Activeloop

Release 0.9

Activeloop

Sep 04, 2020

OVERVIEW

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
```

Upgrade

or in case you would like to upgrade it

```
pip3 install --upgrade hub
```

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 pipelies 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 Tensor

class hub.dataset.**Tensor**(*meta: dict*, *daskarray*, *delayed_objs: tuple = None*)

compute()
Does lazy computation and converts data to numpy array :returns: numpy array of tensor's data :rtype: np.ndarray

property count
returns: Number of elements on axis 0 if that number is known, -1 otherwise :rtype: int

property dtag
returns: Information about data stored in tensor (iamge, mask, label, . . .) :rtype: str

property dtype
returns: Tensor data type, equivalent of numpy dtype :rtype: str

property meta
returns: metadata dict of tensor :rtype: dict

property ndim
returns: Number of dimensions (len(shape)) :rtype: int

property shape
returns: Tensor shape :rtype: tuple

1.4.2 Dataset

class hub.dataset.**Dataset**(*tensors: Dict[str, hub.collections.tensor.core.Tensor]*, *metainfo={}*)

property citation
Dataset citation

property count
len of dataset (len of tensors across axis 0, yes, they all should be = to each other) Returns -1 if length is unknown

delete(*tag*, *creds=None*, *session_creds=True*) → bool
Deletes dataset given tag(filepath) and credentials (optional)

property description
Dataset description

property howtoload
Dataset howtoload

items()
Returns tensors

keys()
Returns names of tensors

property license
Dataset license

property meta
Dict of meta's of each tensor meta of tensor contains all metadata for tensor storage

store(*tag*, *creds=None*, *session_creds=True*) → hub.collections.dataset.core.Dataset
Stores dataset by tag(filepath) given credentials (can be omitted)

to_pytorch(*transform=None*)
Transforms into pytorch dataset

Parameters **transform** (*func*) – any transform that takes input a dictionary of a sample and returns transformed dictionary

to_tensorflow()
Transforms into tensorflow dataset

values()
Returns tensors

1.4.3 Transform

class hub.**Transform**

forward()
Takes a an element of a list or sample from dataset and returns sample of the dataset

Parameters **input** – an element of list or dict of arrays

Returns dict of numpy arrays

Return type dict

Examples

```
>>> def forward(input):
>>>     ds = {}
>>>     ds["image"] = np.empty(1, object)
>>>     ds["image"][0] = np.array(256, 256)
>>>     return ds
```

meta()
Provides the metadata for all tensors including shapes, dtypes, dtags and chunksize for each array in the form

Returns dict of tensor

Return type returns

Examples

```
>>> def meta()
>>>     return {
>>>         ...
>>>         "tesnor_name":{
>>>             "shape": (1,256,256),
>>>             "dtype": "uint8",
>>>             "chunksize": 100,
>>>             "dtag": "segmentation"
>>>         }
>>>         ...
>>>     }
```

1.4.4 Load

class hub.**load**(*tag*, *creds=None*, *session_creds=True*)
Load a dataset from repository using given url and credentials (optional)

1.4.5 Init

class hub.**init**(*token: str = ''*, *cloud=False*, *n_workers=1*, *memory_limit=None*, *processes=False*, *threads_per_worker=1*, *distributed=True*)
Initializes cluster either local or on the cloud

Parameters

- **token** (*str*) – token provided by snark
- **cache** (*float*) – Amount on local memory to cache locally, default 2e9 (2GB)
- **cloud** (*bool*) – Should be run locally or on the cloud
- **n_workers** (*int*) – number of concurrent workers, default to1
- **threads_per_worker** (*int*) – Number of threads per each worker

1.5 Getting Started with Hub

1.5.1 Access public data. Fast

We've talked the talk, now let's walk through how it works:

```
pip3 install hub
```

You can access public datasets with a few lines of code.

```
import hub

mnist = hub.load("mnist/mnist")
mnist["data"][0:1000].compute()
```

1.5.2 Train a model

Load the data and directly train your model using pytorch

```
import hub
import pytorch

cifar = hub.load("cifar/cifar10")
cifar = cifar.to_pytorch()

train_loader = torch.utils.data.DataLoader(
      cifar, batch_size=1, num_workers=0, collate_fn=cifar.collate_fn
)

for images, labels in train_loader:
   # your training loop here
```

1.5.3 Upload your dataset and access it from anywhere

Register a free account at Activeloop

```
hub login
```

Then create a dataset and upload

```
from hub import tensor, dataset

images = tensor.from_array(np.zeros((4, 512, 512)))
labels = tensor.from_array(np.zeros((4, 512, 512)))

ds = dataset.from_tensors({"images": images, "labels": labels})
ds.store("username/basic")

# Access it from anywhere else in the world
import hub
ds = hub.load("username/basic")
```

1.6 Tensor

Hub Tensors are scalable NumPy-like arrays stored on the cloud accessible over the internet as if they're local NumPy arrays. Their chunkified structure makes it super fast to interact with them.

Tensor represents a single array containing homogeneous data type. It could contain a list of text files, audio, images, or video data. The first dimension represents the batch dimension.

One can specify `dtag` for the element to specify its nature.

1.6.1 Initialize

You can initialize a tensor like this and get the first element.

```
from hub import tensor

t = tensor.from_zeros((10, 512, 512), dtype="uint8")
t[0].compute()
```

You can also initialize the tensor object from a numpy array.

```
import numpy as np
from hub import tensor

t = tensor.from_zeros(np.zeros((10, 512, 512)))
```

1.6.2 Concat or Stack

Concating or stacking tensors works as in other frameworks.

```
from hub import tensor

t1 = tensor.from_zeros((10, 512, 512), dtype="uint8")
t2 = tensor.from_zeros((20, 512, 512), dtype="uint8")
tensors = [t1, t2]

tensor.concat(tensors, axis=0, chunksize=-1)
tensor.stack(tensors, axis=0, chunksize=-1)
```

1.6.3 API

class hub.dataset.**Tensor**(*meta: dict*, *daskarray*, *delayed_objs: tuple = None*)

compute()
Does lazy computation and converts data to numpy array :returns: numpy array of tensor's data :rtype: np.ndarray

property count
returns: Number of elements on axis 0 if that number is known, -1 otherwise :rtype: int

property dtag
returns: Information about data stored in tensor (iamge, mask, label, . . .) :rtype: str

property dtype
returns: Tensor data type, equivalent of numpy dtype :rtype: str

property meta
returns: metadata dict of tensor :rtype: dict

property ndim
returns: Number of dimensions (len(shape)) :rtype: int

property shape
returns: Tensor shape :rtype: tuple

1.7 Dataset

Hub Datasets are dictionaries containing tensors. You can think of them as folders in the cloud. To store tensor in the cloud we first should put it in dataset and then store the dataset.

1.7.1 Store

To create and store dataset you would need to define tensors and specify the dataset dictionary.

```
from hub import dataset, tensor

tensor1 = tensor.from_zeros((20,512,512), dtype="uint8", dtag="image")
tensor2 = tensor.from_zeros((20), dtype="bool", dtag="label")

dataset.from_tensors({"name1": tensor1, "name2": tensor2})

dataset.store("username/namespace")
```

1.7.2 Load

To load a dataset from a central repository

```
from hub import dataset

ds = dataset.load("mnist/mnist")
```

1.7.3 Combine

You could combine datasets or concat them.

```
from hub import dataset

...

#vertical
dataset.concat(ds1, ds2)

#horizontal
dataset.combine(ds1, ds2)
```

1.7.4 How to Upload a Dataset

For small datasets that would fit into your RAM you can directly upload by converting a numpy array into hub tensor. For complete example please check Uploading MNIST and Uploading CIFAR

For larger datasets you would need to define a dataset generator and apply the transformation iteratively. Please see an example below Uploading COCO. Please pay careful attention to `meta(...)` function where you describe each tensor properties. Please pay careful attention providing full meta description including shape, dtype, dtag, chunk_shape etc.

Dtag

For each tensor you would need to specify a dtag so that visualizer knows how draw it or transformations have context how to transform it.

Guidelines

1. Fork the github repo and create a folder under `examples/dataset`
2. Train a model using Pytorch

```
import hub
import pytorch

ds = hub.load("username/dataset")
ds = ds.to_pytorch()

# Implement a training loop for the dataset in pytorch
...
```

1. Train a model using Tensorflow

```
import hub
import tensorflow

ds = hub.load("username/dataset")
ds = ds.to_tensorflow()

# Implement a training loop for the dataset in tensorflow
...
```

1. Make sure visualization works perfectly at app.activeloop.ai

Final Checklist

So here is the checklist, the pull request.

- Accessible using the sdk
- Trainable on Tensorflow
- Trainable on PyTorch
- Visualizable at app.activeloop.ai
- Pull Request merged into master

Issues

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

1.7.5 API

class hub.dataset.**Dataset**(*tensors: Dict[str, hub.collections.tensor.core.Tensor]*, *metainfo={}*)

property citation
Dataset citation

property count
len of dataset (len of tensors across axis 0, yes, they all should be = to each other) Returns -1 if length is unknown

delete(*tag*, *creds=None*, *session_creds=True*) → bool
Deletes dataset given tag(filepath) and credentials (optional)

property description
Dataset description

property howtoload
Dataset howtoload

items()
Returns tensors

keys()
Returns names of tensors

property license
Dataset license

property meta
Dict of meta's of each tensor meta of tensor contains all metadata for tensor storage

store(*tag*, *creds=None*, *session_creds=True*) → hub.collections.dataset.core.Dataset
Stores dataset by tag(filepath) given credentials (can be omitted)

to_pytorch(*transform=None*)
Transforms into pytorch dataset

Parameters **transform** (*func*) – any transform that takes input a dictionary of a sample and returns transformed dictionary

to_tensorflow()
Transforms into tensorflow dataset

values()
Returns tensors

1.8 Data Pipelines

Data pipelines are usually a series of data transformations on datasets. User needs to implement the transformation in the dataset generator form.

1.8.1 Transform

Hub Transform are user-defined classes that implement Hub Transform interface. You can think of them as user-defined data transformations that stand as nodes from which the data pipelines are constructed.

Transform interface looks like this.

```
class Transform:

    def forward(self, input):
        raise NotImplementedError()

    def meta(self):
        raise NotImplementedError()
```

then you can apply the function on a list or a `hub.dataset` object. `.generate()` function returns a dataset object. Note that all computations are done in lazy mode, and in order to get the final dataset we need to call the compute method.

```
from hub import dataset

ids = [1,2,3]
croped_images = dataset.generate(Transform(), ids)
croped_images.compute()
```

You can stack multiple transformations together before calling compute function.

```
from hub import dataset

ids = [1,2,3]
croped_images = dataset.generate(Transform1(), croped_images)
flipped_images = dataset.generate(Transform2(), ids)
flipped_images.compute()
```

To make it easier to comprehend, let's discuss an example.

1.8.2 Example

Let's say you have a set of images and want to crop the center and then flip them. You also want to execute this data pipeline in parallel on all samples of your dataset.

1. Implement `Crop(Transform)` class that describes how to crop one image.

   We assume we want to crop 256 * 256 rectangle. Then meta should indicate that in output we are going to have one 2 dimensional array with 256 * 256 shape. The call function should implement the actual crop functionality.

   ```
   from hub import Transform

   class Crop(Transform):
      def forward(self, input):
   ```

(continues on next page)

(continued from previous page)

```
        return {"image": input[0:1, :256, :256]}

    def meta(self):
        return {"image": {"shape": (1, 256, 256), "dtype": "uint8"}}
```

2. Implement `Flip(Transform)` class that describes how to flip one image.

```
class Flip(Transform):
    def forward(self, input):
        img = np.expand_dims(input["image"], axis=0)
        img = np.flip(img, axis=(1, 2))
        return {"image": img}

    def meta(self):
        return {"image": {"shape": (1, 256, 256), "dtype": "uint8"}}
```

3. Apply those transformations on the dataset.

```
from hub import dataset

images = [np.ones((1, 512, 512), dtype="uint8") for i in range(20)]
ds = dataset.generate(Crop(), images)
ds = dataset.generate(Flip(), ds)
ds.store("/tmp/cropflip")
```

Special care need to be taken for `meta` information and output dimensions of each sample in `forward` pass. We are planning to simplify this process. Any recommendation as Git issue would be greatly appreciated.

1.8.3 API

class hub.**Transform**

forward()

Takes a an element of a list or sample from dataset and returns sample of the dataset

Parameters **input** – an element of list or dict of arrays

Returns dict of numpy arrays

Return type dict

Examples

```
>>> def forward(input):
>>>     ds = {}
>>>     ds["image"] = np.empty(1, object)
>>>     ds["image"][0] = np.array(256, 256)
>>>     return ds
```

meta()

Provides the metadata for all tensors including shapes, dtypes, dtags and chunksize for each array in the form

Returns dict of tensor

Return type returns

Examples

```
>>> def meta()
>>>     return {
>>>         ...
>>>         "tesnor_name":{
>>>             "shape": (1,256,256),
>>>             "dtype": "uint8",
>>>             "chunksize": 100,
>>>             "dtag": "segmentation"
>>>         }
>>>         ...
>>>     }
```

1.9 PyTorch

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

```
import torch
from hub import dataset

# Load data
ds = dataset.load("mnist/mnist")

# Transform into pytorch
ds = ds.to_pytorch(transform=None)
ds = torch.utils.data.DataLoader(
    ds, batch_size=8, num_workers=8, collate_fn=ds.collate_fn
)

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

Please make sure that `collate_fn` is provided from the dataset `ds.collate_fn` to stack tensors together since they are in dictionary form

1.10 Tensorflow

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

```
from hub import dataset

# Load data
ds = dataset.load("mnist/mnist")

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

# Iterate over the data
```

(continues on next page)

(continued from previous page)

```
for batch in ds:
    print(batch["data"], batch["labels"])
```

- genindex
- modindex
- search

INDEX