COX

Release 0.1

Oct 10, 2020

Contents

Cox is a lightweight, serverless framework for designing and managing experiments. Inspired by our own struggles with ad-hoc filesystem-based experiment collection, and our inability to use heavy-duty frameworks, Cox aims to be a minimal burden while inducing more organization. Created by Logan Engstrom and Andrew Ilyas.

Cox works by helping you easily *log*, *collect*, and *analyze* experimental results.

Why "Cox"? (Aside): The name Cox draws both from Coxswain, the person in charge of steering the boat in a rowing crew, and from the name of Gertrude Cox, a pioneer of experimental design.

CHAPTER 1

Quick Logging Overview

The cox logging system is designed for dealing with repeated experiments. The user defines schemas for Pandas dataframes that contain all the data necessary for each experiment instance. Each experiment ran corresponds to a *data store*, and each specified dataframe from above corresponds to a table within this store. The experiment stores are organized within the same directory. Cox has a number of utilities for running and collecting data from experiments of this nature.

1.1 Walkthroughs

1.1.1 Walkthrough 1: Logging and Reading Data

Note: All of the code for this walkthrough is available here

In this walkthrough, we'll be starting with the following simple piece of code, which tries to finds the minimum of a quadratic function:

```
import sys

def f(x):
    return (x - 2.03)**2 + 3

x = ...
tol = ...
step = ...

for _ in range(1000):
    # Take a uniform step in the direction of decrease
    if f(x + step) < f(x - step):
        x += step
    else:
        x -= step
```

(continues on next page)

(continued from previous page)

```
    # If the difference between the directions
    # is less than the tolerance, stop
    if f(x + step) - f(x - step) < tol:
        break
```

Initializing stores

Logging in Cox is done through the *Store* class, which can be created as follows:

```
from cox.store import Store
# rest of program here...
store = Store(OUT_DIR)
```

Upon construction, the *Store* instance creates a directory with a random `uuid` generated name in `OUT_DIR`, a `HDFStore` for storing data, some logging files, and a tensorboard directory (named `tensorboard`). Therefore, after we run this command, our `OUT_DIR` directory should look something like this:

```
$ ls OUT_DIR
7753a944-568d-4cc2-9bb2-9019cc0b3f49
$ ls 7753a944-568d-4cc2-9bb2-9019cc0b3f49
save        store.h5    tensorboard
```

The experiment ID string `7753a944-568d-4cc2-9bb2-9019cc0b3f49` was autogenerated. If we wanted to name the experiment something else, we could pass it as the second parameter; i.e. making a store with `Store(OUT_DIR, 'exp1')` would make the corresponding experiment ID `exp1`.

Creating tables

The next step is to declare the data we want to store via _tables_. We can add arbitrary tables according to our needs, but we need to specify the structure ahead of time by passing the schema. In our case, we will start out with just a simple metadata table containing the parameters used to run an instance of the program above, along with a table for writing the result:

```
store.add_table('metadata', {
  'step_size': float,
  'tolerance': float,
  'initial_x': float,
  'out_dir': str
})

store.add_table('result', {
    'final_x': float,
    'final_opt':float
})
```

Each table corresponds exactly to a Pandas dataframe found in an `HDFStore` object.

A note on serialization

Cox supports basic object types (like `float`, `int`, `str`, etc) along with any kind of serializable object (via `dill` or using PyTorch's serialization method). In particular, if we want to serialize an object we can pass one of the following types: `cox.store.[OBJECT|PICKLE|PYTORCH_STATE]` as the type value that is mapped to in the schema dictionary. `cox.store.PYTORCH_STATE` is particularly useful for dealing with PyTorch objects like model weights. In detail: *OBJECT* corresponds to storing the object as a serialized string in the table, *PICKLE* corresponds to storing the object as a serialized string on disk in a separate file, and *PYTORCH_STATE* corresponds to storing the object as a serialized string on disk using `torch.save`. Note that saving large objects using *OBJECT* is not recommended as it will adversely affect loading times.

Logging

Now that we have a table, we can write rows to it! Logging in Cox is done in a row-by-row manner: at any time, there is a *working row* that can be appended to/updated; the row can then be flushed (i.e. written to the file), which starts a new (empty) working row. The relevant commands are:

```
# This updates the working row, but does not write it permenantly yet!
store['result'].update_row({
  "final_x": 3.0
})

# This updates it again
store['result'].update_row({
  "final_opt": 3.9409
})

# Write the row permenantly, and start a new working row!
store['result'].flush_row()

# A shortcut for appending a row directly
store['metadata'].append_row({
  'step_size': 0.01,
  'tolerance': 1e-6,
  'initial_x': 1.0,
  'out_dir': '/tmp/'
})
```

Incremental updates with *update_row*

Subsequent calls to `update_row()` will edit the same working row. This is useful if different parts of the row are computed in different functions/locations in the code, as it removes the need for passing statistics around all over the place.

Reading data

By populating tables rows, we are really just adding rows to an underlying `HDFStore` table. If we want to read the store later, we can simply open another store at the same location, and then read dataframes with simple commands:

```
# Note that EXP_ID is the directory the store wrote to in OUT_DIR
s = Store(OUT_DIR, EXP_ID)

# Read tables we wrote earlier
metadata = s['metadata'].df
result = s['result'].df

print(result)
```

Inspecting the `result` table, we see the expected result in our Pandas dataframe!:

```
    final_x   final_opt
0   3.000000   3.940900
```

CollectionReader: Reading many experiments at once

Now, in our quadratic example, we aren't just going to try one set of parameters, we are going to try a number of different values for `step_size`, `tolerance`, and `initial_x` (we haven't yet discovered convex optimization). To do this, we just run the code above a bunch of times with the desired hyperparameters, supplying the *same* `OUT_DIR` for all of the runs (recall that `cox` will automatically create different, `uuid`-named folders inside `OUT_DIR` for each experiment).

Imagine that we have done so (using any standard tool, e.g. *sbatch* in SLURM, *sklearn* grid search, or even a for loop like in our example file), and that we have a directory full of stores:

```
$ ls $OUT_DIR
drwxr-xr-x  6 engstrom  0424807a-c9c0-4974-b881-f927fc5ae7c3
...
...
drwxr-xr-x  6 engstrom  e3646fcf-569b-46fc-aba5-1e9734fedbcf
drwxr-xr-x  6 engstrom  f23d6da4-e3f9-48af-aa49-82f5c017e14f
```

Now, we want to collect all the results from this directory. We can use `cox.readers.CollectionReader` to read all the tables together in a concatenated `pandas` table.:

```
from cox.readers import CollectionReader
reader = CollectionReader(OUT_DIR)
print(reader.df('result'))
```

Which gives us all the `result` tables concatenated together as a Pandas DataFrame for easy manipulation:

```
    final_x   final_opt                                exp_id
0   1.000000    4.060900  ed892c4f-069f-4a6d-9775-be8fdfce4713
0   0.000010    7.120859  44ea3334-d2b4-47fe-830c-2d13dc0e7aaa
...
...
0   2.000000    3.000900  f031fc42-8788-4876-8c96-2c1237ceb63d
0 -14.000000  259.960900  73181d27-2928-48ec-9ac6-744837616c4b
```

`pandas` has a ton of powerful utilities for searching through and manipulating DataFrames. We recommend looking at their docs for information on how to do this. For convenience, we've given a few simple examples below:

```
df = reader.df('result')
m_df = reader.df('metadata')

# Filter by experiments have step_size less than 1.0
exp_ids = set(m_df[m_df['step_size'] < 1.0]['exp_id].tolist())
print(df[df['exp_id'].isin(exp_ids)]) # The filtered DataFrame

# Finding which experiment has the lowest final_opt
exp_id = df[df['final_opt'] == min(df['final_opt'].tolist())]['exp_id'].tolist()[0]
print(m_df[m_df['exp_id'] == exp_id]) # Metadata of the best experiment
```

1.1.2 Walkthrough 2: Using *cox* with *tensorboardX*

Note: As with the first walkthrough**, a working example file with all of these commands can be found** here

Here, we'll show how to use `cox` and `tensorboardX` in unison for logging. We'll use the following simple running example:

```
from cox.store import Store

for slope in range(5):
    s = Store(OUT_DIR) # Create OUT_DIR/RANDOM_UUID
    s.add_table('line_graphs', {'mx': int, 'mx^2': int})
    s.add_table('metadata', {'slope': int})
    s['metadata'].append_row({'slope': slope})

 # GOAL: plot and log the lines "y=slope*x" and "y=slope*x^2"
```

As previously mentioned, *`cox.store.Store`* objects also automatically creates a `tensorboard` folder that is written to via the tensorboardX library. A created *`cox.store.Store`* object will actually expose a `writer` property that is a fully functioning SummaryWriter object. That means we can plot the lines we want in TensorBoard as follows:

```
for x in range(10):
    s.writer.add_scalar('line', slope*x, x)
    s.writer.add_scalar('parabola', slope*(x**2), x)
```

Unfortunately, TensorBoard data is quite hard to read/manipulate through means other than the TensorBoard interface. For convenience, the *`Store`* object also provides the ability to write to a table and the `tensorboardX` writer at the same time through the *`cox.store.Store.log_table_and_tb()`* function, meaning that we can replace the above with:

```
# Does the same thing as the example above but also stores the results in a
# readable 'line_graphs' table
for x in range(10):
    s.log_table_and_tb('line_graphs', {'mx': slope*x, 'mx^2': slope*(x**2)})
    s['line_graphs'].flush_row()
```

Viewing multiple tensorboards with *cox.tensorboard_view*

Note: the `python3 -m cox.tensorboard_view` **command can be called as** `cox-tensorboard` **from the command line

Continuing with our running example, we may now want to visually compare TensorBoards across multiple parameter settings. Fortunately, `cox` provides utilities for comparing TensorBoards across experiments in a readable way. In our example, where we made a *`Store`* object and a table called `metadata` where we stored hyperparameters. We also showed how to integrate TensorBoard logging via `tensorboardX`. We'll now use the `cox.tensorboard_view` utility to view the tensorboards from multiple jobs at once (this is useful when comparing parameters for a grid search).

The way to achieve this is through the `cox.tensorboard_view` command, which is called as `python3 -m cox.tensorboard_view` with the following arguments:

`--logdir`: (required) the directory where all of the stores are located

`--port` (default 6006) the port on which to run the tensorboard server

`--metadata-table` (default "metadata") the name of the table where the hyperparameters are saved (i.e. "metadata" in our running example). This should be a table with a single row, as in our running example.

`--filter-param` (optional) Can be used more than once, filters out stores from the tensorboard aggregation. For each argument of the form `--filter-param PARAM_NAME PARAM_REGEX`, only the stores where `PARAM_NAME` in the metadata matches `PARAM_REGEX` will be kept.

`--format-str` (required) How to display the name of the stores. Recall that each store has a `uuid`-generated name by default. This argument determines how their names will be displayed in the TensorBoard. Curly braces represent parameter values, and the uuid will always be appended to the name. So in our running example, `--format-str ss-{step_size}` will result in a TensorBoard with names of the form `ss-1.0-ed892c4f-069f-4a6d-9775-be8fdfce4713`.

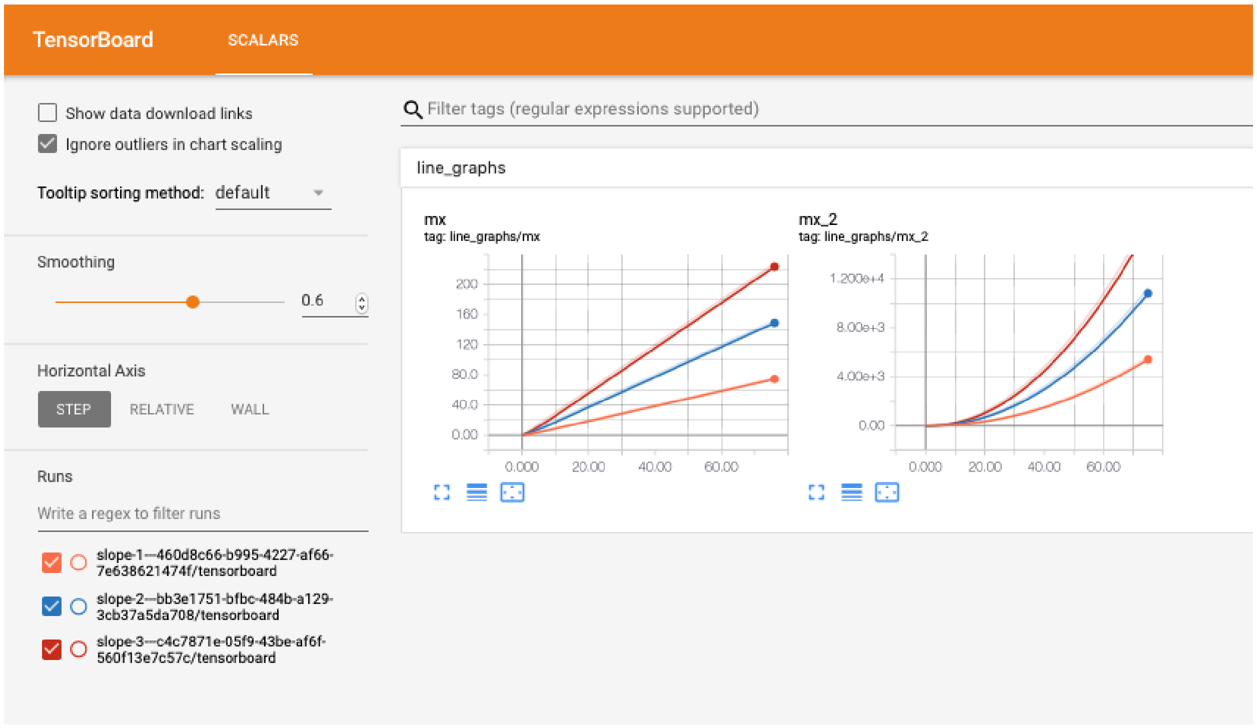
So in our running example, if we run the following command, displaying the slope in the TensorBoard names and filtering for slopes between 1 and 3:

```
python3 -m cox.tensorboard_view --logdir OUT_DIR --format-str slope-{slope} \
    --filter-param slope [1-3] --metadata-table metadata
```

or:

```
cox-tensorboard --logdir OUT_DIR --format-str slope-{slope} \
    --filter-param slope [1-3] --metadata-table metadata
```

then navigating to *localhost:6006* yields:



Tooltip sorting method: default

Smoothing 0.6

slope-1---460d8c66-b995-4227-af66-7e638621474f/tensorboard

slope-2---bb3e1751-bfbc-484b-a129-3cb37a5da708/tensorboard

slope-3---c4c7871e-05f9-43be-af6f-560f13e7c57c/tensorboard

1.2 Submodules

1.2.1 cox.readers module

class `cox.readers.`**`CollectionReader`**(*directory*, *log_warnings=True*, *mode='r'*, *exp_filter=None*, *skip_errs=False*)

Bases: `object`

Class for collecting, viewing, and manipulating directories of stores.

Initialize the CollectionReader object. This will immediately open each store in *directory* and see which table are available for viewing.

Parameters

- **directory** (*str*) – Path to directory with stores in it. The directory should contain directories corresponding to stores.
- **log_warnings** (*bool*) – Log warnings if tables with the same name have different schemas
- **mode** (*str*) – mode to open stores in. Default 'r' (read only), if you want to write you will need to make the mode 'a' (append only) or 'w' (write).
- **exp_filter** (*method*) – Call exp_filter on the experiment id of each store, excludes store from collection if it returns *false*.

close()

Closes all the stores opened by the collection reader.

df(*key*, *append_exp_id=True*, *keep_serialized=[]*, *union_schemas=False*, *exp_filter=None*, *skip_errors=False*)

Makes a large concatenated PD dataframe from all the stores' tables matching this table key.

Parameters

- **key** (*str*) – name of table to collect
- **append_exp_id** (*bool*) – if true, append corresponding experiment id to each row.
- **keep_serialized** (*list of strings*) – list corresponding to column names. If in this list, do not unserialize the string within the column name and make it a python object within the pandas table.
- **union_schemas** (*bool*) – If true, union columns of all collected tables, otherwise error out.
- **exp_filter** (*method*) – If function of exp_id returns false, ignore this store. Otherwise include.
- **skip_errors** (*bool*) – If true, skip an experiment upon error occurs.

Returns Concatenated dataframe of all corresponding tables in the dataframes matching the key.

1.2.2 cox.store module

class `cox.store.`**Store**(*storage_folder*, *exp_id=None*, *new=False*, *mode='a'*)

Bases: `object`

Serializes and saves data from experiment runs. Automatically makes a tensorboard. Access the tensorboard field, and refer to the TensorboardX documentation for more information about how to manipulate it (it is a tensorboardX object).

Directly saves: int, float, torch scalar, string Saves and links: np.array, torch tensor, python object (via pickle or pytorch serialization)

Note on python object serialization: you can choose one of three options to serialize using: *OBJECT* (store as python serialization inline), *PICKLE* (store as python serialization on disk), or *PYTORCH_STATE* (save as pytorch serialization on disk). All these types are represented as properties, i.e. `store_instance.PYTORCH_STATE`. You will need to manually decode the objects using the static methods found in the *Table* class (`get_pytorch_state`, `get_object`, `get_pickle`), or use a *cox.readers.CollectionReader* which will handle this for you.

Make new experiment store in `storage_folder`, within its subdirectory `exp_id` (if not none). If an experiment exists already with this corresponding directory, open it for reading.

Parameters

- **storage_folder** (*str*) – parent folder in which we will put a folder with all our experiment data (this store).
- **exp_id** (*str*) – dir name in `storage_folder` under which we will store experimental data.
- **new** (*str*) – enforce that this store has never been created before.
- **mode** (*str*) – mode for accessing tables. a is append only, r is read only, w is write.

OBJECT = '__object__'
Python serialized datatype (saved as string in the h5 table—not recommended for large objects as these objects must be loaded along with the table)

PICKLE = '__pickle__'
Pickle datatype (saved on disk and referenced from the table—recommended for larger objects)

PYTORCH_STATE = '__pytorch_state__'
PyTorch state, e.g. from model.state_dict() (saved on disk and linked)

add_table(*table_name*, *schema*)
Add a new table to the experiment.

Parameters

- **table_name** (*str*) – a name for the table
- **schema** (*dict*) – a dict for the schema of the table. The entries should be of the form name:type. For example, if we wanted to add a float column in the table named acc, we would have an entry `'acc':float`.

Returns The table object of the new table.

add_table_like_example(*table_name*, *example*, *alternative='__object__'*)
Add a new table to the experiment, using an example dictionary as the basis for the types of the columns.

Parameters

- **table_name** (*str*) – a name for the table
- **example** (*dict*) – example for the schema of the table. Make a table with columns with types corresponding to the types of the objects in the dictionary.
- **alternative** (*self.OBJECT|self.PICKLE|self.PYTORCH_STATE*) – how to store columns that are python objects.

close()
Closes underlying HDFStore of this store.

get_table(*table_id*)
Gets table with key `table_id`.

Parameters **table_id** (*str*) – id of table to get from this store.

Returns The corresponding table (Table object).

log_table_and_tb(*table_name*, *update_dict*, *summary_type='scalar'*)
Log to a table and also a tensorboard.

Parameters

- **table_name** (*str*) – which table to log to

- **`update_dict`** (*`dict`*) – values to log and store as a dictionary of column mapping to value.
- **`summary_type`** (*`str`*) – what type of summary to log to tensorboard as

class `cox.store.`**`Table`**(*name*, *schema*, *table_obj_dir*, *store*, *has_initialized=False*)

Bases: `object`

A class representing a single storer table, to be written to by the experiment. This is essentially a single HDFStore table.

Create a new Table object.

Parameters

- **`name`** (*`str`*) – name of table
- **`schema`** (*`dict`*) – schema of table (as described in *`cox.store.Store`* class)
- **`table_obj_dir`** (*`str`*) – where to store serialized objects on disk store (Store) : parent store.
- **`has_initialized`** (*`bool`*) – has this table been created yet.

`append_row`(*data*)

Write a dictionary with format column name:value as a row to the table. Must have a value for each column. See `update_row()` for more mechanics.

Parameters **`data`** (*`dict`*) – dictionary with format `column name`:`value`.

`df`

Access the underlying pandas dataframe for this table.

`flush_row`()

Writes the current row we have staged (using *`update_row()`*) to the table. Another row is immediately staged for *`update_row()`* to act on.

`get_object`(*s*)

Unserialize object of store.OBJECT type (a pickled object stored as a string in the table).

Parameters **`s`** (*`str`*) – pickle string to unpickle into a python object.

`get_pickle`(*uid*)

Unserialize object of store.PICKLE type (a pickled object stored as a string on disk).

Parameters **`uid`** (*`str`*) – identifier corresponding to stored object in the table.

`get_state_dict`(*uid*, ***kwargs*)

Unserialize object of store.PYTORCH_STATE type (object stored using pytorch's serialization system).

Parameters **`uid`** (*`str`*) – identifier corresponding to stored object in the table.

`nrows`

How many rows this table has.

`schema`

Access the underlying schema for this table.

`update_row`(*data*)

Update the currently considered row in the data store. Our database is append only using the *`cox.store.Table`* API. We can update this single row as much as we desire, using column:value mappings in `data`. Eventually, the currently considered row must be written to the database using *`cox.store.Table.flush_row()`*. This model allows for writing rows easily when not all the values are known in a single context. Each `data` object does not need to contain every column, but by the time that the row is flushed every column must obtained a value. This update model is stateful.

Python primitives (`int`, `float`, `str`, `bool`), and their numpy equivalents are written automatically to the row. All other objects are serialized (see *`Store`*).

Parameters **`data`** (*`dict`*) – a dictionary with format `column name`:`value`.

`cox.store.`**`schema_from_dict`**(*d*, *alternative='__object__'*)

Given a dictionary mapping column names to values, make a corresponding schema.

Parameters

- **`d`** (*`dict`*) – dict of values we are going to infer the schema from
- **`alternative`** (*`self.OBJECT|self.PICKLE|self.PYTORCH_STATE`*) – how to store columns that are python objects.

1.2.3 cox.tensorboard_view module

`cox.tensorboard_view.`**`main`**()

This function is meant to be run via command line (see Walkthrough 2 for more information.

1.2.4 cox.utils module

`class` `cox.utils.`**`Parameters`**(*params*)

Bases: `object`

Parameters class, just a nice way of accessing a dictionary

```
ps = Parameters({"a": 1, "b": 3})
ps.A # returns 1
```

`as_dict`()

`cox.utils.`**`consistent`**(*old*, *new*)

Asserts that either first argument is None or both arguments are equal, and returns the non-None argument.

`cox.utils.`**`has_tensorboard`**(*dirname*)

Given a directory path, return whether or not it has a tensorboard directory in it.

Parameters **`dirname`** (*`str`*) – path to directory

Returns Whether or not the directory has a "tensorboard" folder in it.

`cox.utils.`**`mkdirp`**(*x*, *should_msg=False*)

Tries to make a directory, but doesn't error if the directory exists/can't be created.

`cox.utils.`**`obj_to_string`**(*obj*)

Serialize an object to a string

`cox.utils.`**`override_json`**(*args*, *json_path*, *check_consistency=False*)

Overrides the null values in an arguments object with values extracted from a JSON file.

Parameters

- **`args`** (*`object`*) – A python object with the arguments as properties.
- **`json_path`** (*`str`*) – Path to the JSON file with which to override.
- **`check_consistency`** (*`bool`*) – If true, make sure that the keys in the JSON file and the args object match up exactly

Returns A new args object with appropriately overriden None values.

`cox.utils.`**`string_to_obj`**`(`*s*`)`

Unserialize a string back into an object.

Python Module Index

Index