Neural Network for mapping interstellar medium

Arnaud Siebert

Mar 12, 2024

CONTENTS:

CHAPTER

ONE

NEURAL NETWORK FOR MAPPING INTERSTELLAR MEDIUM

1.1 CreateDataFile module

class CreateDataFile.**CreateDataFile**(*star_number*, *model*, *config_file_path*)

Bases: `object`

Class for creating a data file based on an extinction model (ExtinctionModel).

 # Args:

- *star_number (int)*: Number of stars to be used in the data file.
- *model (ExtinctionModel)*: The extinction model to be used for creating the data file.
- *config_file_path (str)*: The path to the configuration file.

 # Attributes:

- *star_number (int)*: Number of stars to be used in the data file.
- *model (ExtinctionModel)*: The extinction model to be used for creating the data file.
- *config_file_path (str)*: The path to the current test configuration file.

 # Methods:

- *execute()*: Executes the process of creating the data file.

 # Example:

The following example demonstrates how to create a data file using the CreateDataFile class.

```
>>> creator = CreateDataFile(100, model, "config.json")
>>> creator.execute()
```

execute()

Executes the process of creating the data file.

1.2 CustomLossFunctions module

class CustomLossFunctions.**CustomLossFunctions**

Bases: object

A class containing custom loss functions for PyTorch models.

 # Methods:

- loglike_loss(prediction, label, reduction_method): Computes the log likelihood loss. (Static)

 # Example:

The following example demonstrates how to use the CustomLossFunctions class to compute the log likelihood loss.

```
>>> loss = CustomLossFunctions.loglike_loss(prediction, label, 'mean')
```

static loglike_loss(*prediction*, *label*, *reduction*)

Function to compute the log likelihood loss.

 # Remarks:

This function implements the log likelihood loss function. It assumes that 'label' is a pair (extinction, sigma) and returns <((x - label(E)) / label(sigma))**2>, where <.> is either the mean or the sum, depending on the 'reduction' parameter.

 # Args:

- *prediction (torch.Tensor)*: Model predictions.
- *label (torch.Tensor)*: Labels in the form (extinction, sigma).
- *reduction (str)*: Method for reducing the loss.

 # Raises:

- *Exception*: Raised if the reduction value is unknown. Should be 'sum' or 'mean'.

 # Returns:

- *torch.Tensor*: Value of the log likelihood loss.

1.3 Dataset2D module

class Dataset2D.**Dataset2D**(*ell*, *distance*, *K*, *error*)

Bases: Dataset

PyTorch dataset for 2D extinction data.

 # Args:

- *ell (numpy.ndarray)*: Array of Galactic longitudes in degrees.
- *distance (numpy.ndarray)*: Array of distances in kiloparsecs (kpc).
- *K (numpy.ndarray)*: Array of total Absorption values.
- *error (numpy.ndarray)*: Array of errors on Absorption values.

 # Attributes:

- *ell (numpy.ndarray)*: Array of Galactic longitudes in degrees.

- *cosell (numpy.ndarray)*: Cosine of Galactic longitudes.
- *sinell (numpy.ndarray)*: Sine of Galactic longitudes.
- *distance (numpy.ndarray)*: Array of distances in kiloparsecs (kpc).
- *K (numpy.ndarray)*: Array of total Absorption values.
- *error (numpy.ndarray)*: Array of errors on Absorption values.

 # Methods:

- *__len__()* : Returns the size of the dataset.
- *__getitem__(index)*: Returns the sample at the given index.

1.4 ExtinctionModel module

`class ExtinctionModel.ExtinctionModel(N)`

Bases: `object`

Class to store a given model.

 # Remarks:
It stores the total mass, the location, the size, and the orientation of the clouds.

 # Args:

- *N (int)*: Number of clouds to generate.

 # Attributes:

- *m_tot (numpy.ndarray)*: Total mass for each model.
- *x0 (numpy.ndarray)*: Cartesian location of cloud along the x-axis [-4, 4] kpc for each model.
- *y0 (numpy.ndarray)*: Cartesian location of cloud along the y-axis [-4, 4] kpc for each model.
- *z0 (numpy.ndarray)*: Cartesian location of cloud along the z-axis [-0.5, 0.5] kpc for each model.
- *s1 (numpy.ndarray)*: Size along the first axis for each model in kpc.
- *s2 (numpy.ndarray)*: Size along the second axis for each model in kpc.
- *s3 (numpy.ndarray)*: Size along the third axis for each model in kpc.
- *a1 (numpy.ndarray)*: Orientation angle along the first axis [-30, 30] degrees for each model.
- *a2 (numpy.ndarray)*: Orientation angle along the second axis [-45, 45] degrees for each model.

 # Methods:

- *__len__* : Returns the total number of samples, which is the length of the 'm_tot' array.

1.5 ExtinctionNetwork module

class ExtinctionNetwork.**ExtinctionNetwork**(*hidden_size*)

Bases: Module

Neural network model for extinction and density predictions.

 # Args:

- *hidden_size (int)*: Number of hidden units in the neural network.
- *device (torch.device)*: Device to run the neural network on.

 # Attributes:

- *hidden_size (int)*: Number of hidden units.
- *linear1 (nn.Linear)*: First linear layer.
- *linear2 (nn.Linear)*: Second linear layer.
- *sigmoid (nn.Sigmoid)*: Sigmoid activation function.

 # Methods:

- *forward(tensor)*: Forward pass of the neural network.

 # Examples:

The following example shows how to do a forward pass using the ExtinctionNetwork class.

```
>>> network = ExtinctionNetwork(hidden_size=128)
>>> network.to(device)
>>> input_tensor = torch.randn(1, 3, device=device)
>>> output = network(input_tensor)
```

Or using forward explicitly:

```
>>> output = network.forward(input_tensor)
```

forward(*tensor*)

Forward pass of the neural network.

 # Remarks:

This method can be called implicitly by passing the input tensor to the network.

 # Args:

- *tensor (torch.Tensor)*: Input tensor of shape (batch_size, 3).

 # Returns:

torch.Tensor: Output tensor of shape (batch_size, 1).

1.6 FileHelper module

class FileHelper.**FileHelper**

Bases: object

A utility class for file-related operations.

 # Methods:

- *init_test_directory()*: Initializes a test directory based on parameters from 'Parameters.json'. (Static)
- ***give_config_value(config_file_path, key)*: Retrieves a specific value from a given** configuration file. (Static)

 # Example:

The following example demonstrates how to use the FileHelper class to initialize a test directory and retrieve a specific value from a configuration file (here the *DataSet2D* of the current test).

```
>>> config_file_path = FileHelper.init_test_directory()
>>> value = FileHelper.give_config_value(config_file_path, 'datafile')
```

static give_config_value(*config_file_path*, *key*)

Retrieves a specific value from a given configuration file.

Args: - *config_file_path (str)*: Path to the configuration file. - *key (str)*: Key for the desired value in the configuration file.

Returns: - *Any*: The value associated with the specified key in the configuration file.

static init_test_directory()

Initializes a test directory based on parameters from 'Parameters.json'.

 # Remarks:

The test directory is created based on the parameters from 'Parameters.json' and the directory name is formed by concatenating the parameter keys and values. If the directory already exists, the user is prompted to decide whether to proceed with calculations in that directory. If the user chooses to proceed, the path to the configuration file in the test directory is returned. Otherwise, a ValueError is raised.

 # Returns:

str: The path to the config file in the test directory.

1.7 MainProgram module

class MainProgram.**MainProgram**

Bases: object

A class representing the main program for training an extinction model.

 # Attributes:

- *nu_ext (float)*: Coefficient for extinction loss.
- *nu_dens (float)*: Coefficient for density loss.
- *ext_loss_function (function)*: Loss function for extinction.
- *dens_loss_function (function)*: Loss function for density.

- *star_number (int)*: Number of stars in the dataset.
- *ext_reduction_method (str)*: Reduction method for extinction loss.
- *dens_reduction_method (str)*: Reduction method for density loss.
- *epoch_number (int)*: Number of training epochs.
- *learning_rate (float)*: Learning rate for optimization.
- *config_file_path (str)*: Path to the current test configuration file.
- *device (torch.device)*: Device for running the program.
- *loader (ExtinctionModelLoader)*: Instance of the model loader.
- *dataset (torch.Tensor)*: Dataset for training.
- *maintrainer (MainTrainer)*: Instance of the main trainer.
- *network (torch.nn.Module)*: Neural network model.
- *opti (torch.optim)*: Optimization method.
- *hidden_size (int)*: Size of the hidden layer.
- *max_distance (float)*: Maximum distance in the dataset.
- *parameters (dict)*: Dictionary of current test parameters.
- *batch_size (int)*: Batch size for training.
- *is_new_network (bool)*: Flag indicating whether the network is new.

 # Methods:

- *get_parameters_from_json()*: Loads parameters from the "Parameters.json" file.
- *set_model()*: Sets the model for training.
- *set_hidden_size()*: Sets the size of the hidden layer.
- *get_max_distance()*: Sets the maximum distance in the dataset.
- *load_dataset()*: Loads the dataset for training.
- *set_parameters()*: Sets the program's parameters using loaded values.
- ***check_and_assign_loss_function(loss_function, custom_loss_function)*: Assigns the loss function based**
  on parameters.
- *create_data_file()*: Creates a data file for training.
- *train()*: Initiates and runs the training process.
- *calculate_density_extinction()*: Calculates the density and extinction values.
- *Visualize()*: Visualizes the results.
- ***execute()*: Executes the complete program, including loading parameters, setting them, creating a data file,**
  and training.

 # Example:

The following example demonstrates how to use the MainProgram class to execute the complete program.

```
>>> main_program = MainProgram()
>>> main_program.execute()
```

`calculate_density_extinction()`

Calculates the density and extinction values using the *Calculator* class.

`check_and_assign_loss_function(`*loss_function*, *custom_loss_function*`)`

Assigns the loss function based on parameters.

 # Remarks:

This method returns a callable loss function based on the given parameters. If the loss_function is custom in return the corresponding callable from the *CustomLossFunctions* module. If the loss_function is not custom, it returns the corresponding callable from the *torch.nn.functional* module.

 # Important Note:

Some of the loss functions does not take same parameters, for example, the loglike_loss function from the *CustomLossFunctions* module takes tar_batch parameters which contains the extinction and the sigma, while the *torch.nn.functional.mse_loss* function only takes the extinction as target. So be careful when using the loss functions, and make sure that the loss function you are using takes the right parameters. Some try/catch are implemented to avoid some errors and clarify the problem.

 # Args:

- *loss_function (str)*: The name of the loss function.
- *custom_loss_function (bool)*: A flag indicating whether a custom loss function is used.

 # Raises:

- *ValueError*: If the loss function is unknown.

 # Returns:

callable: The assigned loss function.

`create_data_file()`

Creates a data file for training using the *CreateDataFile* class.

`execute()`

Executes the complete program.

 # Remarks:

This method executes the complete program, including loading parameters, setting them, creating a data file, and training.

`get_max_distance()`

Gets the maximum distance in the dataset.

`get_parameters_from_json()`

Loads parameters from the "Parameters.json" file and assigns them to the MainProgram *parameters* attribute.

`load_dataset()`

Loads the dataset for training.

`set_hidden_size()`

Compute and sets the size of the hidden layer based on the dataset size.

`set_model()`

Sets the model for training using the *ModelLoader* class.

 # Remarks:

If the model is new, it creates a new model, otherwise it loads the existing model.

`set_parameters()`

Sets the program's parameters using loaded values in the *parameters* dictionary.

`train()`

Initiates and runs the training process using *MainTrainer* class.

`visualize()`

Visualize the results.

 # Remarks:

This method visualizes the results using the *Visualizer* class and saves the plots in the "Plots" subdirectory of the current test directory.

1.8 MainTrainer module

`class MainTrainer.MainTrainer(`*epoch_number*, *nu_ext*, *nu_dens*, *ext_loss_function*, *dens_loss_function*, *ext_reduction_method*, *dens_reduction_method*, *learning_rate*, *device*, *dataset*, *network*, *opti*, *hidden_size*, *max_distance*, *config_file_path*, *batch_size*`)`

Bases: `object`

Main program for training a neural network for extinction and density estimation.

 # Remarks:

This class encapsulates the main workflow for training a neural network, including configuring, setting up log files, preparing the dataset, creating the network, initializing training, and executing training steps. The training process is orchestrated by the *run* method, which calls the other methods.

 # Args:

- *epoch_number (int)*: Number of epochs for training the neural network.
- *nu_ext (float)*: Lagrange multiplier for extinction loss calculation.
- *nu_dens (float)*: Lagrange multiplier for density loss calculation.
- *ext_loss_function (callable)*: Loss function for extinction estimation.
- *dens_loss_function (callable)*: Loss function for density estimation.
- *ext_reduction_method (str)*: Method for reducing the extinction loss.
- *dens_reduction_method (str)*: Method for reducing the density loss.
- *learning_rate (float)*: Learning rate for updating network parameters.
- *device (torch.device)*: Computing device for running the neural network.
- *dataset (torch.Dataset)*: Dataset containing training and validation samples.
- *network (ExtinctionNetwork)*: Neural network model for extinction and density estimation.
- *opti (torch.optim.Adam)*: Adam optimizer for updating network parameters.
- *hidden_size (int)*: Size of the hidden layer in the neural network.
- *max_distance (float)*: Maximum distance in the dataset.
- *config_file_path (str)*: Path to the configuration file.
- *batch_size (int)*: Size of the minibatches for training and validation.

 # Attributes:

- *logfile (file)*: Logfile for recording training progress and results.
- *dataset (torch.Dataset)*: Dataset containing training and validation samples.
- *trainer (NetworkTrainer)*: Trainer for the extinction neural network.
- *train_loader (torch.utils.data.DataLoader)*: Dataloader for training minibatches.
- *val_loader (torch.utils.data.DataLoader)*: Dataloader for validation minibatches.
- *network (ExtinctionNetwork)*: Neural network model for extinction and density estimation.
- *opti (torch.optim.Adam)*: Adam optimizer for updating network parameters.
- *epoch_number (int)*: Number of epochs for training the neural network.
- *epoch (int)*: Current epoch in the training process.
- *nu_ext (float)*: Lagrange multiplier for extinction loss calculation.
- *nu_dens (float)*: Lagrange multiplier for density loss calculation.
- *device (torch.device)*: Computing device for running the neural network.
- *loss_ext_total (float)*: Total loss for extinction estimation.
- *loss_dens_total (float)*: Total loss for density estimation.
- *ext_loss_function (callable)*: Loss function for extinction estimation.
- *dens_loss_function (callable)*: Loss function for density estimation.
- *ext_reduction_method (str)*: Method for reducing the extinction loss.
- *dens_reduction_method (str)*: Method for reducing the density loss.
- *learning_rate (float)*: Learning rate for updating network parameters.
- *hidden_size (int)*: Size of the hidden layer in the neural network.
- *max_distance (float)*: Maximum distance in the dataset.
- *config_file_path (str)*: Path to the configuration file.
- *datafile_path (str)*: Path to the dataset file.
- *outfile_path (str)*: Path to the output file for storing trained models.
- *logfile_path (str)*: Path to the logfile for recording training progress and results.
- *lossfile_path (str)*: Path to the logfile for recording training loss values.
- *valfile_path (str)*: Path to the logfile for recording validation loss values.
- *batch_size (int)*: Size of the minibatches for training and validation.
- *val_ext_total (float)*: Total loss for extinction estimation on the validation set.
- *val_dens_total (float)*: Total loss for density estimation on the validation set.

 # Methods:

- *init_files_path()*: Initializes the path for the files used in the training process.
- *setup_logfile()*: Sets up the logfile for recording training progress and results.
- *setup_csv_files()*: Sets up CSV files for recording training and validation loss values.
- *prepare_dataset()*: Loads and preprocesses the dataset for training and validation.
- *create_network()*: Creates the neural network architecture based on the configuration.

- *init_training()*: Initializes the training process, setting up epoch-related variables.
- *train_network()*: Performs the training iterations, updating the neural network parameters.
- *run()*: Executes the main program, orchestrating the entire training process.

 # Examples:

The following example shows how to train a neural network using the MainTrainer class.

```
>>> main_trainer = MainTrainer(epoch_number=1000, nu_ext=0.1, nu_dens=0.1, ext_
↪loss_function=ext_loss_function,
>>>                            dens_loss_function=dens_loss_function, ext_
↪reduction_method='mean',
>>>                            dens_reduction_method='mean', learning_rate=0.
↪001, device=device,
>>>                            dataset=dataset, network=network, opti=opti,␣
↪hidden_size=128,
>>>                            max_distance=20., config_file_path='Config.json',
↪ batch_size=32)
>>> main_trainer.run()
```

`create_network`()

Create the neural network.

 # Remarks:

This method initializes and configures the neural network for extinction and density estimation. The network is created with a hidden layer size determined by a formula based on the dataset size.

`init_files_path`()

Initialize the path for the files using the current test configuration file.

`init_training`()

Initialize the training process.

 # Remarks:

This method sets up the initial conditions for the training process. It initializes the training epoch and specifies Lagrange multipliers for loss calculations. The variables *nu_ext* and *nu_dens* are used by the loss function during training. The initialization details are logged into the logfile.

`prepare_dataset`()

Prepare the dataset for training and validation.

 # Remarks:

This method loads the dataset from the specified datafile, computes normalization coefficients, and prepares training and validation datasets along with corresponding data loaders.

`run`()

Execute the main trainer.

`setup_csv_files`()

Set up CSV files for logging training and validation metrics.

 # Remarks:

This method creates and initializes CSV files for logging training and validation metrics. It opens the specified files, writes the header row, and then closes the files.

 # Files created:

- ***lossfile*: CSV file for training metrics.**

  - Header: ['Epoch', 'TotalLoss', 'ExtinctionLoss', 'DensityLoss', 'Time']

- ***valfile*: CSV file for validation metrics.**
  - Header: ['Epoch', 'TotalValLoss', 'ExtinctionValLoss', 'DensityValLoss', 'ValTime', 'TotalValTime']

`setup_logfile()`

Set up the logfile for logging information during training.

 # Remarks:

This method initializes and configures the log file based on the configuration settings. It writes information about Python and PyTorch versions, CUDA devices (if available), the selected computing device, the specified datafile, and the expected storage location for maps.

`train_network()`

Train the neural network.

 # Remarks:

This method trains the neural network using the specified training configuration. It iterates through the specified number of epochs, performing training steps on each minibatch. The training progress, losses, and validation performance are logged into the logfile. The trained model is saved to the output file every 10000 epochs and at the last step.

1.9 ModelHelper module

`class ModelHelper.ModelHelper`

Bases: `object`

Utility functions for coordinate change and integration.

 # Methods:

- ***convert_galactic_to_cartesian_3D(ell, b, d)*: Converts from galactic coordinates to** cartesian coordinates. (Static)
- ***convert_cartesian_to_galactic_3D(x, y, z)*: Converts from cartesian coordinates to** galactic coordinates. (Static)
- ***convert_cartesian_to_galactic_2D(x, y)*: Converts from cartesian coordinates to** galactic coordinates. (Static)
- ***convert_galactic_to_cartesian_2D(ell, d)*: Converts from galactic coordinates to** cartesian coordinates. (Static)
- ***integ_d(func, ell, b, dmax, model, dd=0,01)*: Integrates a function over a line of sight** in the galactic plane. (Static)
- ***integ_d_async(idx, func, ell, b, dmax, model, dd=0.01)*: Integrates a function over a line of sight** in the galactic plane. (Static)
- ***gauss3d(x, y, z, x0, y0, z0, m_tot, s1, s2, s3, a1, a2)*: Return the value of the density of a cloud** at a given point in the Galactic plane. (Static)
- ***compute_extinction_model_density(extinction_model, x, y, z)*: Computes the density of the model** at a given point in the Galactic plane. (Static)

 # Examples:

The following example shows how to convert from galactic coordinates to cartesian coordinates.

```
>>> x, y, z = ModelHelper.convert_galactic_to_cartesian_3D(ell, b, d)
```

The following example shows how to compute the density of the model at a given point in the Galactic plane.

```
>>> density = ModelHelper.compute_extinction_model_density(extinction_model, x,␣
↪y, z)
```

static compute_extinction_model_density(*exctinction_model*, *x*, *y*, *z*)

Computes the density of the model at a given point in the Galactic plane.

 # Args:

- *extinction_model (ExctinctionModel)*: Model to use.
- *x (float)*: x coordinate in kpc.
- *y (float)*: y coordinate in kpc.
- *z (float)*: z coordinate in kpc.

 # Returns:
float : Value of the density.

static convert_cartesian_to_galactic_2D(*x*, *y*)

Converts from cartesian coordinates to galactic coordinates.

 # Args:

- *x (float)*: x coordinate in kpc.
- *y (float)*: y coordinate in kpc.

 # Returns:
tuple[float, float]: Galactic coordinates (ell,d) in degrees and kpc.

static convert_cartesian_to_galactic_3D(*x*, *y*, *z*)

Converts from cartesian coordinates to galactic coordinates.

 # Args:

- *x (float)*: x coordinate in kpc.
- *y (float)*: y coordinate in kpc.
- *z (float)*: z coordinate in kpc.

 # Returns:
tuple[float, float, float]: Galactic coordinates (ell, b, d) in degrees and kpc.

static convert_galactic_to_cartesian_2D(*ell*, *d*)

Converts from galactic coordinates to cartesian coordinates.

 # Args:

- *ell (float)*: Galactic longitude in degrees (0 to 360).
- *d (float)*: Distance in kpc.

“ Returns:
tuple[float, float]: Cartesian coordinates (x,y) in kpc.

static convert_galactic_to_cartesian_3D(*ell*, *b*, *d*)

Converts from galactic coordinates to cartesian coordinates.

 # Args:

- *ell (float)*: Galactic longitude in degrees (0 to 360).
- *b (float)*: Galactic latitude in degrees (-90 to 90).
- *d (float)*: Distance in kpc.

“ Returns:

tuple[float, float, float] : Cartesian coordinates (x, y, z) in kpc.

static gauss3d(*x*, *y*, *z*, *x0*, *y0*, *z0*, *m_tot*, *s1*, *s2*, *s3*, *a1*, *a2*)

Return the value of the density of a cloud at a given point in the Galactic plane.

 # Args:

- *x (float)*: x coordinate in kpc.
- *y (float)*: y coordinate in kpc.
- *z (float)*: z coordinate in kpc.
- *x0 (float)*: x coordinate of the center in kpc.
- *y0 (float)*: y coordinate of the center in kpc.
- *z0 (float)*: z coordinate of the center in kpc.
- *m_tot (float)*: Total mass of the cloud.
- *s1 (float)*: Size along x axis in kpc.
- *s2 (float)*: Size along y axis in kpc.
- *s3 (float)*: Size along z axis in kpc.
- *a1 (float)*: Rotation angle around x axis in degrees.
- *a2 (float)*: Rotation angle around z axis in degrees.

 # Returns:

float : Value of the density of the cloud at the given point.

static integ_d(*func*, *ell*, *b*, *dmax*, *model*, *dd=0.01*)

Integrates a function f over a line of sight in the galactic plane.

 # Args:

- *func (callable)*: Function to integrate.
- *ell (float)*: Galactic longitude in degrees (0 to 360).
- *b (float)*: Galactic latitude in degrees (-90 to 90).
- *dmax (float)*: Maximum distance in kpc.
- *model (extmy_model)*: Model to use.
- *dd (float, optional)*: Step size in kpc. Defaults to 0.01.

 # Returns:

float: Value of the integral.

static integ_d_async(*idx*, *func*, *ell*, *b*, *dmax*, *model*, *dd=0.01*)

Integrates a function f over a line of sight in the galactic plane.

 # Args:

- *idx (int)*: Index of the integral.
- *func (callable)*: Function to integrate.
- *ell (float)*: Galactic longitude in degrees (0 to 360).
- *b (float)*: Galactic latitude in degrees (-90 to 90).
- *dmax (float)*: Maximum distance in kpc.
- *model (extmy_model)*: Model to use.
- *dd (float, optional)*: Step size in kpc. Defaults to 0.01.

 # Returns:

tuple[int, float]: Index and value of the integral.

1.10 ModelLoader module

class ModelLoader.**ModelLoader**(*fiducial_model_filename*)

Bases: `object`

A utility class for loading and managing *ExtinctionModel* instances.

 # Args:

- *fiducial_model_filename (str)*: File name for the dataset pickle file.

 # Attributes:

- *fiducial_model_filename (str)*: File name for the dataset pickle file.
- *is_new_model (bool)*: Flag indicating whether a new model is being created.
- *model (ExtinctionModel)*: The *ExtinctionModel* instance.

 # Methods:

- *check_existing_model()*: Checks if the dataset file already exists.
- *create_new_model()*: Creates a new *ExtinctionModel* instance and saves it to the dataset file.
- *load_model()*: Loads the *ExtinctionModel* instance from the dataset file.

 # Example:

The following example demonstrates how to use the ModelLoader class to load and manage *ExtinctionModel* instances.

```
>>> loader = ModelLoader("fiducial_model.pkl")
>>> loader.check_existing_model()
>>> if loader.is_new_model:
>>>     loader.create_new_model()
>>> else:
>>>     loader.load_model()
```

check_existing_model()

Checks if the dataset file already exists.

 # Remarks:

If the file exists, sets the *is_new_model* flag to False, indicating that the existing model will be used. If the file does not exist, sets the *is_new_model* flag to True.

create_new_model()

Creates a new *ExtinctionModel* instance and saves it to the dataset file.

load_model()

Loads the *ExtinctionModel* instance from the dataset file.

1.11 NetworkHelper module

class NetworkHelper.**NetworkHelper**

Bases: object

A utility class for building and training neural networks for extinction and density estimation.

 # Methods:

- ***integral(tensor, network_model, min_distance=0., debug=0)*: Custom analytic integral of the network for**
  MSE loss. (Static)
- *init_weights(model)*: Initializes weights and biases using Xavier uniform initialization. (Static)
- *create_net_integ(hidden_size)*: Creates a neural network and sets up the optimizer. (Static)

 # Example:

The following example demonstrates how to use the *NetworkHelper* class to build and intit network weights.

```
>>> network, optimizer = NetworkHelper.create_net_integ(hidden_size=128,␣
↪device=device, learning_rate=0.001,
>>>                                                     is_new_network=True,␣
↪epoch_number=0,
>>>                                                     config_file_path=config_
↪file_path)
>>> NetworkHelper.init_weights(network)
```

static create_net_integ(*hidden_size*, *device*, *learning_rate*, *is_new_network*, *epoch_number*, *config_file_path*)

Function to create the neural network and set the optimizer.

 # Remarks:

This function creates an instance of the *ExtinctionNetwork* neural network with the specified hidden size and initializes an Adam optimizer with the given learning rate.

 # Args:

- *hidden_size (int)*: Size of the hidden layer in the neural network.
- *device (torch.device)*: The device (CPU or GPU) on which the model is to be trained.
- *learning_rate (float)*: The learning rate for the Adam optimizer.
- *is_new_network (bool)*: Flag indicating whether to create a new network or load an existing one.

- *epoch_number (int)*: The epoch number for which the network is being created.
- *config_file_path (str)*: The path to the configuration file.

 # Returns:

***tuple[ExtinctionNetwork, optim.Adam]*: A tuple containing the created neural network and** the Adam optimizer.

`static init_weights(model)`

Function to initialize weights and biases for the given PyTorch model.

 # Remarks:
This function initializes the weights and biases of the linear layers in the model using Xavier (Glorot) uniform initialization for weights and sets bias values to 0.1.

 # Args:

- *model (torch.nn.Module)*: The PyTorch model for which weights and biases need to be initialized.

`static integral(tensor, network_model, min_distance=0.0)`

Custom analytic integral of the network *ExtinctionNetwork* to be used in MSE loss.

 # Remarks:
This function calculates a custom analytic integral of the network *ExtinctionNetwork*, as specified in the Equation 15a and Equation 15b of Lloyd et al. 2020, to be used in Mean Squared Error (MSE) loss during training.

 # Args:

- *tensor (torch.Tensor)*: Input tensor of size (batch_size, 3).
- *network_model (ExtinctionNetwork)*: The neural network model used for the integration.
- *min_distance (float, optional)*: Minimum value for integration. Defaults to 0.

 # Returns:
torch.tensor: Result of the custom analytic integral for each sample in the batch.

1.12 NetworkTrainer module

`class NetworkTrainer.NetworkTrainer(network, device, opti, ext_loss_function, dens_loss_function, ext_reduction_method, dens_reduction_method)`

Bases: `object`

A class for training the extinction neural network.

 # Args:

- *ext_loss_function (callable)*: Loss function for extinction.
- *dens_loss_function (callable)*: Loss function for density.
- *ext_reduction_method (str)*: Method for reducing the extinction loss.
- *dens_reduction_method (str)*: Method for reducing the density loss.
- *network (ExtinctionNetwork)*: The extinction neural network.
- *device (str)*: The device to be used for training (e.g. “cpu” or “cuda”).
- *opti (torch.optim)*: The optimizer to be used for training.

 # Attributes:

- *ext_loss_function (callable)*: Loss function for extinction.
- *dens_loss_function (callable)*: Loss function for density.
- *ext_reduction_method (str)*: Method for reducing the extinction loss.
- *dens_reduction_method (str)*: Method for reducing the density loss.
- *network (ExtinctionNetwork)*: The extinction neural network.
- *device (str)*: The device to be used for training (e.g. "cpu" or "cuda").
- *opti (torch.optim)*: The optimizer to be used for training.

 # Methods:

- **`take_step(in_batch, tar_batch, loss_ext_total, loss_dens_total, nu_ext,** nu_dens)`: Performs one training step.
- **`validation(in_batch_validation_set, tar_batch_validation_set, nu_ext, nu_dens, val_ext_total,** val_dens_total)`: Performs one validation step.

 # Example:

The following example demonstrates how to use the *NetworkTrainer* class to train the extinction neural network.

```
>>> trainer = NetworkTrainer(network, device, opti, ext_loss_function, dens_
↪loss_function, ext_reduction_method,
>>>                           dens_reduction_method)
>>> trainer.take_step(in_batch, tar_batch, loss_ext_total, loss_dens_total, nu_
↪ext, nu_dens)
>>> trainer.validation(in_batch_validation_set, tar_batch_validation_set, nu_
↪ext, nu_dens, val_ext_total,
>>>                    val_dens_total)
```

`take_step`(*in_batch*, *tar_batch*, *loss_ext_total*, *loss_dens_total*, *nu_ext*, *nu_dens*)

Function to perform one training step.

 # Remarks:

This function executes one training step for the neural network. It updates the network's parameters based on the provided input (in_batch) and target (tar_batch) batches.

 # Args:

- *in_batch (torch.Tensor)*: Input batch for the neural network.
- *tar_batch (torch.Tensor)*: Target batch for the neural network.
- *loss_ext_total (float)*: Total loss for extinction.
- *loss_dens_total (float)*: Total loss for density.
- *nu_ext (float)*: Lagrange multiplier for extinction loss calculation.
- *nu_dens (float)*: Lagrange multiplier for density loss calculation.

 # Raises:

- *RuntimeError*: If the target batch has a different shape than expected.

 # Returns:

tuple[float, float]: Total loss for extinction, Total loss for density.

validation(*in_batch_validation_set*, *tar_batch_validation_set*, *nu_ext*, *nu_dens*, *val_ext_total*, *val_dens_total*)

Function to perform one validation step.

 # Remarks:

This function executes one validation step for the neural network. It evaluates the network's performance on the validation set based on the provided input (in_batch_validation_set) and target (tar_batch_validation_set) batches.

 # Args:

- *in_batch_validation_set (torch.Tensor)*: Input batch for the validation set.
- *tar_batch_validation_set (torch.Tensor)*: Target batch for the validation set.
- *nu_ext (float)*: Lagrange multiplier for extinction loss calculation.
- *nu_dens (float)*: Lagrange multiplier for density loss calculation.
- *val_ext_total (float)*: Total loss for extinction in the validation set.
- *val_dens_total (float)*: Total loss for density in the validation set.

 # Raises:

- *RuntimeError*: If the target batch has a different shape than expected.

 # Returns:

***tuple[float, float]*: Total loss for extinction in the validation set, Total loss for density in the** validation set.

1.13 ParallelProcessor module

class ParallelProcessor.**ParallelProcessor**

Bases: `object`

A class providing static methods for parallel processing of extinction model predictions.

 # Methods:

- *process_parallel(model, pool, star_number, device, dtype)*: Process extinction model predictions in parallel.

 # Example:

The following example demonstrates how to use the *ParallelProcessor* class to process extinction model predictions.

```
>>> pool = multiprocessing.Pool(proocessor_num)
>>> processed_dataset = ParallelProcessor.process_parallel(model, pool, star_
↪number, device, dtype)
```

static process_parallel(*model*, *pool*, *star_number*, *device*, *dtype*)

Process extinction model predictions in parallel.

 # Remarks:

This method uses parallel processing to compute extinction model predictions for a given number of stars. It utilizes a multiprocessing pool to distribute the workload across multiple processes. The method returns a processed dataset containing the computed results.

 # Args:

- *model(ExtinctionModel)*: An extinction model for computation.
- *pool(multiprocessing.Pool)*: A multiprocessing pool for parallel processing.
- *star_number (int)*: Number of stars for which predictions need to be computed.
- *device(torch.device)*: Device (CPU/GPU) for PyTorch operations.
- *dtype(torch.dtype)*: Data type for PyTorch operations.

 # Returns:

Dataset2D: A processed dataset containing computed results.

1.14 Visualizer module

class Visualizer.**Visualizer**(*config_file_path*, *dataset*, *max_distance*)

Bases: `object`

A class providing methods for visualizing extinction model predictions.

 # Args:

- *config_file_path (str)*: Name of the configuration file.
- *dataset (ExtinctionDataset)*: Instance of the extinction dataset.
- *max_distance (float)*: Maximum distance in the dataset.

 # Attributes:

- *config_file_path (str)*: Name of the configuration file.
- *ext_grid_filename (str)*: Filename for the extinction grid data.
- *dens_grid_filename (str)*: Filename for the density grid data.
- *ext_los_filename (str)*: Filename for the extinction line-of-sight data.
- *dens_los_filename (str)*: Filename for the density line-of-sight data.
- *dataset (ExtinctionDataset)*: Instance of the extinction dataset.
- *max_distance (float)*: Maximum distance in the dataset.
- *ext_grid_datas (dict)*: Dictionary containing extinction grid data.
- *dens_grid_datas (dict)*: Dictionary containing density grid data.
- *ext_sight_datas (dict)*: Dictionary containing extinction line-of-sight data.
- *dens_sight_datas (dict)*: Dictionary containing density line-of-sight data.
- *lossdatas (pd.DataFrame)*: DataFrame containing training loss data.
- *valdatas (pd.DataFrame)*: DataFrame containing validation loss data.

 # Methods:

- *loss_function()*: Plot the training and validation loss.
- *load_datas()*: Load grid and line-of-sight data.
- *compare_densities()*: Compare true and network density predictions.
- *compare_extinctions()*: Compare true and network extinction predictions.
- *extinction_vs_distance()*: Plot true and network extinction along lines of sight.

- *density_vs_distance()*: Plot true and network density along lines of sight.
- *plot_model()*: Plot the model and save the plot.

 # Example:

The following example demonstrates how to use the *Visualizer* class to compare true and network extinction.

```
>>> visualizer = Visualizer("config.json", dataset, 20.)
>>> visualizer.compare_extinctions()
```

`compare_densities()`

Compare true and network density predictions and save the plot in the 'Plots' subdirectory of the current test directory.

 # Plot Structure:

- True density map.
- Network density map.
- True-Network density map.

`compare_extinctions()`

Compare true and network extinction predictions and save the plot in the Plots subdirectory of the current test directory.

 # Plot Structure:

- True extinction map.
- Network extinction map.
- True-Network extinction map.

`density_vs_distance()`

Plot true and network density along lines of sight.

 # Plot Structure:

The plot contains 8 subplots, each showing the true and network density along a line of sight.

`extinction_vs_distance()`

Plot true and network extinction along lines of sight and save the plot in the 'Plots' subdirectory of the

 # Plot Structure:

The plot contains 8 subplots, each showing the true and network extinction along a line of sight.

`load_datas()`

Load grid and line-of-sight data.

 # Remarks:

This method loads the grid and line-of-sight data from the files specified in the configuration file. It checks if the files exist and loads the data if they do.

`loss_function()`

Plot the training and validation loss and save the plot in the 'Plots' subdirectory of the current test directory.

`plot_model()`

Plot the model (the dataset file) and save the plot in the 'Plots' subdirectory of the current test directory.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX