Code

- All code can be found in the Github: https://github.com/manuelcgallucci/Super-Resolution_PROCOM

- The database with the LST and NDVI images used can be found in the following link: https://we.tl/t-rS5itDQPKk

Each py file and how it is used is explained in this section as well as how to train and verify the results from the model using the test and validation scripts. The scripts have been separated into the following 4 sections:

- **Database**: How to create and use the database of LST and NDVI images from MODIS.
- **Training**: How to train a model on the created database, observe the output results and validate using ASTER images.
- **Other**: Extra functions defined for specific purposes.
- **Classic Methods**: How to use classic methods to parse the LST image to compare with the model output.

Database

The database is constituted from two parts: land surface temperature images at 1km spatial resolution from the MODIS MOD11A1 instrument and Normalized difference vegetation indexes (NDVI) at 250m spatial resolution calculated using reflectance images from the MODIS MOD09GQ instrument. In total, there are 12034 temperature images coupled with 6017 NDVI images. In this document, we will present and explain the different code files that can be used to download the different images, process them, and save them at the end both in **tiff** format and **npz** format.

modis_download_process.py

This file executes the whole pipeline of downloading the database and saving the images in tiff format. The file takes the following two arguments to be executed :

year_begin : The year from where to begin the download. By default, it is set to 2015.

year_end : The year where to end the download. By default, it is set to 2016. Note that this year is not included in the download. For example, using the default arguments will download only images from 2015.

To download the data from the specified month, the file proceeds month by month. It will first download the original images in **hdf** file format from both the MOD11A1 and MOD09GQ products, process them and save the corresponding **tiff** images using the function in **modis_data_preprocessing.py**, and then delete them. We do this to comply with the storage constraints that we had (20gb) as the original **hdf** files are way larger than the saved **tiff** files in the end.

The month by month downloads are parallelized using 18 processes which allows us to download the original hdf files faster. When the images are downloaded and processed, they are saved in **tiff** format in the following folder configuration :

base_dir/
├── MODIS/
├── MOD_aaaa_MOD11A1/
├── tifs_files/
├──1km/
├──MOD11A1.Aaaaaddd.tttt.tif
├── MOD_aaaa_MOD11GQ/
├── tifs_files/
├──250m/
├──MOD11GQ.Aaaaaddd.tttt.tif

Where aaaa is the year, ddd is the day number in the year and ttt is the tile number of the image when cut to 64x64 patches. Here is a description of the main steps in the pipeline of downloading and processing the images:

- The original LST and reflectance images hdf files are downloaded for the first month of the first year and are stored in corresponding folders. The processing is then executed on the images of that month.
- First, we open the LST hdf files and the reflectance images hdf files. If there are any errors when opening the files, we ignore them.
- We use the reflectance images to calculate the normalized difference vegetation index using the formula : $NDVI = NIR - RED / NIR + RED$.
- We cut the 1200x1200 LST images and the 4800x4800 reflectance images into equal 64x64 and 256x256 patches respectively.
- We save the coupled patches that correspond to the same geographical location in both the LST and the NDVI image in tiff format. If the NDVI patch or the LST patch present any cloud or sea pixel coverage, we don't save both patches. Additionally, if an NDVI patch presents a NaN due to a division by zero, we also don't save both patches;

create_npz.py

This file can be used to transform all the saved LST and NDVI images in tiff format into a single npz compressed file so that the images can be loaded easier when training the model. To execute the file, two arguments need to be specified :

year_begin : The year from where to begin. By default, it is set to 2015.
year_end : The year where to end. By default, it is set to 2016. Note that this year is not included. For example, using the default arguments will transform only images from 2015.

When the execution is done, the database file will be saved in the base direction with the name **database.npz** . The saved file has two arrays. An array named 'lst' that contains the LST images of shape (N,64,64,2) where N is the number of LST images. The second array is named 'ndvi' and contains the NDVI images. It is of shape (N,256,256) where N is the number of NDVI images. The number of NDVI images is half that of their LST counterpart.

Training

dataloader.py

Custom dataset class to use in the pytorch default dataloader, *DatasetCustom()*. This way we can load both the ndvi and lst images at the same time. The *__get_item__* function relies on there being half as many ndvi images in the database compared to the lst counterpart, thus the index // 2. Also, the ordering of the images is important, they should be arranged in two arrays structured in this way:

[lst_day_1 | lst_night_1 | lst_day_2 | lst_night_2 | …]
[ndvi_1 | ndvi_2 | …]

Note. If this data distribution was to change then the function should be changed as well

loss.py

Definition of the custom loss class used for training, *MixedGradientLoss()* . We currently use the MSE of the original lst and the output of the model resized to 64x64 via a bilinear interpolation and the MSE of the gradient of the output and the NDVI image (this gradients edge is cut to prevent striking in the output, 16 pixels on each size works according to the depth of the model).

Both of these losses are returned as well as the total loss function which is a linear combination of the two according to the following equation:

$$\text{Mix}\, GE(Y, \hat{Y}) = \alpha MGE + \beta MSE$$

The absolute value of the gradient is calculated using the Sobel operators and then normalized between 0 and 1 using the *get_gradient(self, img)* method.

model.py

Definition of the model as the class *MRUNet(nn.Module)*, a more in depth explanation can be found in the documentation.

validation.py

This function is used to take a trained model and parse an ASTER image through it. Since we only had one aster image it is only possible to do one image. It expects to have the aster LST at 1km, 250m and the NDVI at 250m. The function takes the following arguments:

Argument	Type	Description	Default Value
model_name	string	Model name as saved in outputs folder	None
max_val	float	Maximum LST value found during normalization	333.3200048828125
data_dir	string	Data directory with all the images	./data/
base_dir	string	Output directory to place all plots	./validation/

It expects the following folder distribution with the images saved as NxN numpy .npy files. If done correctly all the output plots will be placed inside *base_dir* and the PSNR and SSIM values will be plotted to the terminal.

data_dir/
├── aster1km.npy
├── lst1km.npy
└── ndvi250m.npy

Note that the model has to run on a GPU and if the device is not found the code will raise an exception.

This script contains various functions that calculate the metrics of the different images such as the PSNR and SSIM. Their names are self-explanatory.

test.py

This function is used when a model is already trained to observe the output results and loss evolutions. It will plot the 5 input original images saved during training and the resulting LST output as well as the original inputs and the comparison histograms, all of these files are generated when training automatically. To run the code only the parameters have to be changed according to what the *m*
eta.txt file had:

- **alpha**
- **beta**
- **out_epoch:** The last output epoch saved.
- **model_name**
- **MAX_CONSTANT:** The constant used to normalize the LST images.

This is accomplished using the following functions:

❖ ***save_losses(metrics_path, out_path, alpha=None, beta=None, epochs=None, first_n=0):***

Saves the plots for the validation and training losses for the MSE and MGE. The *first_n* argument is used to start plotting at a latter epoch. The outputs are saved in out_path.

❖ ***plot_save(x, title, path):***

Saves a plot of x with the according title and path.

❖ ***save_double_histograms(x, y, labelx, labely, out_path, title, n_bins=20)***

Saves the histograms of x and y superimposed over each other.

train.py

The training script to run when training a model. It can receive many arguments, these are explained bellow.

Argument	Type	Default	Description
datapath	string	-	Path to directory containing training tif data
lr	float	0.001	Learning rate for the model
epochs	int	150	Number of epochs to train the model for
batch_size	int	4	Size of batch used in training the model
model_name	string	-	Name of the model
alpha	float	0.0001	Weight of MGE (mean gradient error) in loss function
beta	float	0.9999	Weight of MSE (mean squared error) in loss function
continue_train	string (bool)	'False'	Flag to continue training: True to continue training the model_name model, False to start training from scratch.

When run the script will create an output folder where everything will be saved. The folder structure is as follows:

output/
└── **model_name/** Name given by user to distinguish between models
  ├── ***meta.txt/*** Saves the parameters used in training to keep track of them.
  |
  ├── **metrics/** Training metadata is saved here, such as validation and train losses
  ├── **samples/** This folder is used later in validation but created here
  └── **training_data/** Here a subset of 5 images is saved under the name *orginial_data.npz*, every 5 epochs the model will save the output of these 5 images under the name *output_ep_N.npz*.

- ***main(args):***

This function initializes all the things needed for training, such as the model, all folders, saves the training parameters and arguments into a meta.txt file. Then it processes the data found in the *datapath* argument in *process_data()*, creates the dataloaders and begins training in *run_model()*, saving the model whenever the validation loss goes down, printing to the terminal the output epochs and losses. In the end it prints the total training time.

- **run_model(model, dataloader, optimizer, loss, batch_size, device, phase=None):**

This functions takes the model and performs one epoch using the data in the dataloader and backpropagates the loss. It returns *mge_loss, mse_loss, running_loss, running_loss, running_loss* which are all the cumulative losses over the batches for the

epoch. It can either train the model with phase="train", in this case it will step the optimizer and update the parameters of the model, or in phase="validation", where it will only calculate the losses without modifying the model.

- ***process_data(path, train_size=0.75, n_cores=3):***

This function will return the lst and ndvi sets for validation and training. It reads the data from the npz defined as an argument in the main and performs some processing.

The LST images are divided by the maximum value of all images, thus normalizing between 0 and 1.0. Then the gradient for all NDVI images is calculated using the *MixedGradientLoss()* class defined in loss.py. The proportion of train to validation images is defined in the argument *train_size*.

The returning arrays are: *lst_train, ndvi_train, lst_val, ndvi_val, original_lst_train, original_lst_val, original_lst, original_ndvi*. These are, in order, the LST and NDVI sets ready for training, i.e. normalized LST and normalized NDVI gradients, and the original LST and NDVI sets, which are not modified. The latter sets are not used during training.

Note: In this function (around the line 165) the total data is limited to the first 1000 images of the NDVI and 2000 images of the LST sets as to make testing the model faster. To train the whole model just comment these two lines.

Other

utility.py

This file contains various functions which serve different purposes, such as reading and writing in the .tif format, cropping the images of the different products to the according size, calculating the psnr and many others. Most of these functions are used in many files.

Classic methods *(TsHARP, ATPRK and AATPRK)*

classic_methods.py

This script will take the following images as input, LST at 250m (aster) format .bsq, LST at 1km (modis) format .img, NDVI at 250m and 1km format .img. It will then perform the super resolution using the classic methods TsHARP, ATPRK and AATPRK, these results will be saved as plots and as .npy in the output folder "./classic_methods/images".

Note that for this script to work the library Thunmpy has to be installed from the following github: https://github.com/cgranerob/ThUnmpy.

The data can be placed inside the "./data" folder, however the path to the images can be modified by modifying the corresponding variables:

- **path_aster** = "../data/AST_01262017212447_LST_VALIDATION_250.bsq"
- **path_lst** = "../data/MOD_20170126_LST_1km.img"
- **path_ndvi1** = "../data/MOD_20170126_NDVI_1km.img"
- **path_ndvi250** = "../data/MOD_20170126_NDVI_250.img"