ndsampler Documentation

Release 0.6.6

Jon Crall

Jan 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__(*self*)

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

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

shutdown(*self*)

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__(*self*)

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

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

shutdown(*self*)

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, compress='auto', blocksize=256)	**Parameters** • **fpath** (*PathLike*) -- file path to save the COG to.
_dtype_equality(dtype1, dtype2)	Check for numpy dtype equality
_auto_compress(src_fpath=None, data=None, data_set=None)	Heuristic for automatically choosing compression type
_cli_convert_cloud_optimized_geotiff(src_fpath, dst_fpath, compress='auto', blocksize=256)	For whatever reason using the CLI seems to simply be faster.
_api_convert_cloud_optimized_geotiff(src_fpath, dst_fpath, compress='JPEG', blocksize=256)	Optimization of imwrite specifically for converting files that already
_api_convert_cloud_optimized_geotiff2(src_fpath, dst_fpath, compress='JPEG', blocksize=256)	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='process', max_workers=0, cog_config=None)	Converts many input images to COGs and verifies that the outputs are
batch_validate_cog(dst_fpaths, mode='thread', max_workers=0)	Return cog infos
_validate_cog_worker(dst_fpath, orig_fpath=None)	
_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(*self*)

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

Ignore:

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

property ndim(*self*)

property shape(*self*)

property dtype(*self*)

__nice__(*self*)

__getitem__(*self*, *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__(*self*)
Allow this object to be passed to np.asarray

References

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

validate(*self*, *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__(*self*, *item*)

__delitem__(*self*, *key*)

__nice__(*self*)

update(*self*, *other*)

__iter__(*self*)

items(*self*)

keys(*self*)

values(*self*)

iteritems(*self*)

iterkeys(*self*)

itervalues(*self*)

clear(*self*)

__len__(*self*)

__getitem__(*self*, *key*)

__setitem__(*self*, *key*, *value*)

has_key(*self*, *item*)

set_size(*self*, *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(*cls*, *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(*self*)

_make_hashid(*self*)

property hashid(*self*)

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.

Read more in the User Guide.

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

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

split(*self*, *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=True)	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.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')
```

(continues on next page)

(continued from previous page)

```
>>> 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__(*self*)

__setstate__(*self*, *state*)

_update_backend(*self*, *backend*)
change the backend and update internals accordingly

classmethod _coerce_backend_config(*cls*, *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(*self*)
Returns the path where cached frame representations will be stored.

This will be None if there is no backend.

abstract _build_pathinfo(*self*, *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(*self*, *image_id*)

_populate_chan_info(*self*, *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(*self*)

__len__(*self*)

__getitem__(*self*, *index*)

load_region(*self*, *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(*self*, *image_id*, *cache=True*)

load_image(*self*, *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(*self*, *image_id*)
TODO: FINISHME or rename to lazy frame?

Returns a frame object that lazy loads on slice

prepare(*self*, *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
```

(continues on next page)

(continued from previous page)

```
>>> 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(*self*, *image_id*)

image_ids(*self*)

classmethod demo(*self*, ***kw*)

Get a smple frames object

_build_pathinfo(*self*, *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

---

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(*self*, *chan_name*, *cache=True*)
Load a specific auxiliary channel, optionally caching it

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

_coerce_channels(*self*, *channels=ub.NoParam*)

_load_prefused_region(*self*, *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(*self*, *img_region*, *channels=ub.NoParam*)
Loads crops from multiple channels in aligned base coordinates.

load_region(*self*, *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__(*self*, *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(*self*)

abstract lookup_class_name(*self*, *class_id*)

abstract lookup_class_id(*self*, *class_name*)

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

property n_positives(*self*)

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

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

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

abstract load_image(*self*, *image_id*)

abstract image_ids(*self*)

abstract preselect(*self*, ***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	Mixin methods for CategoryTree that specifically relate to computing

class ndsampler.category_tree.**CategoryTree**
Bases: `kwcoco.CategoryTree`, `Mixin_CategoryTree_Torch`

Mixin methods for CategoryTree that specifically relate to computing normalized probabilities.

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(*self*)

_make_hashid(*self*)

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

_build_pathinfo(*self*, *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=(300, 300), thresh=0.0, rng=None, verbose=0)	Reduce positive example redundency by selecting disparate positive samples
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'})	Create a space time-grid to sample with
new_image_sample_grid(dset, window_dims, window_overlap=0.0, classes_of_interest=None, ignore_coverage_thresh=0.6, negative_classes={'ignore', 'background'})	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(*self*, *index=None*, *rng=None*)

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

abstract overlapping_aids(*self*, *gid*, *box*)

preselect(*self*, *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(*self*)

property n_negatives(*self*)

property n_positives(*self*)

property n_samples(*self*)

property class_ids(*self*)

property image_ids(*self*)

property n_annots(*self*)

property n_images(*self*)

property n_categories(*self*)

lookup_class_name(*self*, *class_id*)

lookup_class_id(*self*, *class_name*)

__nice__(*self*)

classmethod demo(*CocoRegions*)

_make_hashid(*self*)

property isect_index(*self*)

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

_lazy_isect_index(*self*, *verbose=None*)

property targets(*self*)
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(*self*)

overlapping_aids(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*, *index*, *rng=None*)

Loads from positives and then negatives.

_random_negatives(*self*, *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(*self*, *task*, *window_dims*, *window_overlap=0*)

New experimental method to replace preselect positives / negatives

Parameters **task** (*str*) – can be video_detection # image_detection # video_classification # image_classification

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(*self*, *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(*self*, *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(*self*, *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'}*)

Create a space time-grid to sample with

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)
```

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'}*)

Create a space time-grid to sample with

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_grid = new_image_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['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

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'),
>>> ]  # xdoc: +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 (and possibly annotations)
>>> dataset = {
>>>     'images': [{'id': i, 'file_name': fpath} for i, fpath in enumerate(image_paths)],
>>>     'annotations': [],
>>>     'categories': [],
>>> }
>>> coco_dset = kwcoco.CocoDataset(dataset)
>>> print(coco_dset)
<CocoDataset(tag=None, n_anns=0, n_imgs=3, ...n_cats=0)>
>>> # 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
```

(continues on next page)

(continued from previous page)

```
>>> # with an image_id (gid) center location (cx, cy) and width, height.
>>> target = {'gid': 0, 'cx': 200, 'cy': 200, 'width': 100, 'height': 100}
>>> 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', 'tr']
>>> im = sample['im']
>>> print(im.shape)
(100, 100, 3)
>>> # 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)	

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

```
>>> 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(*cls*, *key='shapes'*, *workdir=None*, *backend=None*, ***kw*)

Create a toy coco sampler for testing and demo puposes

SeeAlso:

- kwcoco.CocoDataset.demo

_init(*self*)

property classes(*self*)

property catgraph(*self*)

DEPRICATED, use self.classes instead

_depends(*self*)

lookup_class_name(*self*, *class_id*)

lookup_class_id(*self*, *class_name*)

property n_positives(*self*)

property **n_annots**(*self*)

property **n_samples**(*self*)

__len__(*self*)

property **n_images**(*self*)

property **n_categories**(*self*)

property **class_ids**(*self*)

property **image_ids**(*self*)

preselect(*self*, ***kwargs*)
Setup a pool of training examples before the epoch begins

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

load_image_with_annots(*self*, *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

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> rng = None
>>> img, anns = self.load_image_with_annots(1)
>>> dets = kwimage.Detections.from_coco_annots(anns, dset=self.dset)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'][:])
>>> dets.draw()
>>> kwplot.show_if_requested()
```

load_annotations(*self*, *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(*self*, *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`(*self*, *index*, *pad=None*, *window_dims=None*, *with_annots=True*)

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
- **pad** (*tuple*) – (height, width) extra context to add to each size. This helps prevent augmentation from producing boundary effects
- **window_dims** (*tuple*) – (height, width) area around the center of the target region to sample.
- **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.

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

`load_positive`(*self*, *index=None*, *with_annots=True*, *tr=None*, *pad=None*, *rng=None*, ***kw*)

Load an item from the the positive pool of regions.

Parameters

- **index** (*int*) – index of positive target
- **pad** (*tuple*) – (height, width) extra context to add to each size. This helps prevent augmentation from producing boundary effects
- **tr** (*Dict*) – Extra target arguments like window_dims.
- **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.

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

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> rng = None
>>> sample = self.load_positive(pad=(10, 10), tr=dict(window_dims=(3, 3)))
>>> assert sample['im'].shape[0] == 23
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(sample['im'])
>>> kwplot.show_if_requested()
```

load_negative(*self*, *index=None*, *with_annots=True*, *tr=None*, *pad=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.
- **tr** (*Dict*) – Extra target arguments like window_dims.
- **pad** (*tuple*) – (height, width) extra context to add to each size. This helps prevent augmentation from producing boundary effects

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

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> rng = None
>>> sample = self.load_negative(rng=rng, pad=(0, 0))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> box = kwimage.Boxes(tr.reindex(['rel_cx', 'rel_cy', 'width', 'height']).
↪values, 'cxywh')
>>> kwplot.imshow(sample)
>>> kwplot.draw_boxes(box)
>>> kwplot.show_if_requested()
```

Example

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> rng = None
>>> sample = self.load_negative(rng=rng, pad=(0, 0), tr=dict(window_dims=(64,␣
↪64)))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> box = kwimage.Boxes(tr.reindex(['rel_cx', 'rel_cy', 'width', 'height']).
↪values, 'cxywh')
>>> kwplot.imshow(sample, fnum=1, doclf=True)
>>> kwplot.draw_boxes(box)
>>> kwplot.show_if_requested()
```

load_sample(*self*, *tr*, *with_annots=True*, *visible_thresh=0.0*, *pad=None*, *padkw={'mode': 'constant'}*, *dtype=None*, *nodata=None*)

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

Parameters

- **tr** (*dict*) – target dictionary 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, tr must contain or be able to infer the key *gid (int)*, to specify an image id.

  For 3D video source objects, tr 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

- **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.
- **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*
- **dtype** (*type | None*) – Cast the loaded data to this type. If unspecified returns the data as-is.
- **nodata** (*int | 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.

Returns

sample: dict containing keys im (ndarray | DataArray): image / video 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): 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

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> # The target (tr) lets you specify an arbitrary window
>>> tr = {'gid': 1, 'cx': 5, 'cy': 2, 'width': 6, 'height': 6}
>>> sample = self.load_sample(tr)
...
>>> 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
>>> tr = {'gid': 1, 'cx': 5, 'cy': 2, 'width': 600, 'height': 600}
>>> sample = sampler.load_sample(tr)
>>> annotation_ids = sample['annots']['aids']
>>> aid = annotation_ids[0]
>>> # 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

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> tr = self.regions.get_positive(0)
>>> pad = (25, 25)
>>> tr['window_dims'] = 'square'
>>> sample = self.load_sample(tr, pad=pad)
>>> print('im.shape = {!r}'.format(sample['im'].shape))
im.shape = (135, 135, 3)
>>> pad = (0, 0)
>>> tr['window_dims'] = None
>>> sample = self.load_sample(tr, pad=pad)
>>> print('im.shape = {!r}'.format(sample['im'].shape))
im.shape = (52, 85, 3)
>>> # xdoc: +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()
>>> tr = self.regions.get_positive(0)
>>> tr['window_dims'] = (364, 364)
>>> sample = self.load_sample(tr)
>>> annots = sample['annots']
>>> assert len(annots['aids']) > 0
>>> #assert len(annots['rel_cxywh']) == len(annots['aids'])
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> abs_frame = self.frames.load_image(sample['tr']['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()
>>> 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')
>>> tr = self.regions.get_positive(1)
>>> pad = None
>>> tr['window_dims'] = (300, 150)
>>> sample = self.load_sample(tr, pad)
>>> assert sample['im'].shape[0:2] == tr['window_dims']
>>> # xdoc: +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))
>>> tr = sample_grid['positives'][0]
>>> tr['channels'] = 'B1|B8'
>>> tr['as_xarray'] = False
>>> sample = self.load_sample(tr)
>>> print(ub.repr2(sample['tr'], nl=1))
>>> print(sample['im'].shape)
>>> assert sample['im'].shape == (3, 128, 128, 2)
>>> tr['channels'] = '<all>'
>>> sample = self.load_sample(tr)
>>> assert sample['im'].shape == (3, 128, 128, 5)
```

_infer_target_attributes(*self*, *tr*)

Infer unpopulated target attribues

Example

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

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

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

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

_load_slice(*self*, *tr*, *pad=None*, *padkw={'mode': 'constant'}*, *dtype=None*, *nodata=None*)

Example

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

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

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

CommandLine: xdoctest -m /home/joncrall/code/ndsampler/ndsampler/coco_sampler.py CocoSampler._load_slice –profile

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))
>>> tr = sample_grid['positives'][0]
>>> tr['channels'] = 'B1|B8'
>>> tr['as_xarray'] = False
>>> sample = self.load_sample(tr)
>>> print(ub.repr2(sample['tr'], nl=1))
>>> print(sample['im'].shape)
>>> assert sample['im'].shape == (3, 128, 128, 2)
>>> tr['channels'] = '<all>'
>>> sample = self.load_sample(tr)
>>> assert sample['im'].shape == (3, 128, 128, 5)
```

_populate_overlap(*self*, *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
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> tr = self.regions.get_item(0)
>>> sample = self._load_slice(tr)
>>> 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.coerce_data

Moved to netharn

Module Contents

Functions

coerce_datasets(config, build_hashid=False, verbose=1)	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=3)	**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'),
```

(continues on next page)

(continued from previous page)

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

DEPRECATD. THIS IS BEING MOVED TO KWCOCO FOR DEVELOPMENT AND EVENTUALLY WILL LIVE IN KWIMAGE.

The classes in this file represent a tree of delayed operations.

Proof of concept for delayed chainable transforms in Python.

There are several optimizations that could be applied.

This is similar to GDAL's virtual raster table, but it works in memory and I think it is easier to chain operations.

SeeAlso: ../dev/symbolic_delayed.py

Concepts:

> Each class should be a layer that adds a new transformation on top of underlying nested layers. Adding new layers should be quick, and there should always be the option to "finalize" a stack of layers, chaining the transforms / operations and then applying one final efficient transform at the end.

Conventions:

- dsize = (always in width / height), no channels are present
- shape for images is always (height, width, channels)

- channels are always the last dimension of each image, if no channel dim is specified, finalize will add it.
- **Videos must be the last process in the stack, and add a leading** time dimension to the shape. dsize is still width, height, but shape is now: (time, height, width, chan)

Example

```
>>> # Example demonstrating the modivating use case
>>> # We have multiple aligned frames for a video, but each of
>>> # those frames is in a different resolution. Furthermore,
>>> # each of the frames consists of channels in different resolutions.
>>> from ndsampler.delayed import *  # NOQA
>>> # Create raw channels in some "native" resolution for frame 1
>>> f1_chan1 = DelayedIdentity.demo('astro', chan=0, dsize=(300, 300))
>>> f1_chan2 = DelayedIdentity.demo('astro', chan=1, dsize=(200, 200))
>>> f1_chan3 = DelayedIdentity.demo('astro', chan=2, dsize=(10, 10))
>>> # Create raw channels in some "native" resolution for frame 2
>>> f2_chan1 = DelayedIdentity.demo('carl', dsize=(64, 64), chan=0)
>>> f2_chan2 = DelayedIdentity.demo('carl', dsize=(260, 260), chan=1)
>>> f2_chan3 = DelayedIdentity.demo('carl', dsize=(10, 10), chan=2)
>>> #
>>> # Delayed warp each channel into its "image" space
>>> # Note: the images never actually enter this space we transform through it
>>> f1_dsize = np.array((3, 3))
>>> f2_dsize = np.array((2, 2))
>>> f1_img = DelayedChannelConcat([
>>>     f1_chan1.delayed_warp(Affine.scale(f1_dsize / f1_chan1.dsize), dsize=f1_dsize),
>>>     f1_chan2.delayed_warp(Affine.scale(f1_dsize / f1_chan2.dsize), dsize=f1_dsize),
>>>     f1_chan3.delayed_warp(Affine.scale(f1_dsize / f1_chan3.dsize), dsize=f1_dsize),
>>> ])
>>> f2_img = DelayedChannelConcat([
>>>     f2_chan1.delayed_warp(Affine.scale(f2_dsize / f2_chan1.dsize), dsize=f2_dsize),
>>>     f2_chan2.delayed_warp(Affine.scale(f2_dsize / f2_chan2.dsize), dsize=f2_dsize),
>>>     f2_chan3.delayed_warp(Affine.scale(f2_dsize / f2_chan3.dsize), dsize=f2_dsize),
>>> ])
>>> # Combine frames into a video
>>> vid_dsize = np.array((280, 280))
>>> vid = DelayedFrameConcat([
>>>     f1_img.delayed_warp(Affine.scale(vid_dsize / f1_img.dsize), dsize=vid_dsize),
>>>     f2_img.delayed_warp(Affine.scale(vid_dsize / f2_img.dsize), dsize=vid_dsize),
>>> ])
>>> vid.nesting
>>> print('vid.nesting = {}'.format(ub.repr2(vid.nesting(), nl=-1)))
>>> final = vid.finalize(interpolation='nearest')
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final[0], pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(final[1], pnum=(1, 2, 2), fnum=1)
```

Module Contents

Classes

DelayedOperation	Base class for nodes in a tree of delayed operations
DelayedVideoOperation	Base class for nodes in a tree of delayed operations
DelayedImageOperation	Operations that pertain only to images
DelayedIdentity	Noop leaf that does nothing. Mostly used in tests atm
DelayedLoad	

Example

DelayedFrameConcat	Represents multiple frames in a video
DelayedChannelConcat	Represents multiple channels in an image that could be concatenated
DelayedWarp	POC for chainable transforms
DelayedCrop	Represent a delayed crop operation

Functions

_compute_leaf_subcrop(root_region_bounds, tf_leaf_to_root)	Given a region in a "root" image and a trasnform between that "root" and
_largest_shape(shapes)	Finds maximum over all shapes
_devcheck_corner()	

class ndsampler.delayed.**DelayedOperation**

Bases: ubelt.NiceRepr

Base class for nodes in a tree of delayed operations

__nice__(*self*)

abstract finalize(*self*)

abstract children(*self*)

Abstract method, which should generate all of the direct children of a node in the operation tree.

_optimize_paths(*self*, ***kwargs*)

Iterate through the leaf nodes, which are virtually transformed into the root space.

This returns some sort of hueristically optimized leaf repr wrt warps.

nesting(*self*)

class ndsampler.delayed.**DelayedVideoOperation**

Bases: *DelayedOperation*

Base class for nodes in a tree of delayed operations

class ndsampler.delayed.**DelayedImageOperation**

Bases: *DelayedOperation*

Operations that pertain only to images

delayed_crop(*self*, *region_slices*)

Create a new delayed image that performs a crop in the transformed "self" space.

Parameters **region_slices** (*Tuple[slice, slice]*) – y-slice and x-slice.

Notes

Returns a heuristically "simplified" tree. In the current implementation there are only 3 operations, cat, warp, and crop. All cats go at the top, all crops go at the bottom, all warps are in the middle.

Returns lazy executed delayed transform

Return type *DelayedWarp*

Example

```
>>> from ndsampler.delayed import *  # NOQA
>>> dsize = (100, 100)
>>> tf2 = Affine.affine(scale=3).matrix
>>> self = DelayedWarp(np.random.rand(33, 33), tf2, dsize)
>>> region_slices = (slice(5, 10), slice(1, 12))
>>> delayed_crop = self.delayed_crop(region_slices)
>>> print(ub.repr2(delayed_crop.nesting(), nl=-1, sort=0))
>>> delayed_crop.finalize()
```

Example

```
>>> chan1 = DelayedLoad.demo('astro')
>>> chan2 = DelayedLoad.demo('carl')
>>> warped1a = chan1.delayed_warp(Affine.scale(1.2).matrix)
>>> warped2a = chan2.delayed_warp(Affine.scale(1.5))
>>> warped1b = warped1a.delayed_warp(Affine.scale(1.2).matrix)
>>> warped2b = warped2a.delayed_warp(Affine.scale(1.5))
>>> #
>>> region_slices = (slice(97, 677), slice(5, 691))
>>> self = warped2b
>>> #
>>> crop1 = warped1b.delayed_crop(region_slices)
>>> crop2 = warped2b.delayed_crop(region_slices)
>>> print(ub.repr2(warped1b.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(warped2b.nesting(), nl=-1, sort=0))
>>> # Notice how the crop merges the two nesting layers
>>> # (via the hueristic optimize step)
>>> print(ub.repr2(crop1.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(crop2.nesting(), nl=-1, sort=0))
>>> frame1 = crop1.finalize(dsize=(500, 500))
>>> frame2 = crop2.finalize(dsize=(500, 500))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(frame1, pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(frame2, pnum=(1, 2, 2), fnum=1)
```

delayed_warp(*self*, *transform*, *dsize=None*)
Delayedly transform the underlying data.

Note: this deviates from kwimage warp functions because instead of "output_dims" (specified in c-style shape) we specify dsize (w, h).

Returns new delayed transform a chained transform

Return type *DelayedWarp*

class ndsampler.delayed.**DelayedIdentity**(*sub_data*)
Bases: *DelayedImageOperation*

Noop leaf that does nothing. Mostly used in tests atm

DelayedIdentity.demo('astro', chan=0, dsize=(32, 32))

__hack_dont_optimize__ = True

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

children(*self*)
Abstract method, which should generate all of the direct children of a node in the operation tree.

finalize(*self*)

class ndsampler.delayed.**DelayedLoad**(*fpath*, *dsize=None*, *channels=None*)
Bases: *DelayedImageOperation*

Example

```
>>> fpath = kwimage.grab_test_image_fpath()
>>> self = DelayedLoad(fpath)
>>> print('self = {!r}'.format(self))
>>> self.load_shape()
>>> print('self = {!r}'.format(self))
```

```
>>> f1_img = DelayedLoad.demo('astro', dsize=(300, 300))
>>> f2_img = DelayedLoad.demo('carl', dsize=(256, 320))
>>> print('f1_img = {!r}'.format(f1_img))
>>> print('f2_img = {!r}'.format(f2_img))
>>> print(f2_img.finalize().shape)
>>> print(f1_img.finalize().shape)
```

__hack_dont_optimize__ = True

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

abstract classmethod coerce(*cls*, *data*)

children(*self*)
Abstract method, which should generate all of the direct children of a node in the operation tree.

_optimize_paths(*self*, ***kwargs*)
Iterate through the leaf nodes, which are virtually transformed into the root space.

This returns some sort of hueristically optimized leaf repr wrt warps.

load_shape(*self*)

property shape(*self*)

property num_bands(*self*)

property dsize(*self*)

property channels(*self*)

property fpath(*self*)

finalize(*self*, ***kwargs*)

class ndsampler.delayed.**DelayedFrameConcat**(*frames*, *dsize=None*)

Bases: *DelayedVideoOperation*

Represents multiple frames in a video

Notes

Video[0]:

Frame[0]: Chan[0]: (32) +————————————+ Chan[1]: (16) +————————+ Chan[2]: (8) +————+

Frame[1]: Chan[0]: (30) +————————————+ Chan[1]: (14) +————————+ Chan[2]: (6) +————+

Todo:

- [ ] Support computing the transforms when none of the data is loaded

---

Example

```
>>> # Simpler case with fewer nesting levels
>>> from ndsampler.delayed import *  # NOQA
>>> rng = kwarray.ensure_rng(None)
>>> # Delayed warp each channel into its "image" space
>>> # Note: the images never enter the space we transform through
>>> f1_img = DelayedLoad.demo('astro', (300, 300))
>>> f2_img = DelayedLoad.demo('carl', (256, 256))
>>> # Combine frames into a video
>>> vid_dsize = np.array((100, 100))
>>> self = vid = DelayedFrameConcat([
>>>     f1_img.delayed_warp(Affine.scale(vid_dsize / f1_img.dsize)),
>>>     f2_img.delayed_warp(Affine.scale(vid_dsize / f2_img.dsize)),
>>> ], dsize=vid_dsize)
>>> print(ub.repr2(vid.nesting(), nl=-1, sort=0))
>>> final = vid.finalize(interpolation='nearest', dsize=(32, 32))
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(final[0], pnum=(1, 2, 1), fnum=1)
>>> kwplot.imshow(final[1], pnum=(1, 2, 2), fnum=1)
>>> region_slices = (slice(0, 90), slice(30, 60))
```

children(*self*)
Abstract method, which should generate all of the direct children of a node in the operation tree.

property shape(*self*)

finalize(*self*, ***kwargs*)
Execute the final transform

delayed_crop(*self*, *region_slices*)

Example

```
>>> from ndsampler.delayed import *  # NOQA
>>> # Create raw channels in some "native" resolution for frame 1
>>> f1_chan1 = DelayedIdentity.demo('astro', chan=(1, 0), dsize=(300, 300))
>>> f1_chan2 = DelayedIdentity.demo('astro', chan=2, dsize=(10, 10))
>>> # Create raw channels in some "native" resolution for frame 2
>>> f2_chan1 = DelayedIdentity.demo('carl', dsize=(64, 64), chan=(1, 0))
>>> f2_chan2 = DelayedIdentity.demo('carl', dsize=(10, 10), chan=2)
>>> #
>>> f1_dsize = np.array(f1_chan1.dsize)
>>> f2_dsize = np.array(f2_chan1.dsize)
>>> f1_img = DelayedChannelConcat([
>>>     f1_chan1.delayed_warp(Affine.scale(f1_dsize / f1_chan1.dsize),␣
↪dsize=f1_dsize),
>>>     f1_chan2.delayed_warp(Affine.scale(f1_dsize / f1_chan2.dsize),␣
↪dsize=f1_dsize),
>>> ])
>>> f2_img = DelayedChannelConcat([
>>>     f2_chan1.delayed_warp(Affine.scale(f2_dsize / f2_chan1.dsize),␣
↪dsize=f2_dsize),
>>>     f2_chan2.delayed_warp(Affine.scale(f2_dsize / f2_chan2.dsize),␣
↪dsize=f2_dsize),
>>> ])
>>> vid_dsize = np.array((280, 280))
>>> full_vid = DelayedFrameConcat([
>>>     f1_img.delayed_warp(Affine.scale(vid_dsize / f1_img.dsize), dsize=vid_
↪dsize),
>>>     f2_img.delayed_warp(Affine.scale(vid_dsize / f2_img.dsize), dsize=vid_
↪dsize),
>>> ])
>>> region_slices = (slice(80, 200), slice(80, 200))
>>> crop_vid = full_vid.delayed_crop(region_slices)
>>> print(ub.repr2(full_vid.nesting(), nl=-1, sort=0))
>>> final_full = full_vid.finalize(interpolation='nearest')
>>> final_crop = crop_vid.finalize(interpolation='nearest')
>>> import pytest
>>> with pytest.raises(ValueError):
>>>     # should not be able to crop a crop yet
>>>     crop_vid.delayed_crop(region_slices)
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
```

(continues on next page)

(continued from previous page)

```
>>> kwplot.imshow(final_full[0], pnum=(2, 2, 1), fnum=1)
>>> kwplot.imshow(final_full[1], pnum=(2, 2, 2), fnum=1)
>>> kwplot.imshow(final_crop[0], pnum=(2, 2, 3), fnum=1)
>>> kwplot.imshow(final_crop[1], pnum=(2, 2, 4), fnum=1)
```

class ndsampler.delayed.**DelayedChannelConcat**(*components*, *dsize=None*)

Bases: *DelayedImageOperation*

Represents multiple channels in an image that could be concatenated

Variables **components** (*List[DelayedWarp]*) – a list of stackable channels. Each component may be comprised of multiple channels.

Todo:

- [ ] can this be generalized into a delayed concat?
- [ ] can all concats be delayed until the very end?

Example

```
>>> comp1 = DelayedWarp(np.random.rand(11, 7))
>>> comp2 = DelayedWarp(np.random.rand(11, 7, 3))
>>> comp3 = DelayedWarp(
>>>     np.random.rand(3, 5, 2),
>>>     transform=Affine.affine(scale=(7/5, 11/3)).matrix,
>>>     dsize=(7, 11)
>>> )
>>> components = [comp1, comp2, comp3]
>>> chans = DelayedChannelConcat(components)
>>> final = chans.finalize()
>>> assert final.shape == chans.shape
>>> assert final.shape == (11, 7, 6)
```

```
>>> # We should be able to nest DelayedChannelConcat inside virutal images
>>> frame1 = DelayedWarp(
>>>     chans, transform=Affine.affine(scale=2.2).matrix,
>>>     dsize=(20, 26))
>>> frame2 = DelayedWarp(
>>>     np.random.rand(3, 3, 6), dsize=(20, 26))
>>> frame3 = DelayedWarp(
>>>     np.random.rand(3, 3, 6), dsize=(20, 26))
```

```
>>> print(ub.repr2(frame1.nesting(), nl=-1, sort=False))
>>> frame1.finalize()
>>> vid = DelayedFrameConcat([frame1, frame2, frame3])
>>> print(ub.repr2(vid.nesting(), nl=-1, sort=False))
```

children(*self*)

Abstract method, which should generate all of the direct children of a node in the operation tree.

classmethod random(*cls*, *num_parts=3*, *rng=None*)

CommandLine: xdoctest -m ndsampler.delayed DelayedWarp.random

Example

```
>>> self = DelayedChannelConcat.random()
>>> print('self = {!r}'.format(self))
>>> print(ub.repr2(self.nesting(), nl=-1, sort=0))
```

property shape(*self*)

finalize(*self*, ***kwargs*)
Execute the final transform

class ndsampler.delayed.**DelayedWarp**(*sub_data*, *transform=None*, *dsize=None*)
Bases: *DelayedImageOperation*

POC for chainable transforms

Notes

"sub" is used to refer to the underlying data in its native coordinates and resolution.

"self" is used to refer to the data in the transformed coordinates that are exposed by this class.

Variables

- **sub_data** (DelayedWarp | *ArrayLike*) – array-like image data at a naitive resolution
- **transform** (*Transform*) – transforms data from native "sub"-image-space to "self"-image-space.

Example

```
>>> from ndsampler.delayed import *  # NOQA
>>> dsize = (12, 12)
>>> tf1 = np.array([[2, 0, 0], [0, 2, 0], [0, 0, 1]])
>>> tf2 = np.array([[3, 0, 0], [0, 3, 0], [0, 0, 1]])
>>> tf3 = np.array([[4, 0, 0], [0, 4, 0], [0, 0, 1]])
>>> band1 = DelayedWarp(np.random.rand(6, 6), tf1, dsize)
>>> band2 = DelayedWarp(np.random.rand(4, 4), tf2, dsize)
>>> band3 = DelayedWarp(np.random.rand(3, 3), tf3, dsize)
>>> #
>>> # Execute a crop in a one-level transformed space
>>> region_slices = (slice(5, 10), slice(0, 12))
>>> delayed_crop = band2.delayed_crop(region_slices)
>>> final_crop = delayed_crop.finalize()
>>> #
>>> # Execute a crop in a nested transformed space
>>> tf4 = np.array([[1.5, 0, 0], [0, 1.5, 0], [0, 0, 1]])
>>> chained = DelayedWarp(band2, tf4, (18, 18))
>>> delayed_crop = chained.delayed_crop(region_slices)
```

(continues on next page)

(continued from previous page)

```
>>> final_crop = delayed_crop.finalize()
>>> #
>>> tf4 = np.array([[.5, 0, 0], [0, .5, 0], [0, 0, 1]])
>>> chained = DelayedWarp(band2, tf4, (6, 6))
>>> delayed_crop = chained.delayed_crop(region_slices)
>>> final_crop = delayed_crop.finalize()
>>> #
>>> region_slices = (slice(1, 5), slice(2, 4))
>>> delayed_crop = chained.delayed_crop(region_slices)
>>> final_crop = delayed_crop.finalize()
```

Example

```
>>> dsize = (17, 12)
>>> tf = np.array([[5.2, 0, 1.1], [0, 3.1, 2.2], [0, 0, 1]])
>>> self = DelayedWarp(np.random.rand(3, 5, 13), tf, dsize=dsize)
>>> self.finalize().shape
```

property channels(*self*)

classmethod random(*cls*, *nesting=(2, 5)*, *rng=None*)

CommandLine: xdoctest -m ndsampler.delayed DelayedWarp.random

Example

```
>>> from ndsampler.delayed import *  # NOQA
>>> self = DelayedWarp.random(nesting=(4, 7))
>>> print('self = {!r}'.format(self))
>>> print(ub.repr2(self.nesting(), nl=-1, sort=0))
```

children(*self*)

Abstract method, which should generate all of the direct children of a node in the operation tree.

property shape(*self*)

_optimize_paths(*self*, ***kwargs*)

Example

```
>>> from ndsampler.delayed import *  # NOQA
>>> self = DelayedWarp.random()
>>> leafs = list(self._optimize_paths())
>>> print('leafs = {!r}'.format(leafs))
```

finalize(*self*, *transform=None*, *dsize=None*, *interpolation='linear'*, ***kwargs*)

Execute the final transform

Can pass a parent transform to augment this underlying transform.

Parameters

- **transform** (*Transform*) – an additional transform to perform
- **dsize** (*Tuple[int, int]*) – overrides destination canvas size

Example

```
>>> from ndsampler.delayed import *  # NOQA
>>> tf = np.array([[0.9, 0, 3.9], [0, 1.1, -.5], [0, 0, 1]])
>>> raw = kwimage.grab_test_image(dsize=(54, 65))
>>> raw = kwimage.ensure_float01(raw)
>>> # Test nested finalize
>>> layer1 = raw
>>> num = 10
>>> for _ in range(num):
...     layer1  = DelayedWarp(layer1, tf, dsize='auto')
>>> final1 = layer1.finalize()
>>> # Test non-nested finalize
>>> layer2 = list(layer1._optimize_paths())[0]
>>> final2 = layer2.finalize()
>>> #
>>> print(ub.repr2(layer1.nesting(), nl=-1, sort=0))
>>> print(ub.repr2(layer2.nesting(), nl=-1, sort=0))
>>> print('final1 = {!r}'.format(final1))
>>> print('final2 = {!r}'.format(final2))
>>> print('final1.shape = {!r}'.format(final1.shape))
>>> print('final2.shape = {!r}'.format(final2.shape))
>>> assert np.allclose(final1, final2)
>>> #
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(raw, pnum=(1, 3, 1), fnum=1)
>>> kwplot.imshow(final1, pnum=(1, 3, 2), fnum=1)
>>> kwplot.imshow(final2, pnum=(1, 3, 3), fnum=1)
>>> kwplot.show_if_requested()
```

Example

```
>>> # Test aliasing
>>> from ndsampler.delayed import *  # NOQA
>>> s = DelayedIdentity.demo()
>>> s = DelayedIdentity.demo('checkerboard')
>>> a = s.delayed_warp(Affine.scale(0.05), dsize='auto')
>>> b = s.delayed_warp(Affine.scale(3), dsize='auto')
```

```
>>> # xdoctest: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> # It looks like downsampling linear and area is the same
```

(continues on next page)

(continued from previous page)

```
>>> # Does warpAffine have no alias handling?
>>> pnum_ = kwplot.PlotNums(nRows=2, nCols=4)
>>> kwplot.imshow(a.finalize(interpolation='area'), pnum=pnum_(), title=
↪'warpAffine area')
>>> kwplot.imshow(a.finalize(interpolation='linear'), pnum=pnum_(), title=
↪'warpAffine linear')
>>> kwplot.imshow(a.finalize(interpolation='nearest'), pnum=pnum_(), title=
↪'warpAffine nearest')
>>> kwplot.imshow(a.finalize(interpolation='nearest', antialias=False),␣
↪pnum=pnum_(), title='warpAffine nearest AA=0')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'area'), pnum=pnum_(), title='resize area')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'linear'), pnum=pnum_(), title='resize linear')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'nearest'), pnum=pnum_(), title='resize nearest')
>>> kwplot.imshow(kwimage.imresize(s.finalize(), dsize=a.dsize, interpolation=
↪'cubic'), pnum=pnum_(), title='resize cubic')
```

class ndsampler.delayed.**DelayedCrop**(*sub_data*, *sub_slices*)

Bases: *DelayedImageOperation*

Represent a delayed crop operation

Example

```
>>> from ndsampler.delayed import *  # NOQA
>>> sub_data = DelayedLoad.demo()
>>> sub_slices = (slice(5, 10), slice(1, 12))
>>> self = DelayedCrop(sub_data, sub_slices)
>>> print(ub.repr2(self.nesting(), nl=-1, sort=0))
>>> final = self.finalize()
>>> print('final.shape = {!r}'.format(final.shape))
```

Example

```
>>> from ndsampler.delayed import *  # NOQA
>>> sub_data = DelayedLoad.demo()
>>> sub_slices = (slice(5, 10), slice(1, 12))
>>> crop1 = DelayedCrop(sub_data, sub_slices)
>>> import pytest
>>> # Should only error while huristics are in use.
>>> with pytest.raises(ValueError):
>>>     crop2 = DelayedCrop(crop1, sub_slices)
```

__hack_dont_optimize__ = True

property channels(*self*)

children(*self*)

Abstract method, which should generate all of the direct children of a node in the operation tree.

finalize(*self*, ***kwargs*)

abstract _optimize_paths(*self*, ***kwargs*)

Iterate through the leaf nodes, which are virtually transformed into the root space.

This returns some sort of hueristically optimized leaf repr wrt warps.

ndsampler.delayed.**_compute_leaf_subcrop**(*root_region_bounds*, *tf_leaf_to_root*)

Given a region in a "root" image and a trasnform between that "root" and some "leaf" image, compute the appropriate quantized region in the "leaf" image and the adjusted transformation between that root and leaf.

Example

```
>>> region_slices = (slice(33, 100), slice(22, 62))
>>> region_shape = (100, 100, 1)
>>> root_region_box = kwimage.Boxes.from_slice(region_slices, shape=region_shape)
>>> root_region_bounds = root_region_box.to_polygons()[0]
>>> tf_leaf_to_root = Affine.affine(scale=7).matrix
>>> slices, tf_new = _compute_leaf_subcrop(root_region_bounds, tf_leaf_to_root)
>>> print('tf_new =\n{!r}'.format(tf_new))
>>> print('slices = {!r}'.format(slices))
```

Ignore:

root_region_bounds = kwimage.Coords.random(4) tf_leaf_to_root = np.eye(3) tf_leaf_to_root[0, 2] = -1e-11

ndsampler.delayed.**_largest_shape**(*shapes*)

Finds maximum over all shapes

Example

```
>>> shapes = [
>>>     (10, 20), None, (None, 30), (40, 50, 60, None), (100,)
>>> ]
>>> largest = _largest_shape(shapes)
>>> print('largest = {!r}'.format(largest))
>>> assert largest == (100, 50, 60, None)
```

ndsampler.delayed.**_devcheck_corner**()

ndsampler.frame_cache

Tools for caching intermediate frame representations.

Module Contents

Functions

_cog_cache_write(gpath, cache_gpath, config=None)	CommandLine:
_npy_cache_write(gpath, cache_gpath, config=None)	
_locked_cache_write(_write_func, gpath, cache_gpath, config=None)	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, hack_use_cli=True)	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.kw
_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
>>> 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__(*self*)

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

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

Returns FrameIntersectionIndex

classmethod demo(*cls*, **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(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*)

_support(*self*, *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(*self*, *index*, *pad=None*, *window_dims=None*)

Loads from positives and then negatives.

__len__(*self*)

_depends(*self*)

property class_ids(*self*)

property n_positives(*self*)

property n_annots(*self*)

property n_images(*self*)

image_ids(*self*)

lookup_class_name(*self*, *class_id*)

lookup_class_id(*self*, *class_name*)

_lookup_kpnames(*self*, *class_id*)

property n_categories(*self*)

preselect(*self*, *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(*self*, *image_id=None*, *rng=None*)

load_image_with_annots(*self*, *image_id=None*, *rng=None*)
Returns a random image and its annotations

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

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

load_positive(*self*, *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(*self*, *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	Mixin methods for CategoryTree that specifically relate to computing
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=(300, 300), thresh=0.0, rng=None, verbose=0)	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(*self*)

_make_hashid(*self*)

property hashid(*self*)

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'):
>>>     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__(*self*)

__setstate__(*self*, *state*)

_update_backend(*self*, *backend*)
change the backend and update internals accordingly

classmethod _coerce_backend_config(*cls*, *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(*self*)
Returns the path where cached frame representations will be stored.

This will be None if there is no backend.

abstract _build_pathinfo(*self*, *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(*self*, *image_id*)

_populate_chan_info(*self*, *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(*self*)

__len__(*self*)

__getitem__(*self*, *index*)

load_region(*self*, *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(*self*, *image_id*, *cache=True*)

load_image(*self*, *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(*self*, *image_id*)

TODO: FINISHME or rename to lazy frame?

Returns a frame object that lazy loads on slice

prepare(*self*, *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(*self*, *image_id*)

image_ids(*self*)

classmethod **demo**(*self*, ***kw*)

Get a smple frames object

_build_pathinfo(*self*, *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(*self*)

abstract lookup_class_name(*self*, *class_id*)

abstract lookup_class_id(*self*, *class_name*)

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

property n_positives(*self*)

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

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

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

abstract load_image(*self*, *image_id*)

abstract image_ids(*self*)

abstract preselect(*self*, ***kwargs*)

Setup a pool of training examples before the epoch begins

class ndsampler.**CategoryTree**

Bases: kwcoco.CategoryTree, Mixin_CategoryTree_Torch

Mixin methods for CategoryTree that specifically relate to computing normalized probabilities.

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(*self*)

_make_hashid(*self*)

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

_build_pathinfo(*self*, *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(*self*)

property n_negatives(*self*)

property n_positives(*self*)

property n_samples(*self*)

property class_ids(*self*)

property image_ids(*self*)

property n_annots(*self*)

property n_images(*self*)

property n_categories(*self*)

lookup_class_name(*self*, *class_id*)

lookup_class_id(*self*, *class_name*)

__nice__(*self*)

classmethod demo(*CocoRegions*)

_make_hashid(*self*)

property isect_index(*self*)
Lazy access to a disk-cached intersection index for this dataset

_lazy_isect_index(*self*, *verbose=None*)

property targets(*self*)
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(*self*)

overlapping_aids(*self*, *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(*self*, *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(*self*, *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(*self*, *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(*self*, *index*, *rng=None*)

Loads from positives and then negatives.

_random_negatives(*self*, *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(*self*, *task*, *window_dims*, *window_overlap=0*)

New experimental method to replace preselect positives / negatives

Parameters **task** (*str*) – can be video_detection # image_detection # video_classification # image_classification

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

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

abstract overlapping_aids(*self*, *gid*, *box*)

preselect(*self*, *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

```
>>> 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**(*cls*, *key='shapes'*, *workdir=None*, *backend=None*, ***kw*)
Create a toy coco sampler for testing and demo puposes

SeeAlso:

- kwcoco.CocoDataset.demo

_init(*self*)

property classes(*self*)

property catgraph(*self*)
DEPRICATED, use self.classes instead

_depends(*self*)

lookup_class_name(*self*, *class_id*)

lookup_class_id(*self*, *class_name*)

property n_positives(*self*)

property n_annots(*self*)

property n_samples(*self*)

__len__(*self*)

property n_images(*self*)

property n_categories(*self*)

property class_ids(*self*)

property image_ids(*self*)

preselect(*self*, ***kwargs*)
Setup a pool of training examples before the epoch begins

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

load_image_with_annots(*self*, *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

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> rng = None
>>> img, anns = self.load_image_with_annots(1)
>>> dets = kwimage.Detections.from_coco_annots(anns, dset=self.dset)
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(img['imdata'][:])
>>> dets.draw()
>>> kwplot.show_if_requested()
```

load_annotations(*self*, *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(*self*, *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(*self*, *index*, *pad=None*, *window_dims=None*, *with_annots=True*)
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
- **pad** (*tuple*) – (height, width) extra context to add to each size. This helps prevent augmentation from producing boundary effects
- **window_dims** (*tuple*) – (height, width) area around the center of the target region to sample.
- **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.

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

load_positive(*self*, *index=None*, *with_annots=True*, *tr=None*, *pad=None*, *rng=None*, ***kw*)
Load an item from the the positive pool of regions.

Parameters

- **index** (*int*) – index of positive target
- **pad** (*tuple*) – (height, width) extra context to add to each size. This helps prevent augmentation from producing boundary effects
- **tr** (*Dict*) – Extra target arguments like window_dims.
- **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.

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

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> rng = None
>>> sample = self.load_positive(pad=(10, 10), tr=dict(window_dims=(3, 3)))
>>> assert sample['im'].shape[0] == 23
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> kwplot.imshow(sample['im'])
>>> kwplot.show_if_requested()
```

load_negative(*self*, *index=None*, *with_annots=True*, *tr=None*, *pad=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.
- **tr** (*Dict*) – Extra target arguments like window_dims.
- **pad** (*tuple*) – (height, width) extra context to add to each size. This helps prevent augmentation from producing boundary effects

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

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> rng = None
>>> sample = self.load_negative(rng=rng, pad=(0, 0))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> box = kwimage.Boxes(tr.reindex(['rel_cx', 'rel_cy', 'width', 'height']).
↪values, 'cxywh')
>>> kwplot.imshow(sample)
>>> kwplot.draw_boxes(box)
>>> kwplot.show_if_requested()
```

Example

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> rng = None
>>> sample = self.load_negative(rng=rng, pad=(0, 0), tr=dict(window_dims=(64,␣
↪64)))
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> box = kwimage.Boxes(tr.reindex(['rel_cx', 'rel_cy', 'width', 'height']).
↪values, 'cxywh')
>>> kwplot.imshow(sample, fnum=1, doclf=True)
>>> kwplot.draw_boxes(box)
>>> kwplot.show_if_requested()
```

load_sample(*self*, *tr*, *with_annots=True*, *visible_thresh=0.0*, *pad=None*, *padkw={'mode': 'constant'}*, *dtype=None*, *nodata=None*)

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

Parameters

- **tr** (*dict*) – target dictionary 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, tr must contain or be able to infer the key *gid (int)*, to specify an image id.

  For 3D video source objects, tr 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

- **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.
- **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*
- **dtype** (*type | None*) – Cast the loaded data to this type. If unspecified returns the data as-is.
- **nodata** (*int | 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.

Returns

sample: dict containing keys im (ndarray | DataArray): image / video 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): 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

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> # The target (tr) lets you specify an arbitrary window
>>> tr = {'gid': 1, 'cx': 5, 'cy': 2, 'width': 6, 'height': 6}
>>> sample = self.load_sample(tr)
...
>>> 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
>>> tr = {'gid': 1, 'cx': 5, 'cy': 2, 'width': 600, 'height': 600}
>>> sample = sampler.load_sample(tr)
>>> annotation_ids = sample['annots']['aids']
>>> aid = annotation_ids[0]
>>> # 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

```
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> tr = self.regions.get_positive(0)
>>> pad = (25, 25)
>>> tr['window_dims'] = 'square'
>>> sample = self.load_sample(tr, pad=pad)
>>> print('im.shape = {!r}'.format(sample['im'].shape))
im.shape = (135, 135, 3)
>>> pad = (0, 0)
>>> tr['window_dims'] = None
```

(continues on next page)

(continued from previous page)

```
>>> sample = self.load_sample(tr, pad=pad)
>>> print('im.shape = {!r}'.format(sample['im'].shape))
im.shape = (52, 85, 3)
>>> # xdoc: +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()
>>> tr = self.regions.get_positive(0)
>>> tr['window_dims'] = (364, 364)
>>> sample = self.load_sample(tr)
>>> annots = sample['annots']
>>> assert len(annots['aids']) > 0
>>> #assert len(annots['rel_cxywh']) == len(annots['aids'])
>>> # xdoc: +REQUIRES(--show)
>>> import kwplot
>>> kwplot.autompl()
>>> abs_frame = self.frames.load_image(sample['tr']['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()
>>> 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')
>>> tr = self.regions.get_positive(1)
>>> pad = None
>>> tr['window_dims'] = (300, 150)
>>> sample = self.load_sample(tr, pad)
>>> assert sample['im'].shape[0:2] == tr['window_dims']
```

(continues on next page)

(continued from previous page)

```
>>> # xdoc: +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))
>>> tr = sample_grid['positives'][0]
>>> tr['channels'] = 'B1|B8'
>>> tr['as_xarray'] = False
>>> sample = self.load_sample(tr)
>>> print(ub.repr2(sample['tr'], nl=1))
>>> print(sample['im'].shape)
>>> assert sample['im'].shape == (3, 128, 128, 2)
>>> tr['channels'] = '<all>'
>>> sample = self.load_sample(tr)
>>> assert sample['im'].shape == (3, 128, 128, 5)
```

_infer_target_attributes(*self*, *tr*)

Infer unpopulated target attribues

Example

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

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

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

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

_load_slice(*self*, *tr*, *pad=None*, *padkw={'mode': 'constant'}*, *dtype=None*, *nodata=None*)

Example

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

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

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

CommandLine: xdoctest -m /home/joncrall/code/ndsampler/ndsampler/coco_sampler.py CocoSampler._load_slice –profile

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))
>>> tr = sample_grid['positives'][0]
>>> tr['channels'] = 'B1|B8'
>>> tr['as_xarray'] = False
>>> sample = self.load_sample(tr)
>>> print(ub.repr2(sample['tr'], nl=1))
>>> print(sample['im'].shape)
>>> assert sample['im'].shape == (3, 128, 128, 2)
>>> tr['channels'] = '<all>'
>>> sample = self.load_sample(tr)
>>> assert sample['im'].shape == (3, 128, 128, 5)
```

_populate_overlap(*self*, *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
>>> from ndsampler.coco_sampler import *
>>> self = CocoSampler.demo()
>>> tr = self.regions.get_item(0)
>>> sample = self._load_slice(tr)
>>> 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__(*self*)

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

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

Returns FrameIntersectionIndex

classmethod demo(*cls*, **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(*self*, *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(*self*, *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(*self*, *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(*self*, *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
```

(continues on next page)

(continued from previous page)

```
>>> 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(*self*)

_support(*self*, *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(*self*, *index*, *pad=None*, *window_dims=None*)

Loads from positives and then negatives.

__len__(*self*)

_depends(*self*)

property class_ids(*self*)

property n_positives(*self*)

property n_annots(*self*)

property n_images(*self*)

image_ids(*self*)

lookup_class_name(*self*, *class_id*)

lookup_class_id(*self*, *class_name*)

_lookup_kpnames(*self*, *class_id*)

property n_categories(*self*)

preselect(*self*, *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(*self*, *image_id=None*, *rng=None*)

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

Returns a random image and its annotations

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

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

load_positive(*self*, *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(*self*, *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

K

L

M

N

O

P

R

S

T

U

V