Robust-Raster Documentation

What is Robust-Raster?

Robust-Raster is a Python package that allows users to run their own functions on large geospatial datasets without having to write their own code for parallelization. The idea here is to make large-scale geospatial data analysis more accessible, lowering the barrier of entry for users not familiar with advanced computing techniques.

Why Robust-Raster?

In recent years, the amount of data collected from satellites has grown dramatically. This data can help us understand our planet better, but its sheer size makes it difficult to analyze. Traditional methods struggle to keep up, leading to a need for advanced technological solutions. Google Earth Engine (GEE) is one formidable solution as it provides access to vast amounts of satellite data and various analysis tools without the need for a computer powerful enough to store and process all that data. However, GEE has limitations in the types of analyses it can perform on data. Other data-intensive computing solutions, such as xarray–an extension of Python's NumPy arrays–and Dask, a Python library for parallel and distributed computing, have expanded the types of analyses performed on large datasets. Yet, these tools can be difficult to use for those unfamiliar with parallelization and multidimensional arrays. To address these limitations, I developed "Robust-Raster", a Python software package that allows users to run custom analyses on large data using their own computers. This tool is designed to be user-friendly for scientists who might not be familiar with advanced computing techniques.

How does Robust-Raster work?

Robust-Raster utilizes Dask as well as xarray to handle managing the user's dataset as well as their function. The package will take the user's function and dataset, automatically determine the optimal Dask configuration based on the user's computational resources and function complexity, and apply the function on the dataset. I call this process "tuning" through the scope of this documentation. The user does not need to fully understand how the code (or more specifically how Dask) works as it is performed behind the scenes (although an understanding of Dask is encouraged).

What is Dask?

Imagine you're working with a massive amount of data, like satellite imagery. This data might be too large to fit into the memory of your computer, or it might take forever to process.

Dask is a tool that helps you process this large data more efficiently by breaking it down into smaller chunks and working on them in parallel. This means Dask can distribute the work across multiple processors or even multiple computers, speeding up the whole process.

Think of it like a team of people working on a big project. If one person is doing everything, it might take a long time. But if you divide the work and assign different parts of the project to different team members, the whole project gets completed much quicker. Each person (or in this case, each computer or each processor depending on your environment) works on their part, and the results are combined to get the job done.

More information on Dask can be found here.

Try Out the Demo!

I highly recommend you try out the demo in the GitHub repo first before continuing to read! You don't need a full understanding of the code just yet. However, briefly familiarizing yourself will make the documentation easier to understand.

Breakdown of Robust-Raster's classes and functions

There are three primary sections or steps the user needs to understand when using Robust-Raster. With each step, I describe the related classes that you will need to use to get your code up and running.

1. Setting up the Dask cluster
    a. DaskClusterManager

2. Setting up the dataset
    a. EarthEngineDataset
    b. RasterDataset
3. Tuning/applying the user-defined function (note that tuning is optional)
    a. UserDefinedFunction

Setting Up the Dask Cluster

DaskClusterManager

`DaskClusterManager` is a class for handling Dask client operations and configurations.

Now this class is meant to keep Dask operations for the user as simple as possible. The idea here is to make large-scale data analysis accessible. The only thing I'm asking of you as the user is to boot up Dask using this class. The class will handle all of the bells and whistles (the backend code) with spinning up Dask.

I will now go over the two methods associated with the class:

- `get_dask_client`: A property that returns the current Dask Client instance, allowing the user to monitor or interact with it. If you choose to read up on Dask, having the Dask client handy could be useful. But, if you are just concerned with processing your data, you can completely disregard this method.
- `create_cluster`: A method that creates a Dask cluster of workers needed to parallelize tasks. THIS IS THE ONLY METHOD YOU NEED TO BE CONCERNED WITH IF YOU WANT TO JUST PROCESS YOUR DATA!!!

Let's break down how to use `create_cluster`.

To use `create_cluster`, you can run the following code:

```
>>> from robustraster import dask_cluster_manager

>>> dask_cluster = dask_cluster_manager.DaskClusterManager()

>>> dask_cluster.create_cluster(mode="full")
```

This code will:

1. Import the `DaskClusterManager` class
2. Create a `DaskClusterManager` object.
3. Using this new `DaskClusterManager` object, create a full Dask cluster.

Side note: what is a cluster? And what is a "full" cluster?

A cluster, in simple terms, is a group of computers, servers, or processors that work together to handle a big task. Instead of relying on just one computer to do everything, a cluster spreads the work across multiple computers (or multiple threads/processors if you are running this on one computer), so the task gets done faster and more efficiently.

Think of the team of people example I mentioned in the "What is Dask?" section of this documentation. Each person in this example would be a "worker". A group of "workers" make up a "cluster". "Workers" and "clusters" are commonly used terms in the world of computer science parallelization. I will refer to computers, servers, and processors as "workers" and a group of these as a "cluster" from now on.

What does a "full" cluster mean? Well, this method accepts two modes: "full" and "test".

"full" refers to creating a cluster of one Dask worker per CPU core on the user's machine. Memory will be split evenly between each worker. For example, if you have a machine with 16 CPU cores and 32GB of memory, "full" will create a cluster of 16 workers with each worker having 2GB of memory.

"test" creates a Dask cluster consisting of ONLY one worker. Only use this if you plan on using the `tune_user_function` method in the udf_tuner class. For accurate results, `tune_user_function` requires a Dask cluster configured with the "test" parameter and not "full". Refer back to `demo.ipynb` to see this in use. More information on function tuning can be found in the udf_tuner class.

To create a Dask cluster in "full" mode:

```
>>> from robustraster import dask_cluster_manager

>>> dask_cluster = dask_cluster_manager.DaskClusterManager()

>>> dask_cluster.create_cluster(mode="full")
```

To create a Dask cluster in "test" mode:

```
>>> from robustraster import dask_cluster_manager

>>> dask_cluster = dask_cluster_manager.DaskClusterManager()

>>> dask_cluster.create_cluster(mode="test”)
```

If you are a bit more tech-savvy and want to create a custom Dask cluster, you can pass in your own parameters like so:

```
>>> from robustraster import dask_cluster_manager

>>> dask_cluster = dask_cluster_manager.DaskClusterManager()
>>> dask_cluster.create_cluster(mode="full", n_workers=2,
threads_per_worker=1, memory_limit="2GB")
```

You still need to set the mode to “full”. Feel free to play around with the configuration settings, but I would personally change `threads_per_worker` at your own risk! Keep it at 1 or don’t touch it at all! But you do you.

So you got your Dask cluster running. Great! Now we need to load your dataset!

Loading the Dataset

As of now, there are two different ways to load in your dataset:

1. Via Google Earth Engine
2. Via raster(s) stored locally

I will go over each.

EarthEngineDataset

`EarthEngineDataset` is a class for pulling data from Google Earth Engine into an object on your Dask cluster. This class was made as an extension of xee, a Python package that integrates xarray with Google Earth Engine. The xarray data structure has proven to be a well-optimized data structure for handling and scaling large continuous geospatial datasets, which was ultimately why xee was created. However, for those not familiar with multidimensional array data structures like xarray, I decided to design EarthEngineDataset to make working with xarray more accessible.

In order to properly use this class, you will need a Google Cloud project authenticated with Google Earth Engine. More information can be found here.

To instantiate an EarthEngineDataset object, the user must pass in a dictionary object of parameters. Below is an example `parameters` variable.

```
>>> parameters = {

>>>     'collection': 'LANDSAT/LC08/C02/T1_L2',

>>>     'bands': ['SR_B4', 'SR_B5'],

>>>     'start_date': '2020-05-01',

>>>     'end_date': '2020-08-31',

>>>     'geometry': WSDemo.geometry(),

>>>     'crs': 'EPSG:3310',
>>>     'scale': 30,

>>>     'map_function': prep_sr_l8

>>> }
```

Where:

- `collection` is the Earth Engine path to the image collection of interest.
- `bands` are the bands the user would like to export from Earth Engine. This parameter is optional.
- `start_date` / `end_date` is the date range to filter the image collection. These parameters are optional.
- `geometry` is the geometry object that will be used to clip the image collection. This parameter is optional.
- `crs` is the coordinate system to project the image collection to. This parameter is optional.
- `scale` is the spatial resolution the user would like the image collection to be. This parameter is optional.

For more information on these parameters, see the documentation for Earth Engine's export functions)

- `map_function` is the name of the Earth Engine function the user would like to run on an image collection before exporting the data. See the example usage below to see how this is used.

Example usage for integrating Earth Engine with a custom cloud masking algorithm:

```
1. Import required libraries and modules:

>>> from robustraster import input_driver

>>> import ee

>>> import json

2. Authenticate and initialize Earth Engine:

>>> with open(json_key, 'r') as file:

>>>     data = json.load(file)

>>> credentials =
ee.ServiceAccountCredentials(data["client_email"], json_key)

>>> ee.Initialize(credentials=credentials,
opt_url='https://earthengine-highvolume.googleapis.com')
```

3. Define a cloud masking algorithm for Landsat 8 Surface Reflectance:

```
>>> def prep_sr_l8(image):

>>>     # Bit 0 - Fill

>>>     # Bit 1 - Dilated Cloud

>>>     # Bit 2 - Cirrus

>>>     # Bit 3 - Cloud

>>>     # Bit 4 - Cloud Shadow

>>>     qa_mask =
image.select('QA_PIXEL').bitwiseAnd(int('11111', 2)).eq(0)

>>>     saturation_mask = image.select('QA_RADSAT').eq(0)

>>>     # Apply scaling factors to appropriate bands

>>>     optical_bands =
image.select('SR_B.*').multiply(0.0000275).add(-0.2)

>>>     thermal_bands =
image.select('ST_B.*').multiply(0.00341802).add(149.0)

>>>     # Return the processed image

>>>     return (image.addBands(optical_bands, None, True)

>>>                 .addBands(thermal_bands, None, True)

>>>                 .updateMask(qa_mask)

>>>                 .updateMask(saturation_mask))
```

4. Prepare parameters for data processing:

```
>>> WSDemo =
ee.FeatureCollection("projects/robust-raster/assets/boundaries/W
SDemoSHP_Albers")

>>> test_parameters = {

>>>     'collection': 'LANDSAT/LC08/C02/T1_L2',

>>>     'bands': ['SR_B4', 'SR_B5'],
>>>     'start_date': '2020-05-01',

>>>     'end_date': '2020-08-31',

>>>     'geometry': WSDemo.geometry(),

>>>     'crs': 'EPSG:3310',

>>>     'scale': 30,

>>>     'map_function': prep_sr_l8

>>> }
```

5. Create the EarthEngineDataset object:

```
 >>> earth_engine =
input_driver.EarthEngineDataset(parameters=test_parameters)
```

RasterDataset

`RasterDataset` is a class for pulling data from a raster saved on your computer into an object on your Dask cluster.

To instantiate an object of type `RasterDataset`:

```
>>> reader = RasterDataset("/path/to/raster.tif")
```

If you want to load multiple rasters, pass in a list of raster file paths:

```
from robustraster import input_driver

raster_path_list = ['./ndvi.tif', './ndvi_local.tif']

local_raster = input_driver.RasterDataset(raster_path_list)
```

Tuning/Applying the User-Defined Function

UserDefinedFunction

`UserDefinedFunction` is a class meant to allow users to apply their own custom functions on a dataset derived from using this package (i.e. from an `EarthEngineDataset` or a `RasterDataset` object). This class can also tune the user function, identifying the user's available computational resources to determine an optimal way to parallelize the user's function, though this is optional.

There are two methods associated with this class. These methods are the most important part of the package. Because of this, I am dedicating two long subsections to each of these methods.

tune_user_function

Taking in the user's dataset object and their custom function, tune the function to fit within the constraints of the user's computational infrastructure.

How exactly does this tuning work?

The user's dataset is constructed using xarray, a data structure that is an extension to Python NumPy arrays. A major advantage in using xarray is its ability to parallelize large datasets. This is done through what is called "chunking". Documentation for chunking can be found here.

Chunking in xarray refers to breaking down a dataset into smaller, more manageable pieces called "chunks," which are stored and processed independently. These chunks allow xarray to handle datasets that are too large to fit into memory all at once. So rather than running a user's function on the entire dataset, a set number of chunks are loaded into memory and the function is run on each chunk in parallel. Computed chunks are then freed from memory (and potentially written to disk if the user chooses so) and more uncomputed chunks are loaded. A common question associated with chunking is what is the best chunk size for a dataset? More information on this subject can be found here.

Determining an optimal chunk size can be a challenge due to many external factors such as CPU/RAM constraints and user function complexity. This is what this method tries to accomplish. `tune_user_function` does the following:

1. The user passes in two parameters: the dataset object and their function name.
2. The user's function is then run on a single chunk of the dataset. The initial size of the chunk is the smallest it can possibly be.
3. Write the performance metrics of this run to a file.
4. Increase the chunk size by a factor of 2.
5. Repeat steps 1-4 on the newly created chunk until an optimal chunk size is found.
6. Writes the resulting optimal chunk size to a JSON file that can be passed into `apply_user_function` (although if the UserDefinedFunction object that tuned the user's function is still instantiated, this is not required). This is useful if you want to run `apply_user_function` at a later time and don't want to tune your function again. The JSON will have your progress from the last tuning session saved.

With each iteration, checks are set in place to compare the compute time of the prior iteration to the newest iteration. If the compute time of the most recent iteration is bigger than the compute time of the last iteration, then the last iteration's chunk size is returned. There are a lot more checks than what is mentioned here; I am just summarizing the main idea of `tune_user_function`. Feel free to look at the source code for a full breakdown of the optimization process.

Benefits of chunking include:

- Scalability: Enables working with datasets larger than your computer's memory.
- Parallelism: Allows operations to run across multiple CPU cores or even distributed systems.
- Lazy Evaluation: Operations are deferred until explicitly computed, reducing unnecessary computations.
- Optimized I/O: Only the chunks needed for a computation are read from disk, minimizing disk access.

Parameters:

- `data_source` (`RasterDataset` or `EarthEngineDataset`): The user's dataset object.
- `user_func` (`Callable[[],pd.DataFrame]`): The user's function name. For now, user functions need to return a pandas DataFrame. See the example below for an example function.
- `max_iterations (int)`: An optional argument sets the maximum number of iterations.
- `*args`: Positional arguments that will be passed into their custom function.
- `**kwargs`: Keyword arguments that will be passed into their custom function.

Here is an example of how to instantiate a `UserDefinedFunction` object, running `tune_user_function`, and finally running `apply_user_function`:

In this example, I wrote my own custom function that computes the NDVI on a pandas DataFrame object. I then instantiate an object of type `UserDefinedFunction`. Finally, I call the `tune_user_function` method, passing in my `EarthEngineDataset` object as well as my function as inputs.

```
>>> def compute_ndvi(df):

>>>     # Perform your calculations

>>>     df['ndvi'] = (df['SR_B5'] - df['SR_B4']) / (df['SR_B5']
+ df['SR_B4'])

>>>     return df



>>> from robustraster import udf_tuner
>>> from robustraster import input_driver

>>> earth_engine =
input_driver.EarthEngineDataset(parameters=test_parameters)

>>> user_defined_func = udf_tuner.UserDefinedFunction()

>>>
user_defined_func.tune_user_function(data_source=earth_engine,
user_func=compute_ndvi)
```

At this point, my function has been "tuned". We can do a full run of the function with the following code:

```
>>> full_result =
user_defined_func.apply_user_function(data_source=earth_engine,
user_func=compute_ndvi)
```

Running `tune_user_function` has an optional parameter called `max_iterations`.

Here is an example of running `tune_user_function` and the `max_iterations` parameter.

```
>>> def compute_ndvi(df):

>>>     # Perform your calculations

>>>     df['ndvi'] = (df['SR_B5'] - df['SR_B4']) / (df['SR_B5']
+ df['SR_B4'])

>>>     return df



>>> from robustraster import udf_tuner

>>> user_defined_func = udf_tuner.UserDefinedFunction()

>>>
user_defined_func.tune_user_function(data_source=earth_engine,
user_func=compute_ndvi, max_iterations=10)
```

This will perform steps 1-4 ten times.

If your function requires multiple parameters to be passed:

```
# In this example, I added a new positional argument, "numba".

>>> def compute_ndvi(df, numba):

>>>     # Perform your calculations

>>>     df['ndvi'] = (df['SR_B5'] - df['SR_B4'] + numba) /
(df['SR_B5'] + df['SR_B4'])

>>>     return df
```

You can pass in additional positional or keyword arguments like the following:

Example passing in an additional positional argument:

```
>>> from robustraster import udf_tuner

>>> user_defined_func = udf_tuner.UserDefinedFunction()

>>>
user_defined_func.tune_user_function(data_source=earth_engine,
user_func=compute_ndvi, max_iterations=10, 666)
```

666 in this example will get passed into `compute_ndvi` as `numba`.

Example passing in an additional keyword argument:

```
>>> from robustraster import udf_tuner

>>> user_defined_func = udf_tuner.UserDefinedFunction()

>>>
user_defined_func.tune_user_function(data_source=earth_engine,
user_func=compute_ndvi, max_iterations=10, numba=666)
```

apply_user_function

Apply the user's custom function on the dataset. If the user ran `tune_user_function` prior, the user can use the same `UserDefinedFunction` object to run `apply_user_function` on their data. If done this way, this will run the tuned function on the dataset.

Parameters:

- `data_source` (`RasterDataset` or `EarthEngineDataset`): The user's dataset object.
- `user_func` (`Callable[[],pd.DataFrame]`): The user's function name. For now, user functions need to return pandas DataFrames. See the example below for an example function.
- `chunks` (`Optional[dict | str]`): An optional parameter that allows users to pass in their own custom chunk size on the dataset. For more information on chunks, refer to `tune_user_function`. There is an explanation for the benefits of chunking there. The user can pass in either a dictionary object containing the chunk parameters or a file path to the output JSON file generated after running `tune_user_function`. Otherwise, `apply_user_function` will auto-determine an appropriate chunk size for the dataset (this is not the same as tuning! It will choose an arbitrary "safe" chunk size).
- `*args`: Positional arguments that will be passed into the user's custom function.
- `**kwargs`: Keyword arguments that will be passed into the user's custom function.

Example 1: Running `tune_user_function` first, and then `apply_user_function` afterwards.

```
>>> def compute_ndvi(df):

>>>     # Perform your calculations

>>>     df['ndvi'] = (df['SR_B5'] - df['SR_B4']) / (df['SR_B5']
+ df['SR_B4'])

>>>     return df



>>> from robustraster import udf_tuner

>>> from robustraster import input_driver

>>> earth_engine = input_driver.EarthEngineDataset(parameters)

>>> user_defined_func = udf_tuner.UserDefinedFunction()

>>>
user_defined_func.tune_user_function(data_source=earth_engine,
user_func=compute_ndvi)

>>> full_result =
user_defined_func.apply_user_function(data_source=earth_engine,
user_func=compute_ndvi)
```

Example 2: Running `apply_user_function` without running `tune_user_function`.

```
>>> full_result =
user_defined_func.apply_user_function(earth_engine,
compute_ndvi)
```

Example 3: Running `apply_user_function` and passing in the JSON that `tune_user_function` generates. This could be useful if you boot up your code at a later time and don't want to run `tune_user_function` to tune your function again.

```
>>> full_result =
user_defined_func.apply_user_function(data_source=earth_engine,
user_func=compute_ndvi,
chunks="optimal_chunks_20250124_141211.json")
```

Example 4: Running `apply_user_function` and passing in a dictionary object of the chunk size. If the user wants the option to specify a custom chunk size without tuning (without running `tune_user_function`), they can do so here.

```
>>> my_custom_chunks = {'time': 48, 'X': 256, 'Y': 512}

>>> full_result =
user_defined_func.apply_user_function(data_source=earth_engine,
user_func=compute_ndvi, chunks=my_custom_chunks)
```

If your function requires multiple parameters to be passed:

```
# In this example, I added a new positional argument, `numba`.

>>> def compute_ndvi(df, numba):

>>>     # Perform your calculations

>>>     df['ndvi'] = (df['SR_B5'] - df['SR_B4'] + numba) /
(df['SR_B5'] + df['SR_B4'])

>>>     return df
```

You can pass in additional positional or keyword arguments like the following:

Example passing in an additional positional argument:

```
>>> from robustraster import udf_tuner

>>> user_defined_func = udf_tuner.UserDefinedFunction()

>>> user_defined_func.apply_user_function(earth_engine,
compute_ndvi, max_iteration=10, 666)
```

666 in this example will get passed into `compute_ndvi` as `numba`.

Example passing in an additional keyword argument:

```
>>> from robustraster import udf_tuner

>>> user_defined_func = udf_tuner.UserDefinedFunction()

>>> user_defined_func.tune_user_function(earth_engine,
compute_ndvi, max_iteration=10, numba=666)
```