Activeloop

Release 0.9

Activeloop

Dec 02, 2020

OVERVIEW

1 Problems with Current Workflows **3**

1.1 Installing . . . 3
1.1.1 Install . . . 3
1.2 Why Hub? . . . 4
1.3 Benchmarking . . . 4
1.3.1 Download Parallelism . . . 4
1.3.2 Training Deep Learning Model . . . 5
1.4 API Reference . . . 5
1.4.1 Datasets . . . 5
1.4.2 Pipelines . . . 10
1.4.3 Schema . . . 13
1.5 Getting Started with Hub . . . 21
1.5.1 Intro . . . 21
1.5.2 Getting Started . . . 22
1.5.3 Notes . . . 23
1.5.4 Why commit? . . . 23
1.6 Schema . . . 23
1.6.1 Overview . . . 23
1.6.2 Available Schemas . . . 23
1.6.3 Arguments . . . 25
1.6.4 API . . . 25
1.7 Dataset . . . 33
1.7.1 Create . . . 33
1.7.2 Upload the Data . . . 34
1.7.3 Load the data . . . 34
1.7.4 Convert to Pytorch . . . 34
1.7.5 Convert to Tensorflow . . . 34
1.7.6 Visualize . . . 34
1.7.7 Issues . . . 35
1.7.8 API . . . 35
1.8 Data Pipelines . . . 37
1.8.1 Transform . . . 37
1.8.2 Examples . . . 38
1.8.3 Ray Transform . . . 39
1.8.4 API . . . 40
1.9 PyTorch . . . 42
1.9.1 Dataset to PyTorch Dataset . . . 42
1.9.2 PyTorch Dataset to Dataset . . . 43
1.10 Tensorflow . . . 43
1.10.1 Dataset to Tensorflow Dataset . . . 43

1.10.2 Tensorflow Dataset to Dataset 44
1.10.3 TFDS Dataset to Dataset 44
1.11 Discussions 44

Python Module Index **45**

Index **47**

The fastest way to access and manage datasets for PyTorch and TensorFlow

Hub provides fast access to the state-of-the-art datasets for Deep Learning, enabling data scientists to manage them, build scalable data pipelines and connect to Pytorch and Tensorflow

CHAPTER

ONE

PROBLEMS WITH CURRENT WORKFLOWS

We realized that there are a few problems related with current workflow in deep learning data management through our experience of working with deep learning companies and researchers. Most of the time Data Scientists/ML researchers work on data management and preprocessing instead of doing modeling. Deep Learning often requires to work with large datasets. Those datasets can grow up to terabyte or even petabyte size.

1. It is hard to manage data, version control and track.
2. It is time-consuming to download the data and link with the training or inference code.
3. There is no easy way to access a chunk of it and possibly visualize.

Wouldn't it be more convenient to have large datasets stored & version-controlled as single numpy-like array on the cloud and have access to it from any machine at scale?

1.1 Installing

Hub is available as a simple Python package.

1.1.1 Install

Feel free to run the follow script

```
pip3 install -U hub==1.0.0
```

Upgrade

or in case you would like to upgrade it

```
pip3 install --upgrade hub==1.0.0
```

1.2 Why Hub?

Most of the time Data Scientists/ML researchers work on data management and preprocessing instead of doing modeling. Deep Learning often requires to work with large datasets. Those datasets can grow up to terabyte or even petabyte size. It is hard to manage data, version control and track. It is time consuming to download the data and link with the training or inference code. There is no easy way to access a chunk of it and possibly visualize. **Wouldn't it be more convenient to have large datasets stored & version-controlled as single numpy-like array on the cloud and have access to it from any machine at scale?**

We realized that there are a few problems related with current workflow in deep learning data management through our experience of working with deep learning companies and researchers.

1. **Data locality**. When you have local GPU servers but store the data in a secure remote data center or on the cloud, you need to plan ahead to download specific datasets to your GPU box because it takes time. Sharing preprocessed dataset from one GPU box across your team is also slow and error-prone if there're multiple preprocessing pipelines.
2. **Code dependency on local folder structure**. People use a folder structure to store images or videos. As a result, the data input pipeline has to take into consideration the raw folder structure which creates unnecessary & error-prone code dependency of the dataset folder structure.
3. **Managing preprocessing pipelines**. If you want to run some preprocessing, it would be ideal to save the preprocessed images as a local cache for training.But it's usually hard to manage & version control the preprocessed images locally when there are multiple preprocessing pipelines and the dataset is very big.
4. **Visualization**. It's difficult to visualize the raw data or preprocessed dataset on servers.
5. **Reading a small slice of data**. Another popular way is to store in HDF5/TFRecords format and upload to a cloud bucket, but still you have to manage many chunks of HDF5/TFRecords files. If you want to read a small slice of data, it's not clear which TFRecord/HDF5 chunk you need to load. It's also inefficient to load the whole file for a small slice of data.
6. **Synchronization across team**. If multiple users modify the data, there needs to be a data versioning and synchronization protocol implemented.
7. **RAM management**. Whenever you want to create a numpy array you are worried if the numpy array is going to fit in the local RAM/disk limit.

1.3 Benchmarking

For full reproducibility please refer to the code

1.3.1 Download Parallelism

The following chart shows that hub on a single machine (aws p3.2xlarge) can achieve up to 875 MB/s download speed with multithreading and multiprocessing enabled. Choosing the chunk size plays a role in reaching maximum speed up. The bellow chart shows the tradeoff using different number of threads and processes.

1.3.2 Training Deep Learning Model

The following benchmark shows that streaming data through Hub package while training deep learning model is equivalent to reading data from local file system. The benchmarks have been produced on AWS using p3.2xlarge machine with V100 GPU. The data is stored on S3 within the same region. In the asynchronous data loading figure, first three models (VGG, Resnet101 and DenseNet) have no data bottleneck. Basically the processing time is greater than loading the data in the background. However for more lightweight models such as Resnet18 or SqueezeNet, training is bottlenecked on reading speed. Number of parallel workers for reading the data has been chosen to be the same. The batch size was chosen smaller for large models to fit in the GPU RAM.

Training Deep Learning

Data Streaming

1.4 API Reference

1.4.1 Datasets

Dataset

class hub.**Dataset**(*url: str*, *mode: str = 'a'*, *safe_mode: bool = False*, *shape=None*, *schema=None*, *token=None*, *fs=None*, *fs_map=None*, *cache: int = 67108864*, *storage_cache: int = 268435456*, *lock_cache=True*, *tokenizer=None*)

__getitem__(*slice_*)

Gets a slice or slices from dataset
Usage:

```
>>> return ds["image", 5, 0:1920, 0:1080, 0:3].numpy() # returns numpy array
>>> images = ds["image"]
>>> return images[5].numpy() # returns numpy array
>>> images = ds["image"]
>>> image = images[5]
>>> return image[0:1920, 0:1080, 0:3].numpy()
```

__init__(*url: str*, *mode: str = 'a'*, *safe_mode: bool = False*, *shape=None*, *schema=None*, *token=None*, *fs=None*, *fs_map=None*, *cache: int = 67108864*, *storage_cache: int = 268435456*, *lock_cache=True*, *tokenizer=None*)

Open a new or existing dataset for read/write :param url: The url where dataset is located/should be created :type url: str :param mode: Python way to tell whether dataset is for read or write (ex. “r”, “w”, “a”) :type mode: str, optional (default to “w”) :param safe_mode: if dataset exists it cannot be rewritten in safe mode, otherwise it lets to write the first time :type safe_mode: bool, optional :param shape: Tuple with (num_samples,) format, where num_samples is number of samples :type shape: tuple, optional :param schema: Describes the data of a single sample. Hub schemas are used for that

Required for ‘a’ and ‘w’ modes

Parameters

- **token** (*str or dict, optional*) – If url is refering to a place where authorization is required, token is the parameter to pass the credentials, it can be filepath or dict
- **fs** (*optional*) –

- **fs_map** (*optional*) –
- **cache** (*int, optional*) – Size of the memory cache. Default is 64MB (2**26) if 0, False or None, then cache is not used
- **storage_cache** (*int, optional*) – Size of the storage cache. Default is 256MB (2**28) if 0, False or None, then storage cache is not used
- **lock_cache** (*bool, optional*) – Lock the cache for avoiding multiprocessing errors

__iter__()
Returns Iterable over samples

__len__()
Number of samples in the dataset

__repr__()
Return repr(self).

__setitem__(*slice_*, *value*)

Sets a slice or slices with a value
Usage >>> ds["image", 5, 0:1920, 0:1080, 0:3] = np.zeros((1920, 1080, 3), "uint8")

```
>>> images = ds["image"]
>>> image = images[5]
>>> image[0:1920, 0:1080, 0:3] = np.zeros((1920, 1080, 3), "uint8")
```

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

_check_and_prepare_dir()
Checks if input data is ok. Creates or overwrites dataset folder. Returns True dataset needs to be created opposed to read.

_get_dictionary(*subpath*, *slice_=None*)
"Gets dictionary from dataset given incomplete subpath

append_shape(*size: int*)
Append the shape: Heavy Operation

close()
Save changes from cache to dataset final storage This invalidates this object

commit()
Deprecated alias to flush()

flush()
Save changes from cache to dataset final storage Does not invalidate this object

static from_pytorch(*dataset*)
Converts a pytorch dataset object into hub format :param dataset: The pytorch dataset object that needs to be converted into hub format

static from_tensorflow(*ds*)
Converts a tensorflow dataset into hub format :param dataset: The tensorflow dataset object that needs to be converted into hub format

Examples

ds = tf.data.Dataset.from_tensor_slices(tf.range(10)) out_ds = hub.Dataset.from_tensorflow(ds) res_ds = out_ds.store("username/new_dataset") # res_ds is now a usable hub dataset

ds = tf.data.Dataset.from_tensor_slices({'a': [1, 2], 'b': [5, 6]}) out_ds = hub.Dataset.from_tensorflow(ds) res_ds = out_ds.store("username/new_dataset") # res_ds is now a usable hub dataset

ds = hub.Dataset(schema=my_schema, shape=(1000,), url="username/dataset_name", mode="w") ds = ds.to_tensorflow() out_ds = hub.Dataset.from_tensorflow(ds) res_ds = out_ds.store("username/new_dataset") # res_ds is now a usable hub dataset

static from_tfds(*dataset*, *split=None*, *num=- 1*, *sampling_amount=1*)

Converts a TFDS Dataset into hub format :param dataset: The name of the tfds dataset that needs to be converted into hub format :type dataset: str :param split: A string representing the splits of the dataset that are required such as "train" or "test+train"

If not present, all the splits of the dataset are used.

Parameters

- **num** (*int, optional*) – The number of samples required. If not present, all the samples are taken. If count is -1, or if count is greater than the size of this dataset, the new dataset will contain all elements of this dataset.
- **sampling_amount** (*float, optional*) – a value from 0 to 1, that specifies how much of the dataset would be sampled to determinte feature shapes value of 0 would mean no sampling and 1 would imply that entire dataset would be sampled

Examples

out_ds = hub.Dataset.from_tfds('mnist', split='test+train', num=1000) res_ds = out_ds.store("username/mnist") # res_ds is now a usable hub dataset

property keys

Get Keys of the dataset

resize_shape(*size: int*) → None

Resize the shape of the dataset by resizing each tensor first dimension

to_pytorch(*Transform=None*, *offset=None*, *num_samples=None*)

Converts the dataset into a pytorch compatible format :param offset: The offset from which dataset needs to be converted :type offset: int, optional :param num_samples: The number of samples required of the dataset that needs to be converted :type num_samples: int, optional

to_tensorflow(*offset=None*, *num_samples=None*)

Converts the dataset into a tensorflow compatible format :param offset: The offset from which dataset needs to be converted :type offset: int, optional :param num_samples: The number of samples required of the dataset that needs to be converted :type num_samples: int, optional

DatasetView

class hub.api.datasetview.**DatasetView**(*dataset=None*, *num_samples=None*, *offset=None*, *squeeze_dim=False*)

__getitem__(*slice_*)

Gets a slice or slices from DatasetView
Usage:

```
>>> ds_view = ds[5:15]
>>> return ds_view["image", 7, 0:1920, 0:1080, 0:3].compute() # returns numpy␣
↪array of 12th image
```

__init__(*dataset=None*, *num_samples=None*, *offset=None*, *squeeze_dim=False*)
Creates a DatasetView object for a subset of the Dataset

Parameters

- **dataset** (*hub.api.dataset.Dataset object*) – The dataset whose DatasetView is being created
- **num_samples** (*int*) – The number of samples in this DatasetView
- **offset** (*int*) – The offset from which the DatasetView starts
- **squuze_dim** (*bool*) – For slicing with integers we would love to remove the first dimension to make it nicer

__iter__()
Returns Iterable over samples

__repr__()
Return repr(self).

__setitem__(*slice_*, *value*)

Sets a slice or slices with a value
Usage:

```
>>> ds_view = ds[5:15]
>>> ds_view["image", 3, 0:1920, 0:1080, 0:3] = np.zeros((1920, 1080, 3),
↪"uint8") # sets the 8th image
```

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

_get_dictionary(*subpath*, *slice_=None*)
“Gets dictionary from dataset given incomplete subpath

commit() → None
Commit dataset

property keys
Get Keys of the dataset

resize_shape(*size: int*) → None
Resize dataset shape, not DatasetView

to_pytorch(*Transform=None*)
Converts the dataset into a pytorch compatible format

to_tensorflow()
Converts the dataset into a tensorflow compatible format

TensorView

class hub.api.tensorview.**TensorView**(*dataset=None*, *subpath=None*, *slice_=None*, *squeeze_dims=[]*)

__getitem__(*slice_*)

Gets a slice or slices from tensorview
Usage:

```
>>> images_tensorview = ds["image"]
>>> return images_tensorview[7, 0:1920, 0:1080, 0:3].compute() # returns␣
↪numpy array of 7th image
```

__init__(*dataset=None*, *subpath=None*, *slice_=None*, *squeeze_dims=[]*)
Creates a TensorView object for a particular tensor in the dataset

Parameters

- **dataset** (*hub.api.dataset.Dataset object*) – The dataset whose TensorView is being created
- **subpath** (*str*) – The full path to the particular Tensor in the Dataset
- **slice** (*optional*) – The *slice_* of this Tensor that needs to be accessed

__repr__()
Return repr(self).

__setitem__(*slice_*, *value*)

Sets a slice or slices with a value
Usage:

```
>>> images_tensorview = ds["image"]
>>> images_tensorview[7, 0:1920, 0:1080, 0:3] = np.zeros((1920, 1080, 3),
↪"uint8") # sets 7th image
```

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

_combine(*slice_*, *num=None*, *ofs=0*)
Combines a *slice_* with the current num and offset present in tensorview

check_slice_bounds(*num=None*, *start=None*, *stop=None*, *step=None*)
Checks whether the bounds of slice are in limits

compute()
Gets the value from tensorview

dtype_from_path(*path*)
Gets the dtype of the Tensorview by traversing the schema

numpy()
Gets the value from tensorview

slice_fill(*slice_*)
Fills the slice with zeroes for the dimensions that have single elements and squeeze_dims true

Sharded Dataset

class hub.api.sharded_datasetview.**ShardedDatasetView**(*datasets: list*)

__init__(*datasets: list*) → None

Creates a sharded simple dataset.
Datasets should have the schema.

Parameters **datasets** (*list of Datasets*) –

__iter__()
Returns Iterable over samples

__repr__()
Return repr(self).

__weakref__
list of weak references to the object (if defined)

identify_shard(*index*) → tuple
Computes shard id and returns the shard index and offset

slicing(*slice_*)
Identifies the dataset shard that should be used .. rubric:: Notes

Features of advanced slicing are missing as one would expect from a DatasetView E.g. cross sharded dataset access is missing

1.4.2 Pipelines

Transform

hub.compute.**transform**(*schema*, *scheduler='single'*, *workers=1*)
Transform is a decorator of a function. The function should output a dictionary per sample

Parameters

schema: Schema The output format of the transformed dataset

scheduler: str "single" - for single threaded, "threaded" using multiple threads, "processed", "ray" scheduler, "dask" scheduler

workers: int how many workers will be started for the process

class hub.compute.transform.**Transform**(*func*, *schema*, *ds*, *scheduler: str = 'single'*, *workers: int = 1*, ***kwargs*)

__getitem__(*slice_*)
Get an item to be computed without iterating on the whole dataset Creates a dataset view, then a temporary dataset to apply the transform

slice_: slice Gets a slice or slices from dataset

__init__(*func*, *schema*, *ds*, *scheduler: str = 'single'*, *workers: int = 1*, ***kwargs*)
Transform applies a user defined function to each sample in single threaded manner

Parameters

- **func** (*function*) – user defined function func(x, **kwargs)
- **schema** (*dict of dtypes*) – the structure of the final dataset that will be created
- **ds** (*Iterative*) – input dataset or a list that can be iterated
- **scheduler** (*str*) – choice between "single", "threaded", "processed"
- **workers** (*int*) – how many threads or processes to use
- ****kwargs** – additional arguments that will be passed to func as static argument for all samples

__weakref__
list of weak references to the object (if defined)

classmethod _flatten(*items*, *schema*)
Takes a dictionary or list of dictionary Returns a dictionary of concatenated values Dictionary follows schema

classmethod _flatten_dict(*d: Dict*, *parent_key=''*, *schema=None*)
Helper function to flatten dictionary of a recursive tensor

Parameters **d** (*dict*) –

_pbar(*show: bool = True*)
Returns a progress bar, if empty then it function does nothing

_split_list_to_dicts(*xs*)
Helper function that transform list of dicts into dicts of lists

Parameters **xs** (*list of dicts*) –

Returns xs_new

Return type dicts of lists

classmethod _unwrap(*results*)
If there is any list then unwrap it into its elements

create_dataset(*url*, *length=None*, *token=None*)
Helper function to creat a dataset

classmethod dtype_from_path(*path*, *schema*)
Helper function to get the dtype from the path

store(*url: str*, *token: dict = None*, *length: int = None*, *ds: Iterable = None*, *progressbar: bool = True*, *sample_per_shard=None*)
The function to apply the transformation for each element in batchified manner

Parameters

- **url** (*str*) – path where the data is going to be stored
- **token** (*str or dict, optional*) – If url is refering to a place where authorization is required, token is the parameter to pass the credentials, it can be filepath or dict
- **length** (*int*) – in case shape is None, user can provide length
- **ds** (*Iterable*) –
- **progressbar** (*bool*) – Show progress bar
- **sample_per_shard** (*int*) – How to split the iterator not to overfill RAM

Returns **ds** – uploaded dataset

Return type *hub.Dataset*

store_shard(*ds_in: Iterable*, *ds_out: hub.api.dataset.Dataset*, *offset: int*, *token=None*)
Takes a shard of iteratable ds_in, compute and stores in DatasetView

upload(*results*, *ds: hub.api.dataset.Dataset*, *token: dict*, *progressbar: bool = True*)
Batchified upload of results For each tensor batchify based on its chunk and upload If tensor is dynamic then still upload element by element For dynamic tensors, it disable dynamicness and then enables it back

Parameters

- **dataset** (hub.Dataset) – Dataset object that should be written to
- **results** – Output of transform function
- **progressbar** (*bool*) –

Returns **ds** – Uploaded dataset

Return type *hub.Dataset*

RayTransform

class hub.compute.ray.**RayTransform**(*func*, *schema*, *ds*, *scheduler='ray'*, *workers=1*, ***kwargs*)

__init__(*func*, *schema*, *ds*, *scheduler='ray'*, *workers=1*, ***kwargs*)
Transform applies a user defined function to each sample in single threaded manner

Parameters

- **func** (*function*) – user defined function func(x, **kwargs)
- **schema** (*dict of dtypes*) – the structure of the final dataset that will be created
- **ds** (*Iterative*) – input dataset or a list that can be iterated
- **scheduler** (*str*) – choice between “single”, “threaded”, “processed”
- **workers** (*int*) – how many threads or processes to use
- ****kwargs** – additional arguments that will be passed to func as static argument for all samples

store(*url: str*, *token: dict = None*, *length: int = None*, *ds: Iterable = None*, *progressbar: bool = True*)
The function to apply the transformation for each element in batchified manner

Parameters

- **url** (*str*) – path where the data is going to be stored

- **token** (*str or dict, optional*) – If url is refering to a place where authorization is required, token is the parameter to pass the credentials, it can be filepath or dict
- **length** (*int*) – in case shape is None, user can provide length
- **ds** (*Iterable*) –
- **progressbar** (*bool*) – Show progress bar

Returns **ds** – uploaded dataset

Return type *hub.Dataset*

upload(*results*, *url: str*, *token: dict*, *progressbar: bool = True*)
Batchified upload of results For each tensor batchify based on its chunk and upload If tensor is dynamic then still upload element by element

Parameters

- **dataset** (hub.Dataset) – Dataset object that should be written to
- **results** – Output of transform function
- **progressbar** (*bool*) –

Returns **ds** – Uploaded dataset

Return type *hub.Dataset*

1.4.3 Schema

Serialization

hub.schema.serialize.**serialize**(*input*)
Converts the input into a serializable format

hub.schema.serialize.**serialize_SchemaDict**(*fdict*)
Converts SchemaDict into a serializable format

hub.schema.serialize.**serialize_primitive**(*primitive*)
Converts Primitive into a serializable format

hub.schema.serialize.**serialize_tensor**(*tensor*)
Converts Tensor and its derivatives into a serializable format

Schema

class hub.schema.audio.**Audio**(*shape: Tuple[int, ...] = None*, *dtype='int64'*, *file_format=None*, *sample_rate: int = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)

__init__(*shape: Tuple[int, ...] = None*, *dtype='int64'*, *file_format=None*, *sample_rate: int = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
Constructs the connector.

Parameters

- **file_format** (*str*) – the audio file format. Can be any format ffmpeg understands. If *None*, will attempt to infer from the file extension.
- **shape** (*tuple*) – shape of the data.

- **dtype** (*str*) – The dtype of the data.
- **sample_rate** (*int*) – additional metadata exposed to the user through *info.schema['audio'].sample_rate*. This value isn't used neither in encoding nor decoding.

Raises **ValueError** – If the shape is invalid:

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

class hub.schema.bbox.**BBox**(*dtype='float64'*, *chunks=None*, *compressor='lz4'*)
HubSchema for a normalized bounding box. Output: bbox: Tensor of type *float32* and shape *[4,]* which contains the

normalized coordinates of the bounding box *[ymin, xmin, ymax, xmax]*

__init__(*dtype='float64'*, *chunks=None*, *compressor='lz4'*)
Construct the connector.

Parameters

- **dtype** (*str*) – dtype of bbox coordinates. Default: 'float32'
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

class hub.schema.class_label.**ClassLabel**(*num_classes: int = None*, *names: List[str] = None*, *names_file: str = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for integer class labels.

__init__(*num_classes: int = None*, *names: List[str] = None*, *names_file: str = None*, *chunks=None*, *compressor='lz4'*)

Constructs a ClassLabel HubSchema.

There are 3 ways to define a ClassLabel, which correspond to the 3 arguments:

* *num_classes*: create 0 to (num_classes-1) labels

* *names*: a list of label strings

* *names_file*: a file containing the list of labels.

Note: In python2, the strings are encoded as utf-8.

Usage:

```
>>> class_label_tensor = ClassLabel(num_classes=10)
>>> class_label_tensor = ClassLabel(names=['class1', 'class2', 'class3', ...])
>>> class_label_tensor = ClassLabel(names_file='/path/to/file/with/names')
```

Parameters

- **num_classes** (*int*) – number of classes. All labels must be < num_classes.
- **names** (*list<str>*) – string names for the integer classes. The order in which the names are provided is kept.
- **names_file** (*str*) – path to a file with names for the integer classes, one per line.
- **max_shape** (*Tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks
- **Note** (|) – names or names file

Raises **ValueError** – If more than one argument is provided:

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

int2str(*int_value: int*)
Conversion integer => class name string.

str2int(*str_value: str*)
Conversion class name string => integer.

class hub.schema.image.**Image**(*shape: Tuple[int, ...] = None, None, 3, dtype='uint8', max_shape: Tuple[int, ...] = None, chunks=None, compressor='lz4'*)
HubSchema for images Output: *tf.Tensor* of type *tf.uint8* and shape *[height, width, num_channels]* for BMP, JPEG, and PNG images

Example: ```python image_tensor = Image(shape=(None, None, 1),

encoding_format='png')

```

**__init__**(*shape: Tuple[int, ... ] = None, None, 3, dtype='uint8', max_shape: Tuple[int, ... ] = None, chunks=None, compressor='lz4'*)

Construct the connector.

**Parameters**

- **shape** (*tuple of ints or None*) – The shape of decoded image: (height, width, channels) where height and width can be None. Defaults to (None, None, 3).
- **dtype** (*uint16* or *uint8* (default)) – *uint16* can be used only with png encoding_format
```

- **encoding_format** (*'jpeg' or 'png' (default)*) – Format to serialize np.ndarray images on disk.
- **max_shape** (*Tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

Returns

- *tf.Tensor* of type *tf.uint8* and shape *[height, width, num_channels]*
- *for BMP, JPEG, and PNG images*

Raises **ValueError** – If the shape, dtype or encoding formats are invalid:

__repr__()
Return repr(self).

__str__()
Return str(self).

_set_dtype(*dtype*)
Set the dtype.

_set_encoding_format(*encoding_format*)
Set the encoding format.

get_attr_dict()
Return class attributes.

class hub.schema.features.**FlatTensor**(*path: str*, *shape: Tuple[int, ...]*, *dtype*, *max_shape: Tuple[int, ...]*, *chunks: Tuple[int, ...]*)
Tensor metadata after applying flatten function

__init__(*path: str*, *shape: Tuple[int, ...]*, *dtype*, *max_shape: Tuple[int, ...]*, *chunks: Tuple[int, ...]*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

class hub.schema.features.**HubSchema**
Base class for all datatypes

__weakref__
list of weak references to the object (if defined)

_flatten() → Iterable[*hub.schema.features.FlatTensor*]
Flattens dtype into list of tensors that will need to be stored seperately

class hub.schema.features.**Primitive**(*dtype*, *chunks=None*, *compressor='lz4'*)
Class for handling primitive datatypes All numpy primitive data types like int32, float64, etc... should be wrapped around this class

__init__(*dtype*, *chunks=None*, *compressor='lz4'*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__()
Return str(self).

_flatten()
Flattens dtype into list of tensors that will need to be stored seperately

class hub.schema.features.**SchemaDict**(*dict_*)
Class for dict branching of a datatype SchemaDict dtype contains str -> dtype associations This way you can describe complex datatypes

__init__(*dict_*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__()
Return str(self).

_flatten()
Flattens dtype into list of tensors that will need to be stored seperately

class hub.schema.features.**Tensor**(*shape: Tuple[int, ...] = None*, *dtype='float64'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
Tensor type in schema Has np-array like structure contains any type of elements (Primitive and non-Primitive)

__init__(*shape: Tuple[int, ...] = None*, *dtype='float64'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)

Parameters

- **shape** (*Tuple[int]*) – Shape of tensor, can contains None(s) meaning the shape can be dynamic Dynamic shape means it can change during editing the dataset
- **dtype** (*SchemaConnector or str*) – dtype of each element in Tensor. Can be Primitive and non-Primitive type
- **max_shape** (*Tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

__repr__()
Return repr(self).

__str__()
Return str(self).

_flatten()
Flattens dtype into list of tensors that will need to be stored seperately

hub.schema.features.**featurify**(*schema*) → *hub.schema.features.HubSchema*
This functions converts naked primitive datatypes and ditcs into Primitives and SchemaDicts That way every node in dtype tree is a SchemaConnector type object

hub.schema.features.**flatten**(*dtype*, *root=''*)
Flattens nested dictionary and returns tuple (dtype, path)

class hub.schema.mask.**Mask**(*shape: Tuple[int, ...] = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for mask

Usage:

```
>>> mask_tensor = Mask(shape=(300, 300, 1))
```

__init__(*shape: Tuple[int, ...] = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
Constructs a Mask HubSchema.

Parameters

- **shape** (*tuple of ints or None*) – Shape in format (height, width, 1)
- **dtype** (*str*) – Dtype of mask array. Default: *uint8*
- **max_shape** (*Tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

class hub.schema.polygon.**Polygon**(*shape: Tuple[int, ...] = None*, *dtype='int32'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for polygon

Usage:

```
>>> polygon_tensor = Polygon(shape=(10, 2))
>>> polygon_tensor = Polygon(shape=(None, 2))
```

__init__(*shape: Tuple[int, ...] = None*, *dtype='int32'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
Constructs a Polygon HubSchema. Args: shape: tuple of ints or None, i.e (None, 2)

Parameters

- **shape** (*tuple of ints or None*) – Shape in format (None, 2)
- **max_shape** (*Tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

Raises **ValueError** – If the shape is invalid:

__repr__()
Return repr(self).

__str__()
Return str(self).

_check_shape(*shape*)
Check if provided shape maches polygon characteristics.

get_attr_dict()
Return class attributes.

class hub.schema.segmentation.**Segmentation**(*shape: Tuple[int, ...] = None*, *dtype: str = None*, *num_classes: int = None*, *names: Tuple[str] = None*, *names_file: str = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for segmentation

__init__(*shape: Tuple[int, ...] = None*, *dtype: str = None*, *num_classes: int = None*, *names: Tuple[str] = None*, *names_file: str = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
Constructs a Segmentation HubSchema. Also constructs ClassLabel HubSchema for Segmentation classes.

Parameters

- **shape** (*tuple of ints or None*) – Shape in format (height, width, 1)
- **dtype** (*str*) – dtype of segmentation array: *uint16* or *uint8*
- **num_classes** (*int*) – Number of classes. All labels must be < num_classes.
- **names** (*list<str>*) – string names for the integer classes. The order in which the names are provided is kept.
- **names_file** (*str*) – Path to a file with names for the integer classes, one per line.
- **max_shape** (*tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

get_segmentation_classes()
Get classes of the segmentation mask

class hub.schema.sequence.**Sequence**(*shape=()*, *max_shape=()*, *dtype=None*, *chunks=None*, *compressor='lz4'*)
Sequence correspond to sequence of *features.HubSchema*. At generation time, a list for each of the sequence element is given. The output of *Dataset* will batch all the elements of the sequence together. If the length of the sequence is static and known in advance, it should be specified in the constructor using the *length* param.

Usage:

```
>>> sequence = Sequence(Image(), length=NB_FRAME)
```

__init__(*shape=()*, *max_shape=()*, *dtype=None*, *chunks=None*, *compressor='lz4'*)
Construct a sequence of Tensors. :param shape: Single integer element tuple representing length of sequence

If None then dynamic

Parameters

- **dtype** (*str | HubSchema*) – Datatype of each element in sequence
- **chunks** (*Tuple[int] | int*) – Number of elements in chunk Works only for top level sequence You can also include number of samples in a single chunk

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes

class hub.schema.text.**Text**(*shape: Tuple[int, ...] = None*, *dtype='int64'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for text

__init__(*shape: Tuple[int, ...] = None*, *dtype='int64'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)

Construct the connector.

Parameters

- **shape** (*tuple of ints or None*) – The shape of the text
- **dtype** (*str*) – the dtype for storage.
- **max_shape** (*Tuple[int]*) – Maximum number of words in the text
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

__repr__()
Return repr(self).

__str__()
Return str(self).

_set_dtype(*dtype*)
Set the dtype.

get_attr_dict()
Return class attributes.

class hub.schema.video.**Video**(*shape: Tuple[int, ...] = None*, *dtype: str = 'uint8'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for videos, encoding frames individually on disk.

The connector accepts as input a 4 dimensional *uint8* array representing a video.

Returns **Tensor** – where channels must be 1 or 3

Return type *uint8* and shape [num_frames, height, width, channels],

__init__(*shape: Tuple[int, ...] = None*, *dtype: str = 'uint8'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
Initializes the connector.

Parameters

- **shape** (*tuple of ints*) – The shape of the video (num_frames, height, width, channels), where channels is 1 or 3.
- **encoding_format** (*str*) – The video is stored as a sequence of encoded images. You can use any encoding format supported by Image.
- **dtype** (*uint16* or *uint8* (default)) –

Raises **ValueError** – If the shape, dtype or encoding formats are invalid:

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

1.5 Getting Started with Hub

1.5.1 Intro

Today we introduce our new API/format for hub package.

Here is some features of new hub:

1. Ability to modify datasets on fly. Datasets are no longer immutable and can be modified over time.
2. Larger datasets can now be uploaded as we removed some RAM limiting components from the hub.
3. Caching is introduced to improve IO performance.
4. Dynamic shaping enables very large images/data support. You can have large images/data stored in hub.
5. Dynamically sized datasets. You will be able to increase number of samples dynamically.
6. Tensors can be added to dataset on the fly.

1.5.2 Getting Started

1. Install beta version

```
pip3 install hub==1.0.0
```

2. Register and authenticate to uploade datasests

```
hub register
hub login
```

3. Lets start by creating dataset

```
import numpy as np

import hub
from hub.schema import ClassLabel, Image

my_schema = {
    "image": Image((28, 28)),
    "label": ClassLabel(num_classes=10),
}

url = "./data/examples/new_api_intro" #instead write your {username}/{dataset} to␣
↪make it public

ds = hub.Dataset(url, shape=(1000,), schema=my_schema)
for i in range(len(ds)):
    ds["image", i] = np.ones((28, 28), dtype="uint8")
    ds["label", i] = 3

print(ds["image", 5].compute())
print(ds["label", 100:110].compute())
ds.close()
```

You can also create a dataset in *s3*, *Google CLoud Storage* or *Azure*. :

```
url = 's3://new_dataset'  # s3
url = 'gcs://new_dataset' # gcloud
url = 'https://activeloop.blob.core.windows.net/activeloop-hub/new_dataset' # Azure
```

1. Transferring from TFDS In `hub==1.0.0` we would also have

```
import hub
import tensorflow as tf

out_ds = hub.Dataset.from_tfds('mnist', split='test+train', num=1000)
res_ds = out_ds.store("username/mnist") # res_ds is now a usable hub dataset
```

1.5.3 Notes

New hub mimics TFDS data types. Before creating dataset you have to mention the details of what type of data does it contain. This enables us to compress, process and visualize data more efficiently.

This code creates dataset in *"./data/examples/new_api_intro"* folder with overwrite mode. Dataset has 1000 samples. In each sample there is an *image* and a *label*.

After this we can loop over dataset and read/write from it.

1.5.4 Why commit?

Since caching is in place, you need to tell program to push final changes to permanent storage.

`.close()` saves changes from cache to dataset final storage and does not invalidate dataset object. On the other hand, `.flush()` saves changes to dataset, but invalidates it.

Alternatively you can use the following style.

```
with hub.Dataset(...) as ds:
    pass
```

This works as well.

1.6 Schema

1.6.1 Overview

Hub Schema:

- Define the structure, shapes, dtypes of the final Dataset
- Add additional meta information(image channels, class names, etc.)
- Use special serialization/deserialization methods

1.6.2 Available Schemas

Primitive

Wrapper to the numpy primitive data types like int32, float64, etc…

```
from hub.schema import Primitive

schema = { "scalar": Primitive(dtype="float32") }
```

Tensor

Np-array like structure that contains any type of elements (Primitive and non-Primitive).

```
from hub.schema import Tensor

schema = {"tensor_1": Tensor((None, None), max_shape=(200, 200), "int32"),
          "tensor_2": Tensor((100, 400), "int64", chunks=(6, 50, 200)) }
```

Image

Array representation of image of arbitrary shape and primitive data type.

Default encoding format - `png` (`jpeg` is also supported).

```
from hub.schema import Image

schema = {"image": Image(shape=(None, None),
                         dtype="int32",
                         max_shape=(100, 100)
          ) }
```

ClassLabel

Integer representation of feature labels. Can be constructed from number of labels, label names or a text file with a single label name in each line.

```
from hub.schema import ClassLabel

schema = {"class_label_1": ClassLabel(num_classes=10),
          "class_label_2": ClassLabel(names=['class1', 'class2', 'class3', ...]),
          "class_label_3": ClassLabel(names_file='/path/to/file/with/names')
          ) }
```

Mask

Array representation of binary mask. The shape of mask should have format: (height, width, 1).

```
from hub.schema import Image

schema = {"mask": Mask(shape=(244, 244, 1))}
```

Segmentation

Segmentation array. Also constructs ClassLabel feature connector to support segmentation classes.

The shape of segmentation mask should have format: (height, width, 1).

```
from hub.schema import Segmentation

schema = {"segmentation": Segmentation(shape=(244, 244, 1), dtype='uint8',
                                       names=['label_1', 'label_2', ...])}
```

BBox

Bounding box coordinates with shape (4,).

```
from hub.schema import BBox

schema = {"bbox": BBox()}
```

Audio

Hub schema for audio files. A file can have any format ffmpeg understands. If `file_format` parameter isn't provided will attempt to infer it from the file extension. Also, `sample_rate` parameter can be added as additional metadata. User can access through info.schema['audio'].sample_rate.

```
from hub.schema import Audio

schema = {'audio': Audio(shape=(300,))}
```

Video

Video format support. Accepts as input a 4 dimensional uint8 array representing a video. The video is stored as a sequence of encoded images. `encoding_format` can be any format supported by Image.

```
from hub.schema import Video

schema = {'video': Video(shape=(20, None, None, 3), max_shape=(20, 1200, 1200, 3))}
```

1.6.3 Arguments

If a schema has a dynamic shape, `max_shape` argument should be provided representing the maximum possible number of elements in each axis of the feature.

Argument `chunks` describes how to split tensor dimensions into chunks (files) to store them efficiently. If not chosen, it will be automatically detected how to split the information into chunks.

1.6.4 API

class hub.schema.audio.**Audio**(*shape: Tuple[int, ...] = None*, *dtype='int64'*, *file_format=None*, *sample_rate: int = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)

__init__(*shape: Tuple[int, ...] = None*, *dtype='int64'*, *file_format=None*, *sample_rate: int = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)

Constructs the connector.

Parameters

- **file_format** (*str*) – the audio file format. Can be any format ffmpeg understands. If *None*, will attempt to infer from the file extension.
- **shape** (*tuple*) – shape of the data.
- **dtype** (*str*) – The dtype of the data.

- **sample_rate** (*int*) – additional metadata exposed to the user through *info.schema['audio'].sample_rate*. This value isn't used neither in encoding nor decoding.

Raises **ValueError** – If the shape is invalid:

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

class hub.schema.bbox.**BBox**(*dtype='float64'*, *chunks=None*, *compressor='lz4'*)
HubSchema for a normalized bounding box. Output: bbox: Tensor of type *float32* and shape *[4,]* which contains the

normalized coordinates of the bounding box *[ymin, xmin, ymax, xmax]*

__init__(*dtype='float64'*, *chunks=None*, *compressor='lz4'*)
Construct the connector.

Parameters

- **dtype** (*str*) – dtype of bbox coordinates. Default: 'float32'
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

class hub.schema.class_label.**ClassLabel**(*num_classes: int = None*, *names: List[str] = None*, *names_file: str = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for integer class labels.

__init__(*num_classes: int = None*, *names: List[str] = None*, *names_file: str = None*, *chunks=None*, *compressor='lz4'*)

Constructs a ClassLabel HubSchema.

There are 3 ways to define a ClassLabel, which correspond to the 3 arguments:

* *num_classes*: create 0 to (num_classes-1) labels

* *names*: a list of label strings

* *names_file*: a file containing the list of labels.

Note: In python2, the strings are encoded as utf-8.

Usage:

```
>>> class_label_tensor = ClassLabel(num_classes=10)
>>> class_label_tensor = ClassLabel(names=['class1', 'class2', 'class3', ...])
>>> class_label_tensor = ClassLabel(names_file='/path/to/file/with/names')
```

Parameters

- **num_classes** (*int*) – number of classes. All labels must be < num_classes.
- **names** (*list<str>*) – string names for the integer classes. The order in which the names are provided is kept.
- **names_file** (*str*) – path to a file with names for the integer classes, one per line.
- **max_shape** (*Tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks
- **Note** (*|*) – names or names file

Raises **ValueError** – If more than one argument is provided:

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

int2str(*int_value: int*)
Conversion integer => class name string.

str2int(*str_value: str*)
Conversion class name string => integer.

class hub.schema.image.**Image**(*shape: Tuple[int, ...] = None, None, 3, dtype='uint8', max_shape: Tuple[int, ...] = None, chunks=None, compressor='lz4'*)
HubSchema for images Output: *tf.Tensor* of type *tf.uint8* and shape *[height, width, num_channels]* for BMP, JPEG, and PNG images

Example: ```python image_tensor = Image(shape=(None, None, 1),

encoding_format='png')

```

**__init__**(*shape: Tuple[int, ... ] = None, None, 3, dtype='uint8', max_shape: Tuple[int, ... ] = None, chunks=None, compressor='lz4'*)

Construct the connector.

**Parameters**

- **shape** (*tuple of ints or None*) – The shape of decoded image: (height, width, channels) where height and width can be None. Defaults to (None, None, 3).
- **dtype** (*uint16* or *uint8* (default)) – *uint16* can be used only with png encoding_format
```

- **encoding_format** (*'jpeg' or 'png' (default)*) – Format to serialize np.ndarray images on disk.
- **max_shape** (*Tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

Returns

- *tf.Tensor* of type *tf.uint8* and shape *[height, width, num_channels]*
- *for BMP, JPEG, and PNG images*

Raises **ValueError** – If the shape, dtype or encoding formats are invalid:

__repr__()
Return repr(self).

__str__()
Return str(self).

_set_dtype(*dtype*)
Set the dtype.

_set_encoding_format(*encoding_format*)
Set the encoding format.

get_attr_dict()
Return class attributes.

class hub.schema.features.**FlatTensor**(*path: str*, *shape: Tuple[int, ...]*, *dtype*, *max_shape: Tuple[int, ...]*, *chunks: Tuple[int, ...]*)
Tensor metadata after applying flatten function

__init__(*path: str*, *shape: Tuple[int, ...]*, *dtype*, *max_shape: Tuple[int, ...]*, *chunks: Tuple[int, ...]*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

class hub.schema.features.**HubSchema**
Base class for all datatypes

__weakref__
list of weak references to the object (if defined)

_flatten() → Iterable[*hub.schema.features.FlatTensor*]
Flattens dtype into list of tensors that will need to be stored seperately

class hub.schema.features.**Primitive**(*dtype*, *chunks=None*, *compressor='lz4'*)
Class for handling primitive datatypes All numpy primitive data types like int32, float64, etc... should be wrapped around this class

__init__(*dtype*, *chunks=None*, *compressor='lz4'*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__()
Return str(self).

_flatten()
Flattens dtype into list of tensors that will need to be stored seperately

class hub.schema.features.**SchemaDict**(*dict_*)
Class for dict branching of a datatype SchemaDict dtype contains str -> dtype associations This way you can describe complex datatypes

__init__(*dict_*)
Initialize self. See help(type(self)) for accurate signature.

__repr__()
Return repr(self).

__str__()
Return str(self).

_flatten()
Flattens dtype into list of tensors that will need to be stored seperately

class hub.schema.features.**Tensor**(*shape: Tuple[int, ...] = None*, *dtype='float64'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
Tensor type in schema Has np-array like structure contains any type of elements (Primitive and non-Primitive)

__init__(*shape: Tuple[int, ...] = None*, *dtype='float64'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)

Parameters

- **shape** (*Tuple[int]*) – Shape of tensor, can contains None(s) meaning the shape can be dynamic Dynamic shape means it can change during editing the dataset
- **dtype** (*SchemaConnector or str*) – dtype of each element in Tensor. Can be Primitive and non-Primitive type
- **max_shape** (*Tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

__repr__()
Return repr(self).

__str__()
Return str(self).

_flatten()
Flattens dtype into list of tensors that will need to be stored seperately

hub.schema.features.**featurify**(*schema*) → *hub.schema.features.HubSchema*
This functions converts naked primitive datatypes and ditcs into Primitives and SchemaDicts That way every node in dtype tree is a SchemaConnector type object

hub.schema.features.**flatten**(*dtype*, *root=''*)
Flattens nested dictionary and returns tuple (dtype, path)

class hub.schema.mask.**Mask**(*shape: Tuple[int, ...] = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for mask

Usage:

```
>>> mask_tensor = Mask(shape=(300, 300, 1))
```

__init__(*shape: Tuple[int, ...] = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
Constructs a Mask HubSchema.

Parameters

- **shape** (*tuple of ints or None*) – Shape in format (height, width, 1)
- **dtype** (*str*) – Dtype of mask array. Default: *uint8*
- **max_shape** (*Tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

class hub.schema.polygon.**Polygon**(*shape: Tuple[int, ...] = None*, *dtype='int32'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for polygon

Usage:

```
>>> polygon_tensor = Polygon(shape=(10, 2))
>>> polygon_tensor = Polygon(shape=(None, 2))
```

__init__(*shape: Tuple[int, ...] = None*, *dtype='int32'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
Constructs a Polygon HubSchema. Args: shape: tuple of ints or None, i.e (None, 2)

Parameters

- **shape** (*tuple of ints or None*) – Shape in format (None, 2)
- **max_shape** (*Tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

Raises **ValueError** – If the shape is invalid:

__repr__()
Return repr(self).

__str__()
Return str(self).

_check_shape(*shape*)
Check if provided shape maches polygon characteristics.

get_attr_dict()
Return class attributes.

class hub.schema.segmentation.**Segmentation**(*shape: Tuple[int, ...] = None*, *dtype: str = None*, *num_classes: int = None*, *names: Tuple[str] = None*, *names_file: str = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for segmentation

__init__(*shape: Tuple[int, ...] = None*, *dtype: str = None*, *num_classes: int = None*, *names: Tuple[str] = None*, *names_file: str = None*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
Constructs a Segmentation HubSchema. Also constructs ClassLabel HubSchema for Segmentation classes.

Parameters

- **shape** (*tuple of ints or None*) – Shape in format (height, width, 1)
- **dtype** (*str*) – dtype of segmentation array: *uint16* or *uint8*
- **num_classes** (*int*) – Number of classes. All labels must be < num_classes.
- **names** (*list<str>*) – string names for the integer classes. The order in which the names are provided is kept.
- **names_file** (*str*) – Path to a file with names for the integer classes, one per line.
- **max_shape** (*tuple[int]*) – Maximum shape of tensor shape if tensor is dynamic
- **chunks** (*tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

get_segmentation_classes()
Get classes of the segmentation mask

class hub.schema.sequence.**Sequence**(*shape=()*, *max_shape=()*, *dtype=None*, *chunks=None*, *compressor='lz4'*)
Sequence correspond to sequence of *features.HubSchema*. At generation time, a list for each of the sequence element is given. The output of *Dataset* will batch all the elements of the sequence together. If the length of the sequence is static and known in advance, it should be specified in the constructor using the *length* param.

Usage:

```
>>> sequence = Sequence(Image(), length=NB_FRAME)
```

__init__(*shape=()*, *max_shape=()*, *dtype=None*, *chunks=None*, *compressor='lz4'*)
Construct a sequence of Tensors. :param shape: Single integer element tuple representing length of sequence

If None then dynamic

Parameters

- **dtype** (*str | HubSchema*) – Datatype of each element in sequence
- **chunks** (*Tuple[int] | int*) – Number of elements in chunk Works only for top level sequence You can also include number of samples in a single chunk

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes

class hub.schema.text.**Text**(*shape: Tuple[int, ...] = None*, *dtype='int64'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for text

__init__(*shape: Tuple[int, ...] = None*, *dtype='int64'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)

Construct the connector.

Parameters

- **shape** (*tuple of ints or None*) – The shape of the text
- **dtype** (*str*) – the dtype for storage.
- **max_shape** (*Tuple[int]*) – Maximum number of words in the text
- **chunks** (*Tuple[int] | True*) – Describes how to split tensor dimensions into chunks (files) to store them efficiently. It is anticipated that each file should be ~16MB. Sample Count is also in the list of tensor's dimensions (first dimension) If default value is chosen, automatically detects how to split into chunks

__repr__()
Return repr(self).

__str__()
Return str(self).

_set_dtype(*dtype*)
Set the dtype.

get_attr_dict()
Return class attributes.

class hub.schema.video.**Video**(*shape: Tuple[int, ...] = None*, *dtype: str = 'uint8'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
HubSchema for videos, encoding frames individually on disk.

The connector accepts as input a 4 dimensional *uint8* array representing a video.

Returns **Tensor** – where channels must be 1 or 3

Return type *uint8* and shape [num_frames, height, width, channels],

__init__(*shape: Tuple[int, ...] = None*, *dtype: str = 'uint8'*, *max_shape: Tuple[int, ...] = None*, *chunks=None*, *compressor='lz4'*)
Initializes the connector.

Parameters

- **shape** (*tuple of ints*) – The shape of the video (num_frames, height, width, channels), where channels is 1 or 3.
- **encoding_format** (*str*) – The video is stored as a sequence of encoded images. You can use any encoding format supported by Image.
- **dtype** (*uint16* or *uint8* (default)) –

Raises **ValueError** – If the shape, dtype or encoding formats are invalid:

__repr__()
Return repr(self).

__str__()
Return str(self).

get_attr_dict()
Return class attributes.

1.7 Dataset

1.7.1 Create

To create and store dataset you would need to define shape and specify the dataset structure (schema).

For example, to create a dataset `basic` with 4 samples containing images and labels with shape (512, 512) of dtype 'float' in account `username`:

```
from hub import Dataset, schema
tag = "username/basic"

ds = Dataset(
    tag,
    shape=(4,),
    schema={
        "image": schema.Tensor((512, 512), dtype="float"),
        "label": schema.Tensor((512, 512), dtype="float"),
    },
)
```

1.7.2 Upload the Data

To add data to the dataset:

```
ds["image"][:] = np.ones((4, 512, 512))
ds["label"][:] = np.ones((4, 512, 512))
ds.commit()
```

1.7.3 Load the data

Load the dataset and access its elements:

```
ds = Dataset('username/basic')

# Use .numpy() to get the numpy array of the element
print(ds["image"][0].numpy())
print(ds["label", 100:110].numpy())
```

1.7.4 Convert to Pytorch

```
ds = ds.to_pytorch()
ds = torch.utils.data.DataLoader(
    ds,
    batch_size=8,
    num_workers=2,
)

# Iterate over the data
for batch in ds:
    print(batch["image"], batch["label"])
```

1.7.5 Convert to Tensorflow

```
ds = ds.to_tensorflow().batch(8)

# Iterate over the data
for batch in ds:
    print(batch["image"], batch["label"])
```

1.7.6 Visualize

Make sure visualization works perfectly at app.activeloop.ai

1.7.7 Issues

If you spot any trouble or have any question, please open a github issue.

1.7.8 API

class hub.**Dataset**(*url: str*, *mode: str = 'a'*, *safe_mode: bool = False*, *shape=None*, *schema=None*, *token=None*, *fs=None*, *fs_map=None*, *cache: int = 67108864*, *storage_cache: int = 268435456*, *lock_cache=True*, *tokenizer=None*)

__getitem__(*slice_*)

Gets a slice or slices from dataset
Usage:

```
>>> return ds["image", 5, 0:1920, 0:1080, 0:3].numpy() # returns numpy array
>>> images = ds["image"]
>>> return images[5].numpy() # returns numpy array
>>> images = ds["image"]
>>> image = images[5]
>>> return image[0:1920, 0:1080, 0:3].numpy()
```

__init__(*url: str*, *mode: str = 'a'*, *safe_mode: bool = False*, *shape=None*, *schema=None*, *token=None*, *fs=None*, *fs_map=None*, *cache: int = 67108864*, *storage_cache: int = 268435456*, *lock_cache=True*, *tokenizer=None*)

Open a new or existing dataset for read/write :param url: The url where dataset is located/should be created :type url: str :param mode: Python way to tell whether dataset is for read or write (ex. "r", "w", "a") :type mode: str, optional (default to "w") :param safe_mode: if dataset exists it cannot be rewritten in safe mode, otherwise it lets to write the first time :type safe_mode: bool, optional :param shape: Tuple with (num_samples,) format, where num_samples is number of samples :type shape: tuple, optional :param schema: Describes the data of a single sample. Hub schemas are used for that

Required for 'a' and 'w' modes

Parameters

- **token** (*str or dict, optional*) – If url is refering to a place where authorization is required, token is the parameter to pass the credentials, it can be filepath or dict
- **fs** (*optional*) –
- **fs_map** (*optional*) –
- **cache** (*int, optional*) – Size of the memory cache. Default is 64MB (2**26) if 0, False or None, then cache is not used
- **storage_cache** (*int, optional*) – Size of the storage cache. Default is 256MB (2**28) if 0, False or None, then storage cache is not used
- **lock_cache** (*bool, optional*) – Lock the cache for avoiding multiprocessing errors

__iter__()
Returns Iterable over samples

__len__()
Number of samples in the dataset

__repr__()
Return repr(self).

__setitem__(*slice_*, *value*)

Sets a slice or slices with a value
Usage >>> ds["image", 5, 0:1920, 0:1080, 0:3] = np.zeros((1920, 1080, 3), "uint8")

```
>>> images = ds["image"]
>>> image = images[5]
>>> image[0:1920, 0:1080, 0:3] = np.zeros((1920, 1080, 3), "uint8")
```

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

_check_and_prepare_dir()
Checks if input data is ok. Creates or overwrites dataset folder. Returns True dataset needs to be created opposed to read.

_get_dictionary(*subpath*, *slice_=None*)
"Gets dictionary from dataset given incomplete subpath

append_shape(*size: int*)
Append the shape: Heavy Operation

close()
Save changes from cache to dataset final storage This invalidates this object

commit()
Deprecated alias to flush()

flush()
Save changes from cache to dataset final storage Does not invalidate this object

static from_pytorch(*dataset*)
Converts a pytorch dataset object into hub format :param dataset: The pytorch dataset object that needs to be converted into hub format

static from_tensorflow(*ds*)
Converts a tensorflow dataset into hub format :param dataset: The tensorflow dataset object that needs to be converted into hub format

Examples

ds = tf.data.Dataset.from_tensor_slices(tf.range(10)) out_ds = hub.Dataset.from_tensorflow(ds) res_ds = out_ds.store("username/new_dataset") # res_ds is now a usable hub dataset

ds = tf.data.Dataset.from_tensor_slices({'a': [1, 2], 'b': [5, 6]}) out_ds = hub.Dataset.from_tensorflow(ds) res_ds = out_ds.store("username/new_dataset") # res_ds is now a usable hub dataset

ds = hub.Dataset(schema=my_schema, shape=(1000,), url="username/dataset_name", mode="w") ds = ds.to_tensorflow() out_ds = hub.Dataset.from_tensorflow(ds) res_ds = out_ds.store("username/new_dataset") # res_ds is now a usable hub dataset

static from_tfds(*dataset*, *split=None*, *num=- 1*, *sampling_amount=1*)
Converts a TFDS Dataset into hub format :param dataset: The name of the tfds dataset that needs to be

converted into hub format :type dataset: str :param split: A string representing the splits of the dataset that are required such as “train” or “test+train”

If not present, all the splits of the dataset are used.

Parameters

- **num** (*int, optional*) – The number of samples required. If not present, all the samples are taken. If count is -1, or if count is greater than the size of this dataset, the new dataset will contain all elements of this dataset.
- **sampling_amount** (*float, optional*) – a value from 0 to 1, that specifies how much of the dataset would be sampled to determinte feature shapes value of 0 would mean no sampling and 1 would imply that entire dataset would be sampled

Examples

out_ds = hub.Dataset.from_tfds(‘mnist’, split=’test+train’, num=1000) res_ds = out_ds.store(“username/mnist”) # res_ds is now a usable hub dataset

property keys
Get Keys of the dataset

resize_shape(*size: int*) → None
Resize the shape of the dataset by resizing each tensor first dimension

to_pytorch(*Transform=None*, *offset=None*, *num_samples=None*)
Converts the dataset into a pytorch compatible format :param offset: The offset from which dataset needs to be converted :type offset: int, optional :param num_samples: The number of samples required of the dataset that needs to be converted :type num_samples: int, optional

to_tensorflow(*offset=None*, *num_samples=None*)
Converts the dataset into a tensorflow compatible format :param offset: The offset from which dataset needs to be converted :type offset: int, optional :param num_samples: The number of samples required of the dataset that needs to be converted :type num_samples: int, optional

1.8 Data Pipelines

Data pipelines are usually a series of data transformations on datasets.

1.8.1 Transform

Hub Transform provides a functionality to modify the samples of the dataset or create a new dataset from the existing one. To apply these modifications user needs to add a `@hub.transform` decorator to any custom generator function.

1.8.2 Examples

Basic transform pipeline creation:

```
my_schema = {
    "image": Tensor((28, 28, 4), "int32", (28, 28, 4)),
    "label": "<U20",
    "confidence": "float",
}

ds = hub.Dataset(
   "./data/test/test_pipeline_basic", mode="w", shape=(100,), schema=my_schema
)

for i in range(len(ds)):
   ds["image", i] = np.ones((28, 28, 4), dtype="int32")
   ds["label", i] = f"hello {i}"
   ds["confidence", i] = 0.2

@hub.transform(schema=my_schema)
def my_transform(sample, multiplier: int = 2):
   return {
      "image": sample["image"].compute() * multiplier,
      "label": sample["label"].compute(),
      "confidence": sample["confidence"].compute() * multiplier
   }

out_ds = my_transform(ds, multiplier=2)
res_ds = out_ds.store("./data/test/test_pipeline_basic_output")
```

Transormation function can return either a dictionary that corresponds to the provided schema or a list of such dictionaries. In that case the number of samples in the final dataset will be equal to the number of all the returned dictionaries:

```
dynamic_schema = {
    "image": Tensor(shape=(None, None, None), dtype="int32", max_shape=(32, 32, 3)),
    "label": "<U20",
}

ds = hub.Dataset(
        "./data/test/test_pipeline_dynamic3", mode="w", shape=(1,), schema=dynamic_
↪schema, cache=False
    )

ds["image", 0] = np.ones((30, 32, 3))

@hub.transform(schema=dynamic_schema, scheduler="threaded", nodes=8)
def dynamic_transform(sample, multiplier: int = 2):
   return [{
      "image": sample["image"].compute() * multiplier,
      "label": sample["label"].compute(),
   } for i in range(4)]

out_ds = dynamic_transform(ds, multiplier=4).store("./data/test/test_pipeline_dynamic_
↪output2")
```

You can use transform with multuple processes by adding `scheduler` and `nodes` arguments:

```
my_schema = {
   "image": Tensor((width, width, channels), dtype, (width, width, channels),␣
↪chunks=(sample_size // 20, width, width, channels)),
}

@hub.transform(schema=my_schema, scheduler="processed", nodes=2)
def my_transform(x):

   a = np.random.random((width, width, channels))
   for i in range(10):
         a *= np.random.random((width, width, channels))

   return {
         "image": (np.ones((width, width, channels), dtype=dtype) * 255),
   }

ds = hub.Dataset(
   "./data/test/test_pipeline_basic_4", mode="w", shape=(sample_size,), schema=my_
↪schema, cache=0
)

ds_t = my_transform(ds).store("./data/test/test_pipeline_basic_4")
```

1.8.3 Ray Transform

There is also an option of using `ray` as a scheduler. In this case `RayTransform` will be applied to samples.

```
ds = hub.Dataset(
   "./data/ray/ray_pipeline_basic",
   mode="w",
   shape=(100,),
   schema=my_schema,
   cache=False,
)

for i in range(len(ds)):
   ds["image", i] = np.ones((28, 28, 4), dtype="int32")
   ds["label", i] = f"hello {i}"
   ds["confidence/confidence", i] = 0.2

@hub.transform(schema=my_schema, scheduler="ray")
def my_transform(sample, multiplier: int = 2):
   return {
      "image": sample["image"].compute() * multiplier,
      "label": sample["label"].compute(),
      "confidence": {
            "confidence": sample["confidence/confidence"].compute() * multiplier
      },
   }

out_ds = my_transform(ds, multiplier=2)
```

1.8.4 API

hub.compute.**transform**(*schema*, *scheduler='single'*, *workers=1*)
Transform is a decorator of a function. The function should output a dictionary per sample

Parameters

schema: Schema The output format of the transformed dataset

scheduler: str "single" - for single threaded, "threaded" using multiple threads, "processed", "ray" scheduler, "dask" scheduler

workers: int how many workers will be started for the process

class hub.compute.transform.**Transform**(*func*, *schema*, *ds*, *scheduler: str = 'single'*, *workers: int = 1*, ***kwargs*)

__getitem__(*slice_*)
Get an item to be computed without iterating on the whole dataset Creates a dataset view, then a temporary dataset to apply the transform

slice_: slice Gets a slice or slices from dataset

__init__(*func*, *schema*, *ds*, *scheduler: str = 'single'*, *workers: int = 1*, ***kwargs*)
Transform applies a user defined function to each sample in single threaded manner

Parameters

- **func** (*function*) – user defined function func(x, **kwargs)
- **schema** (*dict of dtypes*) – the structure of the final dataset that will be created
- **ds** (*Iterative*) – input dataset or a list that can be iterated
- **scheduler** (*str*) – choice between "single", "threaded", "processed"
- **workers** (*int*) – how many threads or processes to use
- ****kwargs** – additional arguments that will be passed to func as static argument for all samples

__weakref__
list of weak references to the object (if defined)

classmethod _flatten(*items*, *schema*)
Takes a dictionary or list of dictionary Returns a dictionary of concatenated values Dictionary follows schema

classmethod _flatten_dict(*d: Dict*, *parent_key=''*, *schema=None*)
Helper function to flatten dictionary of a recursive tensor

Parameters **d** (*dict*) –

_pbar(*show: bool = True*)
Returns a progress bar, if empty then it function does nothing

_split_list_to_dicts(*xs*)
Helper function that transform list of dicts into dicts of lists

Parameters **xs** (*list of dicts*) –

Returns xs_new

Return type dicts of lists

classmethod _unwrap(*results*)
If there is any list then unwrap it into its elements

create_dataset(*url*, *length=None*, *token=None*)
Helper function to creat a dataset

classmethod dtype_from_path(*path*, *schema*)
Helper function to get the dtype from the path

store(*url: str*, *token: dict = None*, *length: int = None*, *ds: Iterable = None*, *progressbar: bool = True*, *sample_per_shard=None*)
The function to apply the transformation for each element in batchified manner

Parameters

- **url** (*str*) – path where the data is going to be stored
- **token** (*str or dict, optional*) – If url is refering to a place where authorization is required, token is the parameter to pass the credentials, it can be filepath or dict
- **length** (*int*) – in case shape is None, user can provide length
- **ds** (*Iterable*) –
- **progressbar** (*bool*) – Show progress bar
- **sample_per_shard** (*int*) – How to split the iterator not to overfill RAM

Returns **ds** – uploaded dataset

Return type *hub.Dataset*

store_shard(*ds_in: Iterable*, *ds_out: hub.api.dataset.Dataset*, *offset: int*, *token=None*)
Takes a shard of iteratable ds_in, compute and stores in DatasetView

upload(*results*, *ds: hub.api.dataset.Dataset*, *token: dict*, *progressbar: bool = True*)
Batchified upload of results For each tensor batchify based on its chunk and upload If tensor is dynamic then still upload element by element For dynamic tensors, it disable dynamicness and then enables it back

Parameters

- **dataset** (hub.Dataset) – Dataset object that should be written to
- **results** – Output of transform function
- **progressbar** (*bool*) –

Returns **ds** – Uploaded dataset

Return type *hub.Dataset*

class hub.compute.ray.**RayTransform**(*func*, *schema*, *ds*, *scheduler='ray'*, *workers=1*, ***kwargs*)

__init__(*func*, *schema*, *ds*, *scheduler='ray'*, *workers=1*, ***kwargs*)
Transform applies a user defined function to each sample in single threaded manner

Parameters

- **func** (*function*) – user defined function func(x, **kwargs)
- **schema** (*dict of dtypes*) – the structure of the final dataset that will be created
- **ds** (*Iterative*) – input dataset or a list that can be iterated
- **scheduler** (*str*) – choice between “single”, “threaded”, “processed”
- **workers** (*int*) – how many threads or processes to use

- ****kwargs** – additional arguments that will be passed to func as static argument for all samples

store(*url: str*, *token: dict = None*, *length: int = None*, *ds: Iterable = None*, *progressbar: bool = True*)
The function to apply the transformation for each element in batchified manner

Parameters

- **url** (*str*) – path where the data is going to be stored
- **token** (*str or dict, optional*) – If url is refering to a place where authorization is required, token is the parameter to pass the credentials, it can be filepath or dict
- **length** (*int*) – in case shape is None, user can provide length
- **ds** (*Iterable*) –
- **progressbar** (*bool*) – Show progress bar

Returns **ds** – uploaded dataset

Return type *hub.Dataset*

upload(*results*, *url: str*, *token: dict*, *progressbar: bool = True*)
Batchified upload of results For each tensor batchify based on its chunk and upload If tensor is dynamic then still upload element by element

Parameters

- **dataset** (hub.Dataset) – Dataset object that should be written to
- **results** – Output of transform function
- **progressbar** (*bool*) –

Returns **ds** – Uploaded dataset

Return type *hub.Dataset*

1.9 PyTorch

1.9.1 Dataset to PyTorch Dataset

Here is an example to transform the dataset into Pytorch form.

```
import torch
from hub import dataset

# Create dataset
ds = Dataset(
    "username/pytorch_example",
    shape=(640,),
    mode="w",
    schema={
        "image": schema.Tensor((512, 512), dtype="float"),
        "label": schema.Tensor((512, 512), dtype="float"),
    },
)

# Transform into Pytorch
ds = ds.to_pytorch(transform=None)
```

(continues on next page)

(continued from previous page)

```
ds = torch.utils.data.DataLoader(
    ds,
    batch_size=8,
    num_workers=2,
)

# Iterate
for batch in ds:
    print(batch["image"], batch["label"])
```

1.9.2 PyTorch Dataset to Dataset

You can also use `.from_pytorch()` to convert a PyTorch Dataset into Hub format.

```
from torch.utils.data import Dataset

class TorchDataset(Dataset):
    def __init__(self, transform=None):
        self.transform = transform

    def __len__(self):
        return 12

    def __iter__(self):
        for i in range(len(self)):
            yield self[i]

    def __getitem__(self, idx):
        image = 5 * np.ones((50, 50))
        landmarks = 7 * np.ones((10, 10, 10))
        named = "testing text labels"
        sample = {
            "data": {"image": image, "landmarks": landmarks},
            "labels": {"named": named},
        }

        if self.transform:
            sample = self.transform(sample)
        return sample

tds = TorchDataset()
ds = hub.Dataset.from_pytorch(tds)
```

1.10 Tensorflow

1.10.1 Dataset to Tensorflow Dataset

Here is an example to transform the dataset into Tensorflow form.

```
from hub import Dataset

# Create dataset
```

(continues on next page)

(continued from previous page)

```
ds = Dataset(
    "username/tensorflow_example",
    shape=(64,),
    schema={
        "image": schema.Tensor((512, 512), dtype="float"),
        "label": schema.Tensor((512, 512), dtype="float"),
    },
)

# transform into Tensorflow dataset
ds = ds.to_tensorflow().batch(8)

# Iterate over the data
for batch in ds:
    print(batch["image"], batch["label"])
```

1.10.2 Tensorflow Dataset to Dataset

Hub dataset can be created from tensorflow dataset:

```
import tensorflow as tf
ds = tf.data.Dataset.from_tensor_slices(tf.range(10))
out_ds = hub.Dataset.from_tensorflow(ds)
res_ds = out_ds.store("./data/from_tf/ds")
```

1.10.3 TFDS Dataset to Dataset

Also, it is possible to load a dataset using tensorflow_datasets:

```
import tensorflow_datasets as tfds
with tfds.testing.mock_data(num_examples=5):
    ds = hub.Dataset.from_tfds('mnist', num=5)
    res_ds = ds.store("./data/tfds/mnist", length=5)
```

1.11 Discussions

- Join our Slack community for help from Activeloop team and other users as well as dataset management/preprocessing tips and tricks.
- For feature requests or bug reports, please open a new GitHub issue.
- genindex
- modindex
- search

PYTHON MODULE INDEX

h

hub.compute, 40
hub.schema.features, 28
hub.schema.serialize, 13

INDEX

Symbols

__getitem__() (*hub.Dataset method*), 5, 35
__getitem__() (*hub.api.datasetview.DatasetView method*), 8
__getitem__() (*hub.api.tensorview.TensorView method*), 9
__getitem__() (*hub.compute.transform.Transform method*), 11, 40
__init__() (*hub.Dataset method*), 5, 35
__init__() (*hub.api.datasetview.DatasetView method*), 8
__init__() (*hub.api.sharded_datasetview.ShardedDatasetView method*), 10
__init__() (*hub.api.tensorview.TensorView method*), 9
__init__() (*hub.compute.ray.RayTransform method*), 12, 41
__init__() (*hub.compute.transform.Transform method*), 11, 40
__init__() (*hub.schema.audio.Audio method*), 13, 25
__init__() (*hub.schema.bbox.BBox method*), 14, 26
__init__() (*hub.schema.class_label.ClassLabel method*), 14, 26
__init__() (*hub.schema.features.FlatTensor method*), 16, 28
__init__() (*hub.schema.features.Primitive method*), 16, 28
__init__() (*hub.schema.features.SchemaDict method*), 17, 29
__init__() (*hub.schema.features.Tensor method*), 17, 29
__init__() (*hub.schema.image.Image method*), 15, 27
__init__() (*hub.schema.mask.Mask method*), 18, 30
__init__() (*hub.schema.polygon.Polygon method*), 18, 30
__init__() (*hub.schema.segmentation.Segmentation method*), 19, 31
__init__() (*hub.schema.sequence.Sequence method*), 20, 32
__init__() (*hub.schema.text.Text method*), 20, 32
__init__() (*hub.schema.video.Video method*), 21, 33
__iter__() (*hub.Dataset method*), 6, 35
__iter__() (*hub.api.datasetview.DatasetView method*), 8
__iter__() (*hub.api.sharded_datasetview.ShardedDatasetView method*), 10
__len__() (*hub.Dataset method*), 6, 35
__repr__() (*hub.Dataset method*), 6, 35
__repr__() (*hub.api.datasetview.DatasetView method*), 8
__repr__() (*hub.api.sharded_datasetview.ShardedDatasetView method*), 10
__repr__() (*hub.api.tensorview.TensorView method*), 9
__repr__() (*hub.schema.audio.Audio method*), 14, 26
__repr__() (*hub.schema.bbox.BBox method*), 14, 26
__repr__() (*hub.schema.class_label.ClassLabel method*), 15, 27
__repr__() (*hub.schema.features.Primitive method*), 16, 28
__repr__() (*hub.schema.features.SchemaDict method*), 17, 29
__repr__() (*hub.schema.features.Tensor method*), 17, 29
__repr__() (*hub.schema.image.Image method*), 16, 28
__repr__() (*hub.schema.mask.Mask method*), 18, 30
__repr__() (*hub.schema.polygon.Polygon method*), 18, 30
__repr__() (*hub.schema.segmentation.Segmentation method*), 19, 31
__repr__() (*hub.schema.sequence.Sequence method*), 20, 32
__repr__() (*hub.schema.text.Text method*), 20, 32
__repr__() (*hub.schema.video.Video method*), 21, 33
__setitem__() (*hub.Dataset method*), 6, 36
__setitem__() (*hub.api.datasetview.DatasetView method*), 8
__setitem__() (*hub.api.tensorview.TensorView method*), 9
__str__() (*hub.Dataset method*), 6, 36
__str__() (*hub.api.datasetview.DatasetView method*), 8

__str__() (*hub.api.tensorview.TensorView method*), 9
__str__() (*hub.schema.audio.Audio method*), 14, 26
__str__() (*hub.schema.bbox.BBox method*), 14, 26
__str__() (*hub.schema.class_label.ClassLabel method*), 15, 27
__str__() (*hub.schema.features.Primitive method*), 16, 28
__str__() (*hub.schema.features.SchemaDict method*), 17, 29
__str__() (*hub.schema.features.Tensor method*), 17, 29
__str__() (*hub.schema.image.Image method*), 16, 28
__str__() (*hub.schema.mask.Mask method*), 18, 30
__str__() (*hub.schema.polygon.Polygon method*), 18, 30
__str__() (*hub.schema.segmentation.Segmentation method*), 19, 31
__str__() (*hub.schema.sequence.Sequence method*), 20, 32
__str__() (*hub.schema.text.Text method*), 20, 32
__str__() (*hub.schema.video.Video method*), 21, 33
__weakref__ (*hub.Dataset attribute*), 6, 36
__weakref__ (*hub.api.datasetview.DatasetView attribute*), 8
__weakref__ (*hub.api.sharded_datasetview.ShardedDatasetView attribute*), 10
__weakref__ (*hub.api.tensorview.TensorView attribute*), 9
__weakref__ (*hub.compute.transform.Transform attribute*), 11, 40
__weakref__ (*hub.schema.features.FlatTensor attribute*), 16, 28
__weakref__ (*hub.schema.features.HubSchema attribute*), 16, 28
_check_and_prepare_dir() (*hub.Dataset method*), 6, 36
_check_shape() (*hub.schema.polygon.Polygon method*), 18, 30
_combine() (*hub.api.tensorview.TensorView method*), 9
_flatten() (*hub.compute.transform.Transform class method*), 11, 40
_flatten() (*hub.schema.features.HubSchema method*), 16, 28
_flatten() (*hub.schema.features.Primitive method*), 16, 28
_flatten() (*hub.schema.features.SchemaDict method*), 17, 29
_flatten() (*hub.schema.features.Tensor method*), 17, 29
_flatten_dict() (*hub.compute.transform.Transform class method*), 11, 40
_get_dictionary() (*hub.Dataset method*), 6, 36
_get_dictionary() (*hub.api.datasetview.DatasetView method*), 8
_pbar() (*hub.compute.transform.Transform method*), 11, 40
_set_dtype() (*hub.schema.image.Image method*), 16, 28
_set_dtype() (*hub.schema.text.Text method*), 20, 32
_set_encoding_format() (*hub.schema.image.Image method*), 16, 28
_split_list_to_dicts() (*hub.compute.transform.Transform method*), 11, 40
_unwrap() (*hub.compute.transform.Transform class method*), 11, 40

A

append_shape() (*hub.Dataset method*), 6, 36
Audio (*class in hub.schema.audio*), 13, 25

B

BBox (*class in hub.schema.bbox*), 14, 26

C

check_slice_bounds() (*hub.api.tensorview.TensorView method*), 9
ClassLabel (*class in hub.schema.class_label*), 14, 26
close() (*hub.Dataset method*), 6, 36
commit() (*hub.api.datasetview.DatasetView method*), 8
commit() (*hub.Dataset method*), 6, 36
compute() (*hub.api.tensorview.TensorView method*), 9
create_dataset() (*hub.compute.transform.Transform method*), 11, 41

D

Dataset (*class in hub*), 5, 35
DatasetView (*class in hub.api.datasetview*), 8
dtype_from_path() (*hub.api.tensorview.TensorView method*), 10
dtype_from_path() (*hub.compute.transform.Transform class method*), 11, 41

F

featurify() (*in module hub.schema.features*), 17, 29
flatten() (*in module hub.schema.features*), 17, 29
FlatTensor (*class in hub.schema.features*), 16, 28
flush() (*hub.Dataset method*), 6, 36
from_pytorch() (*hub.Dataset static method*), 6, 36
from_tensorflow() (*hub.Dataset static method*), 6, 36
from_tfds() (*hub.Dataset static method*), 7, 36

G

get_attr_dict() (*hub.schema.audio.Audio method*), 14, 26
get_attr_dict() (*hub.schema.bbox.BBox method*), 14, 26
get_attr_dict() (*hub.schema.class_label.ClassLabel method*), 15, 27
get_attr_dict() (*hub.schema.image.Image method*), 16, 28
get_attr_dict() (*hub.schema.mask.Mask method*), 18, 30
get_attr_dict() (*hub.schema.polygon.Polygon method*), 19, 31
get_attr_dict() (*hub.schema.segmentation.Segmentation method*), 19, 31
get_attr_dict() (*hub.schema.sequence.Sequence method*), 20, 32
get_attr_dict() (*hub.schema.text.Text method*), 20, 32
get_attr_dict() (*hub.schema.video.Video method*), 21, 33
get_segmentation_classes() (*hub.schema.segmentation.Segmentation method*), 19, 31

H

hub.compute
    module, 10, 40
hub.schema.features
    module, 16, 28
hub.schema.serialize
    module, 13
HubSchema (*class in hub.schema.features*), 16, 28

I

identify_shard() (*hub.api.sharded_datasetview.ShardedDatasetView method*), 10
Image (*class in hub.schema.image*), 15, 27
int2str() (*hub.schema.class_label.ClassLabel method*), 15, 27

K

keys() (*hub.api.datasetview.DatasetView property*), 8
keys() (*hub.Dataset property*), 7, 37

M

Mask (*class in hub.schema.mask*), 17, 29
module
    hub.compute, 10, 40
    hub.schema.features, 16, 28
    hub.schema.serialize, 13

N

numpy() (*hub.api.tensorview.TensorView method*), 10

P

Polygon (*class in hub.schema.polygon*), 18, 30
Primitive (*class in hub.schema.features*), 16, 28

R

RayTransform (*class in hub.compute.ray*), 12, 41
resize_shape() (*hub.api.datasetview.DatasetView method*), 8
resize_shape() (*hub.Dataset method*), 7, 37

S

SchemaDict (*class in hub.schema.features*), 17, 29
Segmentation (*class in hub.schema.segmentation*), 19, 31
Sequence (*class in hub.schema.sequence*), 19, 31
serialize() (*in module hub.schema.serialize*), 13
serialize_primitive() (*in module hub.schema.serialize*), 13
serialize_SchemaDict() (*in module hub.schema.serialize*), 13
serialize_tensor() (*in module hub.schema.serialize*), 13
ShardedDatasetView (*class in hub.api.sharded_datasetview*), 10
slice_fill() (*hub.api.tensorview.TensorView method*), 10
slicing() (*hub.api.sharded_datasetview.ShardedDatasetView method*), 10
store() (*hub.compute.ray.RayTransform method*), 12, 42
store() (*hub.compute.transform.Transform method*), 11, 41
store_shard() (*hub.compute.transform.Transform method*), 12, 41
str2int() (*hub.schema.class_label.ClassLabel method*), 15, 27

T

Tensor (*class in hub.schema.features*), 17, 29
TensorView (*class in hub.api.tensorview*), 9
Text (*class in hub.schema.text*), 20, 32
to_pytorch() (*hub.api.datasetview.DatasetView method*), 9
to_pytorch() (*hub.Dataset method*), 7, 37
to_tensorflow() (*hub.api.datasetview.DatasetView method*), 9
to_tensorflow() (*hub.Dataset method*), 7, 37
Transform (*class in hub.compute.transform*), 10, 40
transform() (*in module hub.compute*), 10, 40

U

upload() (*hub.compute.ray.RayTransform method*), 13, 42

upload() (*hub.compute.transform.Transform method*), 12, 41

V

Video (*class in hub.schema.video*), 20, 32