rsds_downscale

Code written and developed by Rosie Eade, 2025.

Aim:

Downscale solar irradiance to increase temporal resolution at a point location.

Python Requirements:

- pytorch https://pytorch.org/ (BSD-3 license)
- pysolar https://github.com/pingswept/pysolar/ (GPL-3.0 license)
- see python code for other python needs

Data Requirements:

SARAH-3 Data:
https://wui.cmsaf.eu/safira/action/viewDoiDetails?acronym=SARAH_V003

For a single point location

- Download SARAH-3 data e.g. using wget script in write_data_files (including the ancillary file containing timestamp adjustments)
- Create a set of input and target time series files to train the models using python code in write_data_files (for a specific location)
    - Input = Low resolution time series, emulated using SARAH-3 data averaged over a region similar to a CMIP6 General Circulation Model (GCM) sized grid box and a 3-hour time window
    - Target = SARAH-3 time series at single point within the input region (GCM grid box) on 30 minute timesteps (instantaneous) for Total (SIS) and also Direct-only (SID) solar irradiance.

Method:

Train a machine learning (ML) model on solar irradiance satellite observations to convert low spatial and temporal resolution Total solar irradiance data into the target 30 minute output at a point location (Total and also Direct-only; SIS and SID). For comparison, a simple multiple-linear regression (MLR) model is also trained using the same data.

Choose a point location X.

Training Data

Coarse Input time series of Allsky Total solar irradiance (SIS)

- Averaged over a region representative of a CMIP6 General Circulation Model (GCM) sized grid box, e.g. 2 degrees lat/lon, that contains the point location X.
- Averaged over a 3 hour window (length 8 timesteps per day).

Fine Input time series of Clearsky Direct-only solar irradiance

- Computed using pysolar on 30 minute timesteps at point location X (length 48 timesteps per day).

Fine Target time series of Allsky Total and Direct-only solar irradiance (SIS and SID)

- Using nearest grid point to the point location X.
- Original 30 minute timesteps (length 48 timesteps per day).

The default method is to isolate the final year of the training data to be used for out-of-sample validation within the training loops.

Normalisation of Data

It is standard procedure to normalise the input and target data when training ML models as this makes the optimisation process more efficient. The normalisation methods used here are:

- Allsky Total solar irradiance divided by Clearsky Total solar irradiance (estimate function of date and time)
- Allsky Direct-only solar irradiance divided by Allsky Total solar irradiance
- Clearsky Direct-only solar irradiance divided by maximum value (identify day in training period that has maximum daily mean of Clearsky Direct-only solar irradiance)
- For night-time values where the solar irradiance is 0 (or very close to 0), this ratio is set to equal 1 to avoid issues due to dividing by 0 (see parameter SM_Thresh in rsdsMain*.py to define what is considered “very close” to 0).

The advantage of this normalisation method over using standard scalers is that:

- It is easily generalisable between different observation and climate model datasets of solar irradiance, which may have the nearest point located at subtly different locations and at slightly different timesteps
- It enables the machine learning model architecture to cap the predictions by a physically realistic maximum (by including a final sigmoid layer in the machine learning model decoder).

Estimating Clearsky Total Solar Irradiance Function

To estimate Clearsky Total solar irradiance a MLR approach is used to train a function on input values of date and time. Using Allsky Total solar irradiance data from the training period, the Maximum Allsky Total solar irradiance is computed over all training years as a function of date and time (after removing 29th February so all years have length 365 days). This is used to represent the target Clearsky Total solar irradiance which is the predictand for the MLR model, trained using predictors:

- sin(D’) and cos(D’)
- sin(T’) and cos(T’)
- Maximum Clearsky Direct-only solar irradiance as a function of day and time

where

- D’ = day in year / 365 (1st January is day 1)
- T’ = time in day in seconds / number of seconds in a day
- Maximum Clearsky Direct-only solar irradiance calculated over training years using pysolar, i.e. same method as for Maximum Allsky Total

This gives a good smooth estimate of the Clearsky Total solar irradiance that has the required periodic behaviour and can subsequently be computed for any date and timestep (see Figure 1). The same method is used separately for the 3 hour mean input data and the 30 minute target data, leading to 2 slightly different MLR functions. Due to the small sample size in the training data (generally of order just 10 years), it is likely that this function slightly underestimates the true Clearsky Total. To account for this, an estimate of the inflation factor is required (parameter maxFac: hard coded in rsdsMain*.py code files). For example, the average ratio of Maximum Allsky Total solar irradiance to the MLR smooth function for values that are underestimated (e.g. maxFac ≈ 1.05 for location X near Paris, France).

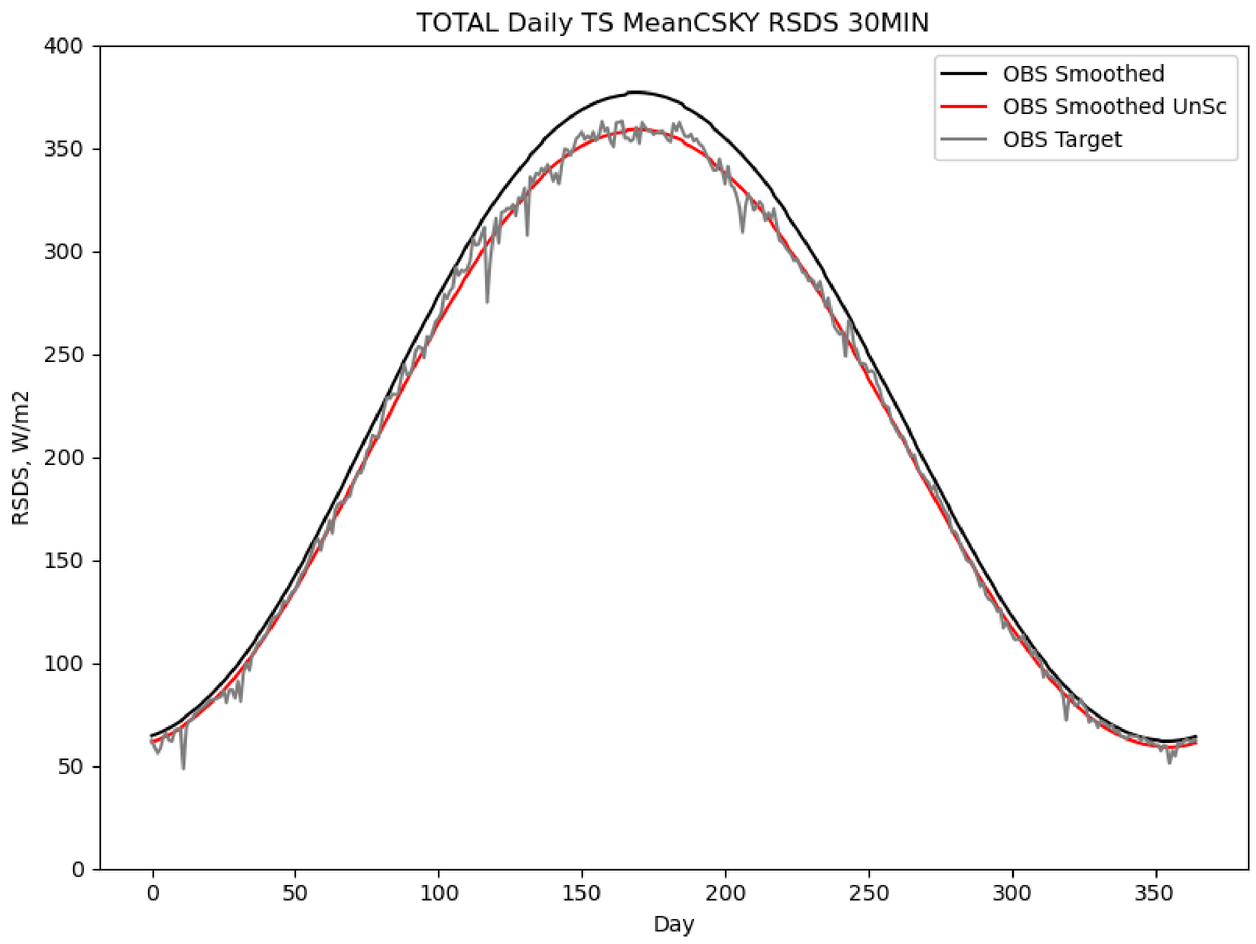

Figure 1: Example function for Clearsky Total solar irradiance versus day in the year. The gray curve shows the daily mean of the maximum Allsky Total solar irradiance computed over the training period 2013-2022 for a point location near Paris, France (computed on 30 minute timesteps then averaged to get daily mean). The red curve shows the daily mean of the MLR function fit to the sub-daily time series of maximum Allsky Total solar irradiance. The black curve is the version of the MLR function after scaling by maxFac = 1.05. This leads to a better fit for most of the year, but with a slight overestimate in the summer.

Machine Learning Model Architecture
The ML model architecture is based on a 1 dimensional Convolutional Neural Network (1dCNN) divided into an encoder part and a decoder part. The input data consists of 2 features of different length (8 and 48 timesteps per day). The encoder part uses Conv1d pytorch layers for both input features, with the addition of MaxPool1d pytorch layers for the fine input time series. The layer parameters are chosen such that the input features are transformed to have the same length, enabling the concatenation of features along the channel dimension. A pair of decoders are then used to achieve the target Total and Direct-only solar irradiance. These use ConvTranspose1d layers, with parameters chosen such that the output features have the desired length of 48 timesteps. The final layer is a Sigmoid layer to make sure that the output features are within the range [0,1] as the training data has all been normalised (see description in normalisation section above). After application of the ML model, the output data needs to be un-normalised following the reverse of the procedure to normalise.

Custom Loss Function
A custom loss function has been developed for the training of this ML model. This loss function is applied separately to the Total and to the Direct-only solar irradiance, then a final value is produced by computing the average (with the option of a weighted average: see parameter weightTOT in the function train_model() in rsds_ml_models.py).

The loss function is based on a combination of factors that are considered important (see class ClimStatsLoss() in rsds_ml_models.py):

1. Closeness of whole output sub-daily time series to target: Standard MSE Loss function on all individual timesteps, computed over all days in batch.
2. Closeness of daily mean values to target: Standard MSE Loss function on daily mean values, computed over all days in batch.
3. Closeness of daily standard deviation values to target: Standard MSE Loss function on the standard deviation of sub-daily values within each day, computed over all days in batch.
4. Closeness of sub-daily climatology to target: Standard MSE Loss function on batch mean across all days in batch, i.e. computed over all 48 timesteps in batch mean.

For loss function elements 2 to 4, there is an option to compute the loss just on daylight timesteps, i.e. ignoring the night-time timesteps. Advantages of this step are:

- The daylight hours are the most important part of the training as the night-time values will automatically be set to 0 when the ML model output is un-normalised at the end.

- The normalised time series can end up with the largest values occurring close to the transition between night and day as the normalisation method leads to division by values close to zero. It is desirable to avoid these points dominating the training as after un-normalising, they represent very small amounts of variability with respect to the variability in the more central part of the day.

The final loss function value is a weighted average of these 4 factors. The weights and the definition of daylight timesteps are parameters defined in the class ClimStatsLoss in rsds_ml_models.py. This flexible custom loss function enables some tuning of the ML model to produce the desired results for specific user case needs. For example, if only element 1 of the loss function is used, this will be more similar to the MLR model which uses a similar method for fitting the MLR parameters (least squares method). The 3rd element of the loss function is particularly useful if it is important to get realistic sub-daily variability, without trying to get the sub-daily timing of peaks and dips exactly right.

Performance Metrics:

To assess the performance of the ML model output, similar criteria are used as for the custom loss function but in relation to the un-normalised output as well as the normalised output. In addition, it is interesting to consider the added value of the ML model over the simpler MLR model. The main code files rsdsMain*.py output tables of performance metrics and some simple figures to assess the performance of the ML model, including comparisons with the MLR model (see python code files for more details). These performance metrics can be computed on the training data (in-sample tests) or on out-of-sample data. The partitioning of the data into training and test years is defined in the rsdsMain*.py code using the function define_traintest_years() in rsds_data_prep.py which can be amended as desired. It is assumed that the partitioning is defined using full years.