Testing Geovision (Draft)

The process of conducting deep learning experiments on satellite data typically follows a standard workflow. This package is designed to be highly modular, enabling new components to be seamlessly integrated into the ecosystem. Implementing unit tests for each module, along with integration tests for the entire system, is crucial for research. These tests allow researchers to concentrate on their work without being distracted by the need to hunt down systematic or, worse, logical errors within the codebase.

Routine Steps

1. **Extract** data from original source(s)
   - Only when the dataset is hosted on a 'permanent' dissemination server and the dataset needs to be continually updated with new data does it make sense to write tests for the download scripts.
   - In almost all other cases, this step is performed only once, thus a clear and well documented download script should suffice.

2. **Transform** the data into a structured dataset
   - The data is arranged into hierarchical filesystem (Imagefolder).
   - The data is encoded into a chunked file format, optionally compressed (Archive, HDF5, LitData).

3. **(Up)Load** the dataset to a (cloud) warehouse
   - The dataset is uploaded to an object storage server, where successfull uploads and access paths can be tested, along with uploaded dataset size. Also performed very infrequently.

4. **(Down)Load** the dataset to a local machine for train / eval
   - Test directory structure is as expected, with files at expected paths
   - Test against checksum, expected size, etc.

5. **The Dataset Class** maps integer indices to samples

   - Test schema for :df and :split_df
     - Tabular Sampling: test if the entire dataset is subset, without overlaps, display num samples per class
     - Spatial Sampling: test if the crs and affine transformations are within bounds
     - Spectral Sampling: test if the bands tuple is within the expected range
     - Temporal Sampling:
     - Dtypes: index(int, unique), dataset_idx(int, unique, foreign key), image_path(str, valid_file), mask_path(str, valid_file), label_idx(int, isin ...), split(str, isin Dataset.valid_splits), band_1(int), band_3(int), band_3(int), ..., crs(int), x_off(float), y_off(float), x_range(float), y_range(float). These vary per dataset and should be implemented per dataset.

   - Test init with an invalid root throws an Error

   - Test init with an invalid split throws an Error

   - Test init without :split sets split to "all"

   - Test init with wrong dataframes throws an Error

   - Test init with different splits sets a unique split_df (no overlap), and the total sums up to "all" split

   - Test `__len__(self)` output matches len(Dataset.split_df)

   - Test `__getitem__(self, int)` by init in the "all" split with T.Identity() transformations and loading each sample by calling ds[idx] over the entire range to test output is a 3-tuple function, usually a `tuple[Tensor, int, int]` for classification and `tuple[Tensor, Tensor, int]` for segmentation, with the expected shapes for images and targets.

6. **The DataLoader Class** adds shuffling, batching and parallelizing to the dataloading process

- Test dataloading by passing default init dataset to a shuffling DataLoader with a small batch size (4 should work for most datasets) and 4 worker processes
  - Test the collate_fn works and batches are loaded with the expected shape, dtypes
  - Test that each sample is loaded only once, and the entire dataset is passed (unique and exhaustive)
  - Test for memory leaks?

7. **The Datamodule Class** organizes datasets and dataloaders in a single class
   - Add a dm.test() function to run one epoch on the GPU with each split and (optionally) display plots of each batch, and save them to a temporary directory. Also display the shape and dtype of the batches, ensure uniqueness and exhaustive loading, along with time taken, memory usage and image statistics. (add pytorch profiler?)