UNIVERSITY OF CENTRAL FLORIDA

MARGE Users Manual

MACHINE LEARNING ALGORITHM FOR RADIATIVE TRANSFER OF GENERATED EXOPLANETS

Authors:
Michael D. HIMES

Supervisor:
Dr. Joseph HARRINGTON

August 15, 2025

Contents

1 Team Members

- Michael D. Himes, Morgan State University/NASA GSFC (michael.d.himes@nasa.gov)
- 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, and
- Sergey Korkin for valuable feedback on the software and User Manual.

2 Introduction

This document describes MARGE, a generalized package to train, validate, and test neural network (NN) models. The name is an acronym for its original application: the Machine learning Algorithm for Radiative transfer of Generated Exoplanets. Its name remains despite that the package has been applied to multiple other domains since its development.

MARGE can optionally be interfaced with a software to generate data for NN training. At present, MARGE contains functions that interface with BART, the Bayesian Atmospheric Radiative Transfer code, to compute exoplanet spectra. We welcome community contributions to interface MARGE with additional software packages.

The detailed MARGE code documentation and User Manual[1] 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
 |   |__BART_example
 |   |__quick_example
 |__lib
 |   |__datagen
 |   |   |__BART
 |   |__loss
 |__modules
     |__BART
```

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

3 Installation

3.1 System Requirements

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

- Python 3.8.17
- Keras 2.11.0
- Numpy 1.24.3
- Matplotlib 3.7.1
- Scipy 1.10.1
- sklearn 1.2.2
- Tensorflow 2.11.0
- CUDA 12.4
- cuDNN 9.0.0
- Optuna 3.2.0
- Dask 2023.3.2

If using BART for data generation, MARGE requires a working MPI distribution. The last verified versions are given below. MPI installation is included in the BART setup instructions below.

- MPICH 3.3.2
- mpi4py 3.0.3

Additionally, if using ONNX models, the ONNX runtime and associated packages to convert from/to ONNX are required. Installation instructions are not provided for ONNX, as the package to convert ONNX to Keras models is no longer maintained. If the 'tf2onnx' package is installed, it is used to convert the trained Keras model into ONNX once training completes.

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; this guide will assume it is under your working directory called MARGE. Now, clone the repository:

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

If you wish to use BART for data generation, you must recursively 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
```

You are now ready to run MARGE!

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

```
conda install swig
conda install mpich
```

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. If you still encounter problems/warnings, contact the lead author so that it can be addressed.

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 provide the following rough guidelines, but be aware that they have not been tested in quite some time and so may not be accurate anymore.

For a pure-Windows installation, some minor adjustments to the Unix instructions are required. If using BART, users must install the Microsoft MPI (MS-MPI) package, rather than mpich and mpi4py. 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 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 that wish to utilize BART 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.

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 'example' directory in MARGE contains 2 subdirectories, 'quick_example' and 'BART_example'. These demonstrate a basic use case as well as the BART-driven use case presented in Himes et al. (2022). Note that the following examples do not seek to optimize the batch size, but that can be performed in a similar manner to what is demonstrated below.

4.1 Quick Example

This example covers training an NN to predict the (x, y) location on a unit circle from a given angle. Note that this example is extremely simple and does not require NNs to solve it. Its sole purpose is to demonstrate how to use the software.

First, generate the data:

```
 python make_data.py
```

This will create a new subdirectory ('data') which contains the training, validation, and testing data.

Now, run a 'range test' to determine reasonable limits for the learning rate policy:

```
 python ../../MARGE.py MARGE_rangetest.cfg > rangetest.log
```

Look at the history_clr_loss.png plot in the 'outputs_rangetest/plots' directory and determine the minimum and maximum learning rates as described in Section 7.3. $10^{-3} - 10^{-1}$ should be a good enough range to use. For completeness, it would be best to run the optimization for a few different (reasonable) learning rate ranges, to ensure that the optimal model architecture is not dependent on the selected range.

Next, optimize the model architecture. The BART example in the next section demonstrates a grid search, so here we will focus on a Bayesian optimization. This problem is simple, so we likely do not need many hidden layers or nodes per layer. The optimization will thus consider 1–3 layers with 2–256 nodes per layer. Look at the relevant configuration file for more details; note that it assumes there is 1 GPU available. Run it:

```
python ../../MARGE.py MARGE_optimization.cfg > optimization.log
```

This will take some time to complete. Once it finishes, look at the optuna-ordered-summary.txt file in the 'outputs_optimization' directory. From my test run, it looks like an architecture of Dense(64)—ReLU—Dense(256)—ELU(0.05585)—Dense(16)—Sigmoid is a good choice, as it had the smallest validation loss by a factor of $\sim$2. Your run will likely be different, as there are many model architectures that can be well suited to solve a given problem.

Finally, train the optimal model in full:

```
python ../../MARGE.py MARGE_run.cfg > run.log
```

When it finishes, check the resulting RMSE and R^2 values. For my selected model, it achieved a mean RMSE of $\sim 4\times10^{-4}$ and mean R^2 of 0.99999984 on the test set. This appears to be a sufficiently accurate model, so the problem is solved.

4.2 BART 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.

MARGE handles N-dimensional data. The data set 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). It is recommended that users try to balance the number of total files and the size of each file, but this is not strictly required. For example, some systems can have issues if more than 1000 files are in a directory, and systems that have limited RAM may see a performance hit or even crash if the data files are too large.

Each data file must be a Numpy binary zipped archive (.NPZ) file. Each file must contain ‘x’ and ‘y’ keys, which contain the inputs and outputs, respectively. If the 0th axes contain the same dimensionality, it is assumed that the 0th axis corresponds to the number of cases in that data file. If the 0th axes are different dimensionality, then it is assumed that each file contains a single data case. If each file contains a single data case but the 0th axes have matching dimensionality, then the user should add a new axis of size 1 to explicitly specify that.
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. Generally, developing code to utilize MARGE’s ‘processdat’ feature will only pay dividends if the user plans to make use of it multiple times.

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 an associated BART configuration file (see Section 5.2.3).
The available options for a MARGE configuration file are listed below.
General Parameters

- verb : int. Determines how much output is logged. 0: Only errors/critical information. 1: Like above, but also warnings. 2: Like above, but also important details. 3: Like above, but a little bit more detail. 4: Like above, but also the defaults assumed if a parameter is not specified. 5: Maximum detail.

Directories

- inputdir : str. Directory containing MARGE inputs.
- outputdir : str. Directory containing MARGE outputs.
- plotdir : str. Directory to save plots. If relative path, it is assumed to be relative to ‘outputdir‘, e.g., if plotdir = plots, then the directory will be ‘outputdir‘/plots.
- datadir : str. Directory to store generated data. If relative path, it is assumed to be relative to ‘outputdir‘.
- preddir : str. Directory to store validation and test set predictions and true values. If relative path, it is assumed to be relative to ‘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. **This only has an effect if processdat = True and the ‘datagenfile‘ has coded this behavior**; the BART datagen file has this feature, but datagen_pypsg.py does not. Note: if False, it will PERMANENTLY DELETE the original, unprocessed data!

Neural Network (NN) Parameters

- nnmodel : bool. Determines whether to use an NN model.
- use_cpu : bool. Determines whether to use the CPU for NN calculations. Note: This will be VERY slow when training for most real applications.
- resume : bool. Determines whether to resume training a previously-trained model.
- seed : int. Random seed. Note: currently not used within the code.
- 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. Has no impact other than naming the files.
- 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. MUST end in .keras.
- ishape : int. Dimensionality of the input to the NN. Separate each dimension by a comma. Example: 6,41,7
- oshape : int. Dimensionality of the output of the NN, specified as above.
- 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; for multi- dimensional inputs, the indices are for the last axis.
- olog : bool. Determines whether to take the log10 of the output data, as above.

- gridsearch : bool. Determines whether to perform a gridsearch over architectures.
- nodes : ints. Number of nodes per layer that has nodes. That is, if you have pooling layers, then there will be fewer entries here than the total number of layers.
- layers: strings. Type of each hidden layer. For a list of valid options, see Section 5.2.1.
- 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. For a list of valid options, see Section 5.2.2.
- act_params: list. Parameters for each activation. Use None for no parameter or the default value. Values specified only apply to LeakyReLU and ELU.
- optimize: int. Specifies the number of models to train for a Bayesian hyperparameter optimization per GPU.
- optngpus: int. Number of GPUs to use for the Bayesian optimization.
- optnlays: ints. Minimum and maximum number of layers for the optimization, separated by spaces.
- optlayer: strings. Each layer's type during the optimization. Must specify up to the maximum number of layers in the optimization. Note that layer types are not varied during the optimization. For a list of valid options, see Section 5.2.1.
- optnnode: ints. List of numbers of nodes that will be considered during the optimization, separated by spaces.
- optmaxconvnode: ints. Maximum number of nodes (feature maps) for convolutional layers. This can be used to avoid out of memory issues.
- optactiv: strings. List of activation functions to consider during the optimization. For a list of valid options, see Section 5.2.2.
- optminlr: int. Minimum learning rate that will be considered in the optimization. Use None to keep the minimum learning rate fixed during the optimization (it will use 'lengthscale' instead).
- optmaxlr: int. Maximum learning rate that will be considered in the optimization, as above.
- 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.
- min_lr: float. Minimum learning rate. It is the same as 'lengthscale', it is only a different way to specify this information.

- 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 over the same window. 'triangular2' performs similar to 'triangular', except that the 'max_lr' value is decreased such that the range is halved every complete cycle. '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. 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
At present, these only work for 1D data sets.

- 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

- fxmean : str. File name containing the mean of each input. If a relative path, it is assumed to be relative to 'inputdir'.
- fymean : str. Like above, but for the outputs.
- fxstd : str. File name containing the standard deviation of each input. If a relative path, it is assumed to be relative to 'inputdir'.
- fystd : str. Like above, but for the outputs.
- fxmin : str. File name containing the minimum of each input. If a relative path, it is assumed to be relative to 'inputdir'.
- fymin : str. Like above, but for the outputs.
- fxmax : str. File name containing the maximum of each input. If a relative path, it is assumed to be relative to 'inputdir'.
- fymax : str. Like above, but for the outputs.
- rmse_file : str. Prefix for the file to be saved containing the root mean squared error of predictions on the validation & test data. Saved relative to '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‘. Each file must have two columns, wavelength and transmission. Used to compute statistics for band-integrated values.
- filtconv : float. (default: 1.0) Factor to convert the filter files’ X-axis values to the units of ‘xvals‘. Only used if ‘filters‘ is specified.

5.2.1 Layers

The allowed layer types, how they are specified in a MARGE configuration file, whether the layer type has nodes, what layer parameter MARGE allows, and the default layer parameter used when not specified by the user are provided below. For details on the specific implementation of each layer, refer to the relevant Keras documentation.

Layer	MARGE configuration file option	Has nodes?	Layer parameter	Default layer parameter
Dense or fully-connected	dense	Yes	n/a	n/a
Dense concrete dropout	concretedropout	Yes	n/a	n/a
1D convolutional	conv1d	Yes	Kernel size	3
2D convolutional	conv2d	Yes	Kernel size	(3, 3)
3D convolutional	conv3d	Yes	Kernel size	(3, 3, 3)
2D convolutional transpose	conv2dtranspose	Yes	Kernel size	(3, 3)
3D convolutional transpose	conv3dtranspose	Yes	Kernel size	(3, 3, 3)
1D separable convolutional	separableconv1d	Yes	Kernel size	3
2D separable convolutional	separableconv2d	Yes	Kernel size	(3, 3)
1D depthwise convolutional	depthwiseconv1d	Yes	Kernel size	3
2D depthwise convolutional	depthwiseconv2d	Yes	Kernel size	(3, 3)
Dropout	dropout	No	n/a	n/a
1D max pooling	maxpool1d	No	Pooling size	2
2D max pooling	maxpool2d	No	Pooling size	(2, 2)
3D max pooling	maxpool3d	No	Pooling size	(2, 2, 2)
1D average pooling	avgpool1d	No	Pooling size	2
2D average pooling	avgpool2d	No	Pooling size	(2, 2)
3D average pooling	avgpool3d	No	Pooling size	(2, 2, 2)
Flatten	flatten	No	n/a	n/a
Batch normalization	batchnorm	No	n/a	n/a

5.2.2 Activation functions

The allowed activation functions and how they are specified in a MARGE configuration file are provided below. For convenience, some activation functions can be specified multiple ways. For details on the specific formula for each activation function, refer to the relevant Keras documentation.

Activation function	MARGE configuration file options
Exponential linear unit	elu
Exponential	exponential, exp
Gaussian error linear unit	gelu
Gated linear unit	glu
Hard sigmoid	hard_sigmoid, hardsigmoid, hardsig
Hard sigmoid linear unit (a.k.a. swish)	hard_silu, hardsilu, hard_swish, hardswish
Hard hyperbolic tangent	hard_tanh, hardtanh
Leaky rectified linear unit	leaky_relu, leakyrelu, lrelu
Linear	linear, identity, None
Logarithm of sigmoid	log_sigmoid, logsigmoid, logsig
Logarithm of softmax	log_softmax, logsoftmax
Mish	mish
Rectified linear unit	relu
Rectified linear unit truncated to a maximum value of 6	relu6
Scaled exponential linear unit	selu
Sigmoid	sigmoid, sig
Sigmoid linear unit (a.k.a. swish)	silu, swish
Softmax	softmax
Softplus	softplus
Softsign	softsign
SparsePlus	sparse_plus, sparseplus
Sparsemax	sparsemax, sparse_max
Hyperbolic tangent	tanh
Tanh shrink	tanh_shrink, tanhshrink

5.2.3 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 Files produced when running MARGE

MARGE produces the following files, depending on the executed mode(s):

- .NPY files of the mean of training set inputs and outputs (located in 'inputdir')
- .NPY files of the standard deviation of training set inputs and outputs (located in 'inputdir')

- .NPY files of the minima of training set inputs and outputs (located in ‘inputdir‘)
- .NPY files of the maxima of training set inputs and outputs (located in ‘inputdir‘)
- .NPY file of the sizes of the training, validation, and test sets (located in ‘inputdir‘)
- TFRecords files of the data set (located in ‘inputdir‘)
- a log file named YYYYMMDDThhmmss.log, where YYYY is the year, MM is the month, DD is the day, T stands for time to divide the date and time, hh is the hours, mm is the minutes, and ss is the seconds
- file containing the NN model and weights (located in ‘outputdir‘; for grid searches they will be in a nested directory named after the architecture)
- The following output files are NOT produced for range tests, grid searches, or Bayesian optimizations:
  - predicted and true targets (‘y‘ dataset in NPZ files) for the validation and test sets (located in ‘outputdir‘)
  - RMSE and $R\hat{2}$ statistics over the validation and test sets (located in ‘outputdir‘)

Note that the last 2 items in the above list are only produced when the relevant flags are set (validflag for validation targets and statistics, testflag for the test targets and statistics).
If using BART for data generation, there will also be spectra.npy files produced. For details on BART output, see the BART User Manual.

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 or Bayesian hyperparameter optimization as well as a CLR ‘range test’ to optimize the model parameters. This section will describe these steps.

7.1 Model Grid Search

A model grid search is 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 better performance). 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 – 100, 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: 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 (both in the log file and in gridsearch.txt in 'outputdir'), allowing a straightforward comparison of the architectures considered. In this file, the string identifiers for each architecture take the format of
`<layertype>-<layerparam>-nodes<#>-<activation><activationparam>--<nextla`
where items inside < > denote the user-specified variable. For convolutional layers, the `layerparam` is formatted as, e.g., `kernel3.3` for a 2D convolutional layer with a kernel size of (3, 3). Note that layer types and activation functions that do not have a parameter will not have a corresponding layer parameter or activation parameter included in this string identifier.

If this description seems confusing, it may be helpful to look at the supplied configuration file, MARGE_gridsearch.cfg, in the example directories. These files contains many different architectures, demonstrating how users should format these configuration keys. It is important to keep in mind that the architectures included in this file are selected for the problem at hand and are not exhaustive, so users may need to consider simpler or more complex models, depending on their desired use case.

7.2 Bayesian Hyperparameter Optimization

While a grid search allows the user to specify which models to consider, it is likely that a better model exists that was not considered in the grid search. To automate the exploration and selection process, MARGE offers a Bayesian optimization via the Optuna package. This requires that the user specifies the boundaries of the optimization space, and then it is automatically explored in a Bayesian manner to determine optimal architecture(s) for the problem.
To perform a Bayesian optimization, users must set the 'optimization' configuration file parameter to some positive integer, along with valid arguments for all other configuration file parameters that begin with 'opt' (see Section 5.2). An example is provided below:

```
optimize = 50
optngpus = 4
optnlays = 2 4
optlayer = dense dense dense dense
optnnode = 32 64 128 256 512 1024 2048
optmaxconvnode = 256
optactiv = relu elu leakyrelu sig tanh
```

```
optactrng = 0.01 0.5
optminlr = None
optmaxlr = None
```

The above settings would run the optimization for a total of 200 models, each with 2–4 dense hidden layers with 32–2048 nodes, with activation functions among those specified. The learning rate would not be optimized here; rather, it will use the CLR range specified by 'lengthscale' (or equivalently, 'min_lr') and 'max_lr'. In this case, it is recommended to run multiple Bayesian optimizations with different CLR parameters, as this will ensure the results are robust to the selected CLR policy.
At the end of the Bayesian optimization, two files will be produced in the output directory:

- optuna-ordered-summary.txt: a human-readable file that orders the considered models by minimum validation loss. Users can quickly determine which model architectures performed best.
- optuna-study.dat: a Pickled binary file containing the results from each considered model.

7.3 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 (or min_lr): set to a small number, like 1e-7
- max_lr: set to a value closer to 1, like 1e-1

For very large data sets, 'epochs' could be set to a smaller number like 2–5, while for small data sets, 'epochs' may need to be a larger number like 30. There is no harm in setting this to be larger than necessary, while setting it smaller than necessary will limit the accuracy of the range test for the lower bound due to undersampling.

When executing MARGE, there will be a plot produced of the loss vs. learning rate value (history_clr_loss.png, in 'plotdir'). 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 dramatically increase.

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

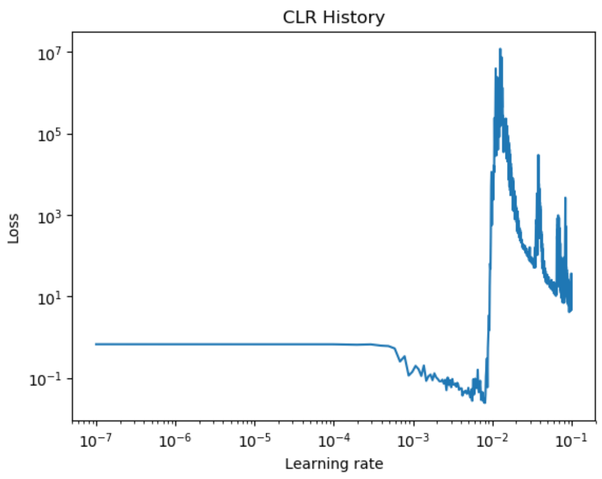

Figure 1: Example output of a range test using MARGE. For small learning rates (<5e-4), the loss remains unchanged because the learning rate is too small to adjust the NN's weights. Above this value, the loss begins to decrease until around a learning rate of 8e-3, where the loss dramatically increases.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

'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.

Note that the range test as described here can require selecting a larger minimum LR than suggested by the plot when working with data that are not deterministic, such as observational data. In this regime, at the smallest learning rates, the model can overfit the validation data and have poor generalization.

8 Using a MARGE model in another code

You've trained your model, and now you wish to use it as part of some other code, possible on another system. How is that done?

8.1 General information

Keep in mind that inference using NNs is very fast. Thus, you do not necessarily need a GPU on whatever system you plan to deploy your model on. The only hard requirement is that there is sufficient RAM to load your trained model.

When using the model, users must manually perform the desired data pre-processing in the same way that MARGE does. MARGE first applies any base-10 logarithmic transformations and then performs data scaling or normalization on those data. Note that the data statistics files saved in the inputs directory were computed on the log-transformed data. If both scaling and normalization are applied, MARGE will first apply normalization and then apply scaling; users must normalize the minima and maxima to achieve the proper transformation in their code. Note that this is equivalent to simply scaling, so it is recommended that users do not use both normalization and scaling.

Users will also need to post-process the predictions in the same way that MARGE does. This follows the opposite order as the pre-processing: first the data are de-scaled, then de-normalized, and finally any log-transforms are undone by exponentiation.

8.2 Python

This is the simplest case. First, ensure your environment is compatible with MARGE. To load the model in your code, there are two options:

- import TensorFlow and pass your `.keras` model weights file to the `tensorflow.keras.models.lo` function, ensuring you pass any custom objects (e.g., non-standard loss functions, concrete dropout layers) as a dictionary to the function's `custom_objects` keyword parameter, or
- import MARGE.py and call the main MARGE function using a copy of your configuration file with the following settings:
  - `NNModel = True`
  - `trainflag = False`
  - `validflag = False`
  - `testflag = False`

The first of those is simplest for models without any custom loss functions and/or layers, while the latter is simplest for models that use those custom loss functions and/or layers. To use the first option with custom objects, the user will need to import the relevant custom objects from the MARGE files, then assign them in a dictionary to keys that are named exactly like the objects (e.g., to use the "maxmse" loss function, the dictionary would be { `"maxmse" : losses.maxmse` }). In the latter case, the MARGE function will return the loaded model. Note that it will require some additional RAM compared to the first option, since the NNModel object holds more than just the model.

For pre- and post-processing data, users can make use of the relevant functions in `utils.py`.

8.3 Other languages

Specific instructions for your non-Python language of choice are not provided at this time. In general, users will need to search for a library that can read in a Keras or ONNX model in their desired language. Then, users will need to implement transformation functions to apply the necessary transformations as described in Section 8.1.

If a user finds a useful library for a language of their choice, please consider informing the author or submitting a pull request so it can be mentioned here. Below are two such libraries that the author is aware of, but there are surely other, possibly more feature-rich, libraries out there.

For C-based codes, users can consider the cONNXr library. Note that there may be more recent and feature-rich libraries than cONNXr, so users are encouraged to shop around. If you find a particularly useful library to make use of a MARGE-trained model, please let the author know so it can be mentioned here.

For Fortran-based codes, the neural-fortran library may be useful. Like above, if a user finds another library that is useful, please pass that information along so it can be included here.

9 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. 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 or Bayesian hyperparameter optimization. See Section 7 for details.

Q: How do I decide on a batch size? A: This is largely selected empirically, as part of the architecture optimization. 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 originally 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). This behavior may be changed in a future version of MARGE, depending on whether or not TensorFlow has corrected this limitation.

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. If this doesn't work for you, please contact the lead author.

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 to some location that isn't the 'inputdir' or 'outputdir' specified in the configuration file.

10 Be Kind

Please cite this paper if you found this package useful for your research:
Himes et al. 2022, PSJ, 3, 91

```
@ARTICLE{2022PSJ.....3...91H,
  author = {{Himes}, Michael D. and {Harrington}, Joseph and
   {Cobb}, Adam D. and {G{\"u}ne{\c{s}} Baydin}, At{\i}l{\i}m and
   {Soboczenski}, Frank and {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-net
   Surrogate Model for Radiative Transfer}",
  journal = {\psj},
  keywords = {Exoplanet atmospheres, Bayesian statistics, Posterior distribu
   Convolutional neural networks, Neural networks,
   487, 1900, 1926, 1938, 1933},
  year = 2022,
  month = apr,
  volume = {3},
  number = {4},
  eid = {91},
  pages = {91},
  doi = {10.3847/PSJ/abe3fd},
  adsurl = {https://ui.adsabs.harvard.edu/abs/2022PSJ.....3...91H},
  adsnote = {Provided by the SAO/NASA Astrophysics Data System}
 }
```

Thanks!