UNIVERSITY OF CENTRAL FLORIDA

MARGE Users Manual

MACHINE LEARNING ALGORITHM FOR RADIATIVE TRANSFER OF GENERATED EXOPLANETS

Authors:
Michael D. HIMES

Supervisor:
Dr. Joseph HARRINGTON

November 6, 2020

Contents

1 Team Members

- Michael Himes[1], University of Central Florida (mhimes@knights.ucf.edu)
- Joseph Harrington, University of Central Florida
- Adam Cobb, University of Oxford
- David C. Wright, University of Central Florida
- Zacchaeus Scheffer, University of Central Florida

We would also like to acknowledge and thank James Mang and Nick Susemiehl for testing the software and informing the User Manual instructions related to Windows and Mac.

2 Introduction

This document describes MARGE, the Machine learning Algorithm for Radiative transfer of Generated Exoplanets. MARGE generates exoplanet spectra; processes them into a desired format; and trains, validates, and tests neural network (NN) models to approximate radiative transfer (RT). At present, MARGE is configured to use BART, the Bayesian Atmospheric Radiative Transfer code, for spectra generation.

The detailed MARGE code documentation and User Manual[2] are provided with the package to assist users in its usage. For additional support, contact the lead author (see Section 1).

MARGE is released under the Reproducible Research Software License. For details, see https://planets.ucf.edu/resources/reproducible-research/software-license/.

The MARGE package is organized as follows:

```
MARGE
├── doc
├── example
├── lib
│   └── datagen
│       └── BART
└── modules
    └── BART
```

3 Installation

3.1 System Requirements

MARGE was developed on a Unix/Linux machine using the following versions of packages:

- Python 3.7.2

[1] https://github.com/mdhimes/

[2] Most recent version of the manual available at https://exosports.github.io/MARGE/doc/MARGE_User_Manual.html

- Keras 2.2.4
- Numpy 1.16.2
- Matplotlib 3.0.2
- mpi4py 3.0.3
- Scipy 1.2.1
- sklearn 0.20.2
- Tensorflow 1.13.1
- CUDA 9.1.85
- cuDNN 7.5.00
- ONNX 1.6.0
- keras2onnx 1.6.1
- onnx2keras 0.0.22

MARGE also requires a working MPI distribution if using BART for data generation. MARGE was developed using MPICH version 3.3.2.

3.2 Install and Compile

3.2.1 Unix

This is our recommended installation method. The following instructions have been verified on Ubuntu and may need to be slightly modified for other Unix distributions (e.g., Mac).
To begin, obtain the latest stable version of MARGE. Decide on a local directory to hold MARGE. Let the path to this directory be MARGE. Now, clone the repository:

```
git clone --recursive https://github.com/exosports/MARGE MARGE/
cd MARGE/
```

MARGE contains a file to easily build a conda environment capable of executing the software. Create the environment via

```
conda env create -f environment.yml
```

Then, activate the environment:

```
conda activate marge
```

Users that do not have Nvidia GPU drivers installed will need to remove the tensorflow-gpu package:

```
conda remove -n marge tensorflow-gpu
```

Follow the prompt to upgrade/downgrade relevant packages.

You are now ready to run MARGE!

Data generation with BART requires SWIG. For users that do not already have it installed, it can be easily installed via conda:

```
conda install swig
```

To use BART, its submodules must be compiled:

```
make bart
```

You are now ready to run MARGE with BART!

3.2.2 Mac

Mac users can follow the above Unix instructions to set up the conda environment. However, Mac has certain libraries installed by default that can conflict with a related library within the conda environment. This can produce some concerning Runtime Warnings when executing MARGE. After setting up the environment and activating it as detailed above, enter

```
conda install nomkl
```

into the terminal. This will solve the problem.

3.2.3 Windows

We offer support for Windows through two methods. For users that prefer the Windows Subsytem for Linux, simply follow the Unix instructions above. For users that prefer to work purely in Windows, we advise using Anaconda's terminal-like prompt.

For a pure-Windows installation, some minor adjustments are required. These instructions assume the user is working within the Anaconda prompt. First, edit the environment.yml file and remove the entry for mpich and mpi4py. Users must install the Microsoft MPI (MS-MPI) package. Microsoft's Github repo for MS-MPI includes a link to the most recent version of the package: download it and follow the installation instructions. Additionally, it requires the Windows 10 SDK. Two directories must be added to the path; their default locations are C:\Program Files\Microsoft MPI\Bin and C:\Program Files (x86)\Microsoft SDKs\MPI.

After completing the above, users may follow the steps described in the Unix section. After building the conda environment from the modified environment.yml file and activating it, users must enter

```
conda install -c intel mpi4py
```

to finish setting up MPI.

If for whatever reason the installation does not work following these instructions, installing Visual Studio may rectify that. If this is required, we would appreciate feedback letting us know so that

we may update the instructions here.

Lastly, when running MARGE, pure-Windows users may need to slightly adjust the terminal command:

```
python MARGE.py path\to\config
```

If a user finds another method to run MARGE in Windows, we encourage them to share their installation method to improve the documentation.

4 Example

The following script will walk a user through executing all modes of MARGE to simulate the emission spectra of an HD 189733 b-like exoplanet with a variety of thermal profiles and atmospheric compositions, process the data, and train an NN model to quickly approximate spectra over the trained parameter space. These instructions are meant to be executed from a Linux terminal.

We offer a lightweight example meant to be executed on a machine with 4 cores and 6 GB of RAM. Note that we are compromising the completeness and accuracy of the resulting model to ensure that the software can be executed in a reasonable amount of time (though it will still take a while!). To improve the results, users may increase the number of spectra generated (numit in BART config), use a finer opacity grid (reduce tempdelt and wndelt in BART config), increase the number of atmospheric layers (n_layers in BART config), and/or use a more complicated model architecture (more layers, more nodes).
Requirements:

- >= 4 cores
- >= 4 GB RAM available (recommended system RAM >= 6 GB)
- >= 20 GB free space

Optional:

- GPU with >= 4 GB RAM

To begin, copy the requisite files to another directory. Here we assume that directory is parallel to MARGE, called run. From the MARGE directory,

```
mkdir ../run
cp -a ./example/* ../run/.
cd ../run
```

To generate data with BART, a Transit Line-Information (TLI) file containing all line list information must be created. Download the required line lists and create the TLI file (this may take a while!):

```
./get_lists.sh
../MARGE/modules/BART/modules/transit/pylineread/src/pylineread.py -c pyline
```

Now, execute MARGE:

```
../MARGE/MARGE.py MARGE.cfg
```

This will take some time to run. It will

- generate an opacity table,
- run BART to generate spectra,
- process the generated spectra into MARGE's desired format,
- train, validate, and test an NN model, and
- plot specific predicted vs. true spectra.

Users can disable some steps via boolean flags within the configuration file. For details, see the following section.

5 Program Inputs

The executable MARGE.py is the driver for the MARGE program. It takes a a configuration file of parameters. Once configured, MARGE is executed via the terminal as described in Section 4.

5.1 MARGE Input Data Format

In order to use MARGE, users must ensure their data has been formatted into MARGE's desired format.

At present, MARGE handles 1D data cases (e.g., spectra). 2D data cases (e.g., images) will be supported at a later date, though we welcome users to add this capability sooner and submit it via a pull request.

The data must be structured as follows:

```
data
├── test
├── train
└── valid
```

The 'data' directory, specified by the configuration key 'datadir' (see the following section), can take any name the user desires. However, the 3 subdirectories must be named exactly as shown (test, train, valid). These subdirectories will hold the data files. Users may have any number of data files within the subdirectories (e.g., test could have a single file with all of the test cases, while train has 100 files with the training cases).

Each data file must be a Numpy binary (.NPY) file. Each file must be a 2D array, shaped as (number of cases, number of inputs + number of outputs). That is, each row of the 2D array will hold the corresponding inputs and outputs for a particular case.

For example, if there are 2 data cases, where Case 1 has inputs (1, 2, 3) and outputs (4, 5, 6, 7), while Case 2 has inputs (10, 11, 12) and outputs (30, 40, 50, 60), the 2D array would look like

```
[[ 1  2  3  4  5  6  7]
 [10 11 12 30 40 50 60]]
```

MARGE includes the ability to take a user-specified function to process the data. For users that wish to utilize this, look at the example functions in MARGE/lib/datagen/, and note how they are included into the example configuration file. Users that prefer to manually split their data set should set processdat = False in the configuration file.

5.2 MARGE Configuration File

The MARGE configuration file is the main file that sets the arguments for a MARGE run. The arguments follow the format **`argument = value`**, where **`argument`** is any of the possible arguments described below. Note that, if generating data via BART, the user must create a BART configuration file (see Section 5.2.1).
The available options for a MARGE configuration file are listed below.
(Directories)

- inputdir : str. Directory containing MARGE inputs.
- outputdir : str. Directory containing MARGE outputs.
- plotdir : str. Directory to save plots. If relative path, subdirectory within 'outputdir'.
- datadir : str. Directory to store generated data. If relative path, subdirectory within 'outputdir'.
- preddir : str. Directory to store validation and test set predictions and true values. If relative path, subdirectory within 'outputdir'.

Datagen Parameters

- datagen : bool. Determines whether to generate data.
- datagenfile: str. File containing functions to generate and/or process data. Do NOT include the .py extension! See the files in the lib/datagen directory for examples. User-specified files must have identically-named functions with an identical set of inputs. If an additional input is required, the user must modify the code in MARGE.py accordingly. Please submit a pull request if this occurs!
- cfile : str. Name of the configuration file for data generation. Can be absolute path, or relative path to 'inputdir'.
- processdat : bool. Determines whether to process the generated data.
- preservedat: bool. Determines whether to preserve the unprocessed data after completing data processing. Note: if False, it will PERMANENTLY DELETE the original, unprocessed data!

Neural Network (NN) Parameters

- nnmodel : bool. Determines whether to use an NN model.

- resume : bool. Determines whether to resume training a previously-trained model.
- seed : int. Random seed.
- trainflag : bool. Determines whether to train an NN model.
- validflag : bool. Determines whether to validate an NN model.
- testflag : bool. Determines whether to test an NN model.
- TFR_file : str. Prefix for the TFRecords files to be created.
- buffer : int. Number of batches to pre-load into memory.
- ncores : int. Number of CPU cores to use to load the data in parallel.
- normalize : bool. Determines whether to normalize the data by its mean and standard deviation.
- scale : bool. Determines whether to scale the data to be within a range.
- scalelims : floats. The min, max of the range of the scaled data. It is recommended to use -1, 1
- weight_file: str. File containing NN model weights. NOTE: MUST end in .h5
- input_dim : int. Dimensionality of the input to the NN.
- output_dim : int. Dimensionality of the output of the NN.
- ilog : bool. Determines whether to take the log10 of the input data. Alternatively, specify comma-, space-, or newline-separated integers to selectively take the log of certain inputs.
- olog : bool. Determines whether to take the log10 of the output data. Alternatively, specify comma-, space-, or newline-separated integers to selectively take the log of certain outputs.
- gridsearch : bool. Determines whether to perform a gridsearch over architectures.
- architectures: strings. Name/identifier for a given model architecture. Names must not include spaces! For multiple architectures, separate with a space or indented newlines.
- nodes : ints. Number of nodes per layer with nodes.
- layers: strings. Type of each hidden layer. Options: dense, conv1d, maxpool1d, avgpool1d, flatten, dropout
- lay_params: list. Parameters for each layer (e.g., kernel size). For no parameter or the default, use None.
- activations: strings. Type of activation function per layer with nodes. Options: linear, relu, leakyrelu, elu, tanh, sigmoid, exponential, softsign, softplus, softmax
- act_params: list. Parameters for each activation. Use None for no parameter or the default value. Values specified only apply to LeakyReLU and ELU.

- epochs : int. Maximum number of iterations through the training data set.
- patience : int. Early-stopping criteria; stops training after ‘patience‘ epochs of no improvement in validation loss.
- batch_size : int. Mini-batch size for training/validation steps.
- lengthscale: float. Minimum learning rate.
- max_lr : float. Maximum learning rate.
- clr_mode : str. Specifies the function to use for a cyclical learning rate (CLR). ’triangular’ linearly increases from ‘lengthscale‘ to ‘max_lr‘ over ‘clr_steps‘ iterations, then decreases. ’triangular2’ performs similar to ‘triangular’, except that the ‘max_lr‘ value is decreased by 2 every complete cycle, i.e., 2 * ‘clr_steps‘. ’exp_range’ performs similar to ’triangular2’, except that the amplitude decreases according to an exponential based on the epoch number, rather than the CLR cycle.
- clr_steps : int. Number of steps through a half-cycle of the learning rate. E.g., if using clr_mode = ’triangular’ and clr_steps = 4, Every 8 epochs will have the same learning rate. It is recommended to use an even value. For more details, see Smith (2015), Cyclical Learning Rates for Training Neural Networks. When executing a range test to determine the minimum/maximum learning rates, set this variable to ‘range test’.

Plotting Parameters

- xvals : str. .NPY file with the x-axis values corresponding to the NN output.
- xlabel : str. X-axis label for plots.
- ylabel : str. Y-axis label for plots.
- plot_cases : ints. Specifies which cases in the test set should be plotted vs. the true spectrum. Note: must be separated by spaces or indented newlines.

Statistics Files

- fmean : str. File name containing the mean of each input/output. Assumed to be in ‘inputdir‘.
- fstdev : str. File name containing the standard deviation of each input/output. Assumed to be in ‘inputdir‘.
- fmin : str. File name containing the minimum of each input/output. Assumed to be in ‘inputdir‘.
- fmax : str. File name containing the maximum of each input/output. Assumed to be in ‘inputdir‘.
- rmse_file : str. Prefix for the file to be saved containing the root mean squared error of predictions on the validation & test data. Saved into ‘outputdir‘.
- r2_file : str. Prefix for the file to be saved containing the coefficient of determination ($R\hat{2}$) of predictions on the validation & test data. Saved into ‘outputdir‘.

- filters : strings. (optional) Paths/to/filter files that define a bandpass over ‘xvals‘. Two columns, wavelength and transmission. Used to compute statistics for band-integrated values.
- filt2um : float. (default: 1.0) Factor to convert the filter files’ X values to microns. Only used if ‘filters‘ is specified.

5.2.1 BART Configuration File

The BART User Manual details the creation of a BART configuration file. For compatibility with MARGE, users must ensure two specific arguments are set within the configuration file:

- savemodel: base file name of the generated data. MUST have ’.npy’ file extension.
- modelper: an integer that sets the batch size of each ‘savemodel‘ file.

Note that ‘modelper‘ batch size corresponds to the iterations per chain. For example, if using 10 parallel chains, a ‘modelper‘ of 512 would save out files of 5120 spectra each.
Executing BART requires a Transit Line-Information (TLI) file to be created. For details on generating a TLI file, see the Transit User Manual. For an example, see Section 4.

6 Program Outputs

MARGE produces the following outputs if all modes are executed:

- simulated spectra.npy files
- processed spectra.npy files
- .NPY file of the mean of training set inputs and outputs
- .NPY file of the standard deviation of training set inputs and outputs
- .NPY file of the minima of training set inputs and outputs
- .NPY file of the maxima of training set inputs and outputs
- .NPY file of the size of the training, validation, and test sets
- TFRecords files of the data set
- file containing the NN model and weights
- predicted and true spectra
- RMSE and $R\hat{2}$ statistics
- plots of true and predicted spectra

Note that BART’s output files are not discussed here; see the BART User Manual for details.

7 Determining the Optimal Model Architecture

While users are free to train an arbitrary model on some data set, it is advised that users perform a model grid search as well as a CLR 'range test' to optimize the model parameters. This section will describe these two important steps.

7.1 Model Grid Search

A model grid search is very helpful in determining model architectures that are well suited to the problem at hand. In short, this process involves training a variety of models (different layers, nodes, activation functions, batch sizes, etc.) and finding which model(s) achieve(s) small loss values (and therefore higher accuracy). Because this process can involve training many models, it can be helpful to use only a subset of the data (ensuring it has similar statistical properties) and/or to train for fewer epochs than normal (e.g., 20 – 50, depending on the problem). This will ensure that models can be quickly trained, evaluated, and compared. It is important that these results are only compared across models trained for the same number of epochs, using the same training and validation sets.

To perform a grid search, users must set the 'gridsearch' configuration key to True. Then, each of the model architecture parameters (config keys: architectures, nodes, layers, lay_params, activations, act_params) will be sequentially considered, with a summary report produced once all models have been trained. Note that each model will be trained using the same learning rate policy, so it is often a good idea to perform the grid search over the same set of architectures using a few different learning rate policies to ensure that the 'best' architectures are found (e.g., a model that performs poorly for Policy 1 may perform best for Policy 2).

In the configuration file, each architecture must be split by indented new lines. For example,

```
layers = conv1d flatten dense dense
         dense dense dense
```

would constitute 2 separate models: one with a 1D convolutional layer followed by 2 dense layers, and the other being 3 dense layers. There would, of course, need to be 2 specifications for the other aforementioned config keys in a similar manner.

When executing the grid search configuration file, each model will be trained. At the end, MARGE outputs a summary of the minimum validation loss for each model considered, allowing a straightforward comparison of the architectures considered.

If this description seems confusing, it may be helpful to look at the supplied configuration file, MARGE_gridsearch.cfg, in the example directory. This file contains many different architectures, demonstrating how users should format these configuration keys as well as an idea about what architectures may be good to consider. It is important to keep in mind that the architectures included in this file are selected for the problem at hand, so users may need to consider simpler or more complex models, depending on their desired use case.

7.2 CLR Parameters

When using a cyclical learning rate (CLR), the selection of the minimum/maximum limits are an important consideration. This section will walk the user through this process, called a 'range test'. The end of the section also includes some discussion about the CLR modes.

First, we must properly set some variables in the configuration file:

- epochs: set to a small number, like 10
- clr_mode: triangular
- clr_steps: range test
- lengthscale: set to a small number, like 1e-7
- max_lr: set to a value closer to 1, like 1e-1

When executing MARGE, there will be a plot produced of the loss vs. learning rate value. It will look somewhat similar to Figure 1. Using this plot, the learning rate boundaries can be readily determined. The minimum learning rate corresponds to the value where the loss first begins to decrease, while the maximum learning rate corresponds to the value where the loss begins to dramtaically increase.

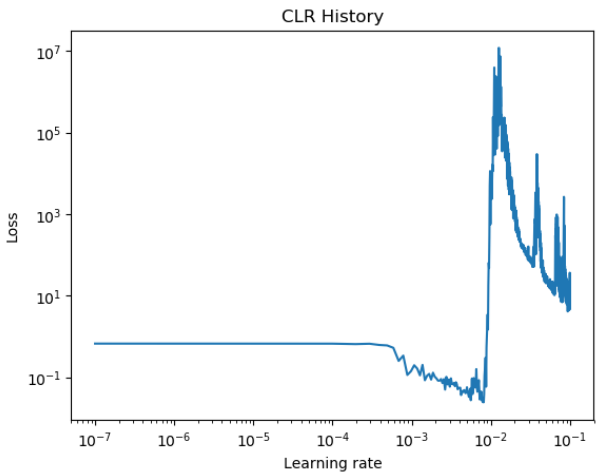

Figure 1: Example output of a range test using MARGE. For small learning rates (<5e-4), the loss remains unchanged becaues the learning rate is too small to adjust the NN's weights. Above this value, the loss begins to decrease until around a learning rate of 8e-3, where the loss dramatically increases. From this range test, a good learning rate range would be perhaps 6e-4 – 7e-3.

It is also important to consider the CLR mode when choosing these boundaries. The 'triangular' mode (which constantly varies the learning rate between the min/max values) allows good performance when choosing the min/max values from the plot, as described above. By comparison, the 'triangular2' mode (which performs like triangular, except that the max learning rate is decreased at the end of each cycle) can require users to select a slightly larger minimum value to achieve good performance. This is because, if the minimum rate corresponds to the smallest possible change in loss, once the max rate has been decreased to be effectively equal to the min rate, it will not allow

the weights to significantly change. For this reason, we recommend that users select a slightly larger min value; in the example of Figure 1, a value like 7e-4 – 9e-4 would be a good choice for the 'triangular2' policy. Similar precautions should be considered when using other CLR modes that adjust the max rate.

8 FAQ

This section will cover some frequently asked questions about training NNs, as well as specific questions about MARGE. Have a question that isn't answered below? Please send it to the corresponding author so we can add it to this section!

Q: I have a data set. How should I split it into training/validation/testing sets? A: There is no set way to split the data, but a general guideline would be something like 70% for training, 20% for validation, and 10% for testing. For large data sets that contain many more cases than are necessary to learn the problem at hand, it may be helpful to reduce the proportion for training and increase the validation and/or testing set sizes (see next Q for related discussion). However, we strongly advise against adjusting this split to improve performance on the testing set, as that situation would mean that the testing set is being used for optimization rather than testing. Users should decide on a split, train and validate the model, and then consider whether the split should be adjusted (e.g., more training cases are needed).

Another important consideration is that the training, validation, and testing sets should have roughly equal statistical properties (mean, standard deviation, minimum, maximum). This is important, as there is an implicit assumption that the training data statistically represents the other subsets of data.

Q: How much data do I need? A: This is somewhat empirical, but in general, more data is better. We generally advise against using small ($<$10,000) data sets. A helpful metric to consider is, if the parameter space were sampled on a grid, how many samples per parameter would your dataset result in? For example, if I have 100,000 training cases for a 10-parameter model, this is just over 3 samples per parameter on an equally-spaced grid ($3.16^{10} \sim 100000$). For some problems, this may be adequate. For comparison, consider 100,000 training cases for a 3-parameter model. That would be roughly equal to 46 samples per parameter, which is likely more than necessary for most problems. In that case, it would be advised to shift some of the training data to the validation and/or testing sets.

Q: How do I decide on the model architecture? A: This is determined via a model grid search. See Section 7 for details.

Q: How do I decide on a batch size? A: This is largely selected empirically, as part of the grid search. Large and small batch sizes each have their own pros and cons, so it is generally a good idea to consider a few different sizes. For some problems, larger batch sizes can perform equivalently to smaller batch sizes yet train faster, due to fewer training steps per epoch. For other problems, smaller batch sizes can achieve higher accuracy, as the mean squared error would be more sensitive to cases that perform poorly. A good starting point is to compare the performance of batch sizes of 256 and 32. Then, adjust it based on observations. On GPUs, calculations will be optimized for

2^N batch sizes.

Q: My testing set size is X, but I'm not able to plot the X-1 case. Why? A: Because of some annoying limitations of the Tensorflow version used in MARGE, the data subsets are truncated to ensure an integer number of batches. For example, if we have 15,000 test cases and are using a batch size of 256, the test set will really only have 14,848 cases (exactly 58 batches). A future update to MARGE's Tensorflow version may correct this limitation.

Q: I changed the batch size/data split/data normalization and now my results look horrible (e.g., negative R^2 values)! What happened? A: For efficiency, MARGE does not produce new TFRecords or .NPY statistics files with each run; if they already exist, they are re-used to save compute time. If you adjust the batch size, data split, or data normalization, this will adjust how the code works internally, but these important input files will remain unchanged, leading to strange behavior (e.g., negative R^2 values). Whenever adjusting anything related to the inputs, it is advised to delete the TFRecords and .NPY statistics files so that they are re-computed for the adjusted inputs.

Q: I'm on Mac, and I'm hitting an OMP error. How do I fix this? A: See the Mac subsection in Section 3.2. Mac has OpenMP installed, which can conflict with some package in the conda environment. The solution is to install the 'nomkl' conda package.

Q: When I try to train a model, I get an "unable to create file" error. What's going on? A: The most likely cause would be that the filename is too long, or that the directory the weights are being saved into does not exist. The latter can only happen when specifying an absolute path for the weight file.

9 Be Kind

Please cite this paper if you found this package useful for your research:

- Himes et al. (2020), submitted to PSJ.

```
@article{HimesEtal2020psjMARGEHOMER,
   author = {{Himes}, Michael D. and {Harrington}, Joseph and {Cobb}, Adam D
         {O'Beirne}, Molly D. and {Zorzan}, Simone and
         {Wright}, David C. and {Scheffer}, Zacchaeus and
         {Domagal-Goldman}, Shawn D. and {Arney}, Giada N.},
    title = "Accurate Machine Learning Atmospheric Retrieval via a Neural Ne
  journal = {PSJ},
     year = 2020,
    pages = {submitted to PSJ}
}
```

Thanks!