid = self.new_sample_grid('video_detection', (3, 128, 128))
>>> target = sample_grid['positives'][0]
>>> target = self._infer_target_attributes(target)
>>> target['channels'] = 'B1|B8'
>>> target['as_xarray'] = False
>>> sample = self.load_sample(target)
```

(continues on next page)

(continued from previous page)

```
>>> print(ub.repr2(sample['target'], nl=1))
>>> print(sample['im'].shape)
>>> assert sample['im'].shape == (3, 128, 128, 2)
>>> target['channels'] = '<all>'
>>> sample = self.load_sample(target)
>>> assert sample['im'].shape == (3, 128, 128, 5)
```

Example

```
>>> # Multispectral video sample example
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo('vidshapes1-multisensor-msi', num_frames=5)
>>> sample_grid = self.new_sample_grid('video_detection', (3, 128, 128))
>>> target = sample_grid['positives'][0]
>>> target = self._infer_target_attributes(target)
>>> target['channels'] = 'B1|B8'
>>> target['as_xarray'] = False
>>> target['space_slice'] = (slice(-64, 64), slice(-64, 64))
>>> sample = self.load_sample(target)
>>> print(ub.repr2(sample['target'], nl=1))
>>> print(sample['im'].shape)
>>> assert sample['im'].shape == (3, 128, 128, 2)
>>> target['channels'] = '<all>'
>>> sample = self.load_sample(target)
>>> assert sample['im'].shape[2] > 5  # probably 16
```

```
>>> # Test jagged native scale sampling
>>> target['use_native_scale'] = True
>>> target['as_xarray'] = True
>>> target['channels'] = 'B1|B8|r|g|b|disparity|gauss'
>>> sample = self.load_sample(target)
>>> jagged_meta = sample['params']['jagged_meta']
>>> frames = sample['im']
>>> jagged_shape = [[p.shape for p in f] for f in frames]
>>> jagged_chans = [[p.coords['c'].values.tolist() for p in f] for f in frames]
>>> jagged_chans2 = [m['chans'] for m in jagged_meta]
>>> jagged_align = [[a.concise() for a in m['align']] for m in jagged_meta]
>>> # all frames should have the same number of channels
>>> assert len(frames) == 3
>>> assert all(sum(p.shape[2] for p in f) == 7 for f in frames)
>>> frames[0] == 3
>>> print('jagged_chans = {}'.format(ub.repr2(jagged_chans, nl=1)))
>>> print('jagged_shape = {}'.format(ub.repr2(jagged_shape, nl=1)))
>>> print('jagged_chans2 = {}'.format(ub.repr2(jagged_chans2, nl=1)))
>>> print('jagged_align = {}'.format(ub.repr2(jagged_align, nl=1)))
```

```
>>> # Test realigned native scale sampling
>>> target['use_native_scale'] = True
>>> target['realign_native'] = 'largest'
>>> target['as_xarray'] = True
```

(continues on next page)

(continued from previous page)

```
>>> gid = None
>>> for coco_img in self.dset.images().coco_images:
>>>     if coco_img.channels & 'r|g|b':
>>>         gid = coco_img.img['id']
>>>         break
>>> assert gid is not None, 'need specific image'
>>> target['gids'] = [gid]
>>> # Test channels that are good early fused groups
>>> target['channels'] = 'r|g|b'
>>> sample1 = self.load_sample(target)
>>> target['channels'] = 'B8|B11'
>>> sample2 = self.load_sample(target)
>>> target['channels'] = 'r|g|b|B11'
>>> sample3 = self.load_sample(target)
>>> shape1 = sample1['im'].shape[1:3]
>>> shape2 = sample2['im'].shape[1:3]
>>> shape3 = sample3['im'].shape[1:3]
>>> print(f'shape1={shape1}')
>>> print(f'shape2={shape2}')
>>> print(f'shape3={shape3}')
>>> assert shape1 != shape2
>>> assert shape2 == shape3
```

_load_slice_3d(*target*)

Breakout the 2d vs 3d logic so they can evolve somewhat independently.

TODO: the 2D logic needs to be updated to be more consistent with 3d logic

Or at least the differences between them are more clear.

_load_slice_2d(*target*)

Breakout the 2d vs 3d logic so they can evolve somewhat independently.

TODO: the 2D logic needs to be updated to be more consistent with 3d logic

Or at least the differences between them are more clear.

_populate_overlap(*sample*, *visible_thresh=0.1*, *with_annots=True*)

Add information about annotations overlapping the sample.

with_annots can be a + separated string or list of the the special keys:
'segmentation' and 'keypoints'.

Example

```
>>> # sample an out of bounds target
>>> import ndsampler
>>> self = ndsampler.CocoSampler.demo()
>>> target = self.regions.get_item(0)
>>> target = self._infer_target_attributes(target)
>>> sample = self._load_slice(target)
>>> sample = self._populate_overlap(sample)
>>> print('sample = {}'.format(ub.repr2(ub.util_dict.dict_diff(sample, ['im']),
↪ nl=-1)))
```

class ndsampler.**FrameIntersectionIndex**

Bases: ubelt.NiceRepr

Build spatial tree for each frame so we can quickly determine if a random negative is too close to a positive. For each frame/image we built a qtree.

Example

```
>>> from ndsampler.isect_indexer import *
>>> import kwcoco
>>> import ubelt as ub
>>> dset = kwcoco.CocoDataset.demo()
>>> dset._ensure_imgsize()
>>> dset.remove_annotations([ann for ann in dset.anns.values()
>>>                          if 'bbox' not in ann])
>>> # Build intersection index aroung coco dataset
>>> self = FrameIntersectionIndex.from_coco(dset)
>>> gid = 1
>>> box = kwimage.Boxes([0, 10, 100, 100], 'xywh')
>>> isect_aids, ious = self.ious(gid, box)
>>> print(ub.repr2(ious.tolist(), nl=0, precision=4))
[0.0507]
```

__nice__()

classmethod **from_coco**(*dset*, *verbose=0*)

Parameters
dset (*kwcoco.CocoDataset*) – positive annotation data

Returns
FrameIntersectionIndex

classmethod **demo**(**args*, ***kwargs*)

Create a demo intersection index.

Parameters

- ***args** – see kwcoco.CocoDataset.demo
- ****kwargs** – see kwcoco.CocoDataset.demo

Returns
FrameIntersectionIndex

static **_build_index**(*dset*, *verbose=0*)

overlapping_aids(*gid*, *box*)

Find all annotation-ids within an image that have some overlap with a bounding box.

Parameters

- **gid** (*int*) – an image id
- **box** (*kwimage.Boxes*) – the specified region

Returns
list of annotation ids

Return type
List[int]

Example

```
>>> self = FrameIntersectionIndex.demo('shapes128')
>>> for gid, qtree in self.qtrees.items():
>>>     box = kwimage.Boxes([0, 0, qtree.width, qtree.height], 'xywh')
>>>     self.overlapping_aids(gid, box)
```

ious(*gid*, *box*)

Find overlaping annotations in a specific image and their intersection over union with a a query box.

Parameters

- **gid** (*int*) – an image id
- **box** (*kwimage.Boxes*) – the specified region

Returns
isect_aids: list of annotation ids ious: jaccard score for each returned annotation id

Return type
Tuple[List[int], ndarray]

iooas(*gid*, *box*)

Intersection over other's area

Parameters

- **gid** (*int*) – an image id
- **box** (*kwimage.Boxes*) – the specified region

Like iou, but non-symetric, returned number is a percentage of the other's (groundtruth) area. This means we dont care how big the (negative) *box* is.

random_negatives(*num*, *anchors=None*, *window_size=None*, *gids=None*, *thresh=0.0*, *exact=True*, *rng=None*, *patience=None*)

Finds random boxes that don't have a large overlap with positive instances.

Parameters

- **num** (*int*) – number of negative boxes to generate (actual number of boxes returned may be less unless *exact=True*)
- **anchors** (*ndarray*) – prior normalized aspect ratios for negative boxes. Mutually exclusive with *window_size*.
- **window_size** (*ndarray*) – absolute (W, H) sizes to use for negative boxes. Mutually exclusive with *anchors*.
- **gids** (*List[int]*) – image-ids to generate negatives for, if not specified generates for all images.
- **thresh** (*float*) – overlap area threshold as a percentage of the negative box size. When thresh=0.0, that means negatives cannot overlap any positive, when threh=1.0, there are no constrains on negative placement.
- **exact** (*bool*) – if True, ensure that we generate exactly *num* boxes
- **rng** (*RandomState*) – random number generator

Example

```
>>> from ndsampler.isect_indexer import *
>>> import ndsampler
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> self = FrameIntersectionIndex.from_coco(dset)
>>> anchors = np.array([[.35, .15], [.2, .2], [.1, .1]])
>>> #num = 25
>>> num = 5
>>> rng = kwarray.ensure_rng(None)
>>> neg_gids, neg_boxes = self.random_negatives(
>>>     num, anchors, gids=[1], rng=rng, thresh=0.01, exact=1)
>>> # xdoc: +REQUIRES(--show)
>>> gid = sorted(set(neg_gids))[0]
>>> boxes = neg_boxes.compress(neg_gids == gid)
>>> import kwplot
>>> kwplot.autompl()
>>> img = kwimage.imread(dset.imgs[gid]['file_name'])
>>> kwplot.imshow(img, doclf=True, fnum=1, colorspace='bgr')
>>> support = self._support(gid)
>>> kwplot.draw_boxes(support, color='blue')
>>> kwplot.draw_boxes(boxes, color='orange')
```

Example

```
>>> from ndsampler.isect_indexer import *
>>> import kwcoco
>>> dset = kwcoco.CocoDataset.demo('shapes8')
>>> self = FrameIntersectionIndex.from_coco(dset)
>>> #num = 25
>>> num = 5
>>> rng = kwarray.ensure_rng(None)
>>> window_size = (50, 50)
>>> neg_gids, neg_boxes = self.random_negatives(
>>>     num, window_size=window_size, gids=[1], rng=rng,
>>>     thresh=0.01, exact=1)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> gid = sorted(set(neg_gids))[0]
>>> boxes = neg_boxes.compress(neg_gids == gid)
>>> img = kwimage.imread(dset.imgs[gid]['file_name'])
>>> kwplot.imshow(img, doclf=True, fnum=1, colorspace='bgr')
>>> support = self._support(gid)
>>> support.draw(color='blue')
>>> boxes.draw(color='orange')
```

_debug_index()

_support(*gid*)

class ndsampler.**DynamicToySampler**(*n_positives=100000.0*, *seed=None*, *gsize=(416, 416)*, *categories=None*)

Bases: *ndsampler.abstract_sampler.AbstractSampler*

Generates positive and negative samples on the fly.

Note: Its probably more robust to generate a static fixed-size dataset with 'demodata_toy_dset' or *kwcoco.CocoDataset.demo*. However, if you need a sampler that dynamically generates toydata, this is for you.

Ignore:

```
>>> from ndsampler.toydata import *
>>> self = DynamicToySampler()
>>> window_dims = (96, 96)
```

img, anns = self.load_positive(window_dims=window_dims) kwplot.autompl() kwplot.imshow(img['imdata'])

img, anns = self.load_negative(window_dims=window_dims) kwplot.autompl() kwplot.imshow(img['imdata'])

CommandLine:

xdoctest -m ndsampler.toydata DynamicToySampler –show

Example

```
>>> # Test that this sampler works with the dataset
>>> from ndsampler.toydata import *
>>> self = DynamicToySampler(1e3)
>>> imgs = [self.load_positive()['im'] for _ in range(9)]
>>> # xdoctest: +REQUIRES(--show)
>>> stacked = kwimage.stack_images_grid(imgs, overlap=-10)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(stacked)
>>> kwplot.show_if_requested()
```

load_item(*index*, *pad=None*, *window_dims=None*)

Loads from positives and then negatives.

__len__()

_depends()

property class_ids

property n_positives

property n_annots

property n_images

image_ids()

lookup_class_name(*class_id*)

lookup_class_id(*class_name*)

_lookup_kpnames(*class_id*)

property n_categories

preselect(*n_pos=None*, *n_neg=None*, *neg_to_pos_ratio=None*, *window_dims=None*, *rng=None*, *verbose=0*)

Setup a pool of training examples before the epoch begins

load_image(*image_id=None*, *rng=None*)

load_image_with_annots(*image_id=None*, *rng=None*)

Returns a random image and its annotations

abstract load_sample(*tr*, *pad=None*, *window_dims=None*)

_load_toy_sample(*window_dims*, *pad*, *rng*, *centerobj*, *n_annots*)

load_positive(*index=None*, *pad=None*, *window_dims=None*, *rng=None*)

Note: window_dims is height / width

Example

```
>>> from ndsampler.toydata import *
>>> self = DynamicToySampler(1e2)
>>> sample = self.load_positive()
>>> annots = sample['annots']
>>> assert len(annots['aids']) > 0
>>> assert len(annots['rel_cxywh']) == len(annots['aids'])
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> # Draw box in relative sample context
>>> kwplot.imshow(sample['im'], pnum=(1, 1, 1), fnum=1)
>>> annots['rel_boxes'].translate([-.5, -.5]).draw()
>>> annots['rel_ssegs'].draw(color='red', alpha=.6)
>>> annots['rel_kpts'].draw(color='green', alpha=.8, radius=4)
```

load_negative(*index=None*, *pad=None*, *window_dims=None*, *rng=None*)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

n

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V