Documentation

Dienstag, 18. Oktober 2022 16:41

Author: Sarah Reisenbauer

Contents:

1. Tool description

NeuralLELF is a tool for training and evaluating load estimation and load flow calculation models. To be more specific, currently implemented is the generation of Artificial Neural Networks and Linear Regression models for determining voltages at the busbars and / or the active and reactive powers of loads and generators.

2. Versions overview

The version as of Oct 2022 is: 0.0.0

3. Documentation

The package provides a library of function that may be used directly, as well as two sample scripts for execution: runner.py and evaluator.py. Both are described in detail below.

The general assumption in the training process is that a certain or multiple grid scenario of observed and unobserved quantities is present. These can be put in via a json-file or generated automatically. Furthermore, multiple grids (datasets) can be trained on simultaneously in a single go.

runner.py:

Arguments

Arguments can be passed / specified either with command line parameters or within the script:

Arguments are:

dir_results: Directroy name for results as subdir of the hardcoded value "results_path" - a directory within the working directory

le_model_dir: Optional (for mode LFfromLE required), a directroy with trained models for mode LFfromLE (a previous dir_results)

mode: the mode, defines the features and labels; either of 'LE','DLEF','DLF','BusLF','LF' or 'LFfromLE'

dir_name: general directory name pattern of the data, important if using multiple datasets that are discerned by an appendix in the directory name

name_list: individual directory name pattern of the data, important if using multiple datasets that are discerned by an appendix in the directory name

name_single: name of the data CSV-file; use this in case of a single datafile, else use name_list and here put None

file_pattern_loads: only if using multiple datasets - data file name pattern for loads; e.g. 'full_df_load.csv'
file_pattern_voltages: only if using multiple datasets - data file name pattern for voltages
file_pattern_pflow: only if using multiple datasets - data file name pattern for pflow
file_pattern_qflow: only if using multiple datasets - data file name pattern for qflow

graph_pattern: only if using multiple datasets - graph data file name pattern parts as list; e.g. ['CLUE_Test_','_Load_Bus_Mapping.txt']
graph_name: graph data file name if using a single datafile; e.g. "CLUE_Gasen_Load_Bus_Mapping.txt"

known_buses: names of known buses (these are not considered in mode LF); enter the original name from the dataset

attempts: number of scenarios per fraction of known loads - either randomly generated if no scenarios file was provided or read from the file (if read from file the attempts value has to fit the file entries - for injected scenarios enter the total number of scenarios); int

n_known: number of known loads (and corresponding buses) per scenario; set to -1 means all possible fractions are included; int

cv: number of cross validation runs, int, minimum of 2

nodes_from_X: if True then input layer of network has nodes equal to features in training data; overwrites inodes value in that case

inodes: number of nodes in the hidden layers

hidden_layers: hidden layer configuration as list of integers - the amount of hidden layers in different runs, idx is the run; e.g. [0,2] produces a net with no hidden layer and one with two

activation: activation function as tuple of str e.g. ('relu',) - the comma is required if a single value is entered

optimizer: optimizers as tuple of str e.g. ('adam',) - the comma is required if a single value is entered

loss: loss functions as tuple of str e.g. ('mse',) - the comma is required if a single value is entered

learning_rate: learning rates as tuple of float e.g. (0.001,) - the comma is required if a single value is entered

batch_size: batch size as tuple of int e.g. (32,64) or (32,) - the comma is required if a single value is entered

Direct entry

Argument can be directly entered in the script file in part:

"Alternative 2: Parameters specifications within the script"

Command line arguments

Example for a command line call of the script with arguments as:

--dir_results 2022_07_06_DLF_2 --mode DLF --name_single df_res.csv --known_buses node_174 --graph_name CLUE_Gasen_Load_Bus_Mapping.txt --attempts 10 --n_known 2 --cv 3 --inodes 0 --hidden_layers 0 1 --dir_name raw_Gasen_ --name_list v4_13June --activation relu --batch_size 32 128 --learning_rate 0.01 --loss mse --optimizer adam

Data input and encoding

Input data for training is loaded into a Dataset object. This is done for every dataset present (single to multiple) so that the individual datasets are then stored in dataset_dict, the dataset dictionary with keys as in the name_list.

Dataset and generators

dataset.py
An abstract dataset class is provided as a guide for creating dataset subclasses for each type of dataset encountered.
It contains the fields:

pq_df	pandas dataframe of active and reactive powers of loads for records; requirement: active and reactive powers corresponding to the same load have to have a consecutive odd and even column index
v_df	pandas dataframe of voltage at busbars for records
pflow_df	pandas datafreme of active power flow at busbars for records; requirement: busbars need to be in the same order as in the v_df
qflow_df	pandas dataframe of reactive power flow at busbars for records; requirement: busbars need to be in the same order as in the v_df
flow_present	boolean, if flow data is present or not
graph_df	a dataframe containing a load-to-bus matching

known_buses	list of name(s) of buses that are always observed in all the scenarios

A dataset has a single abstract method for its creation: create_dataset()

dataset_generators.py
Datasets can be retrieved from a single or multiple input files - for each case one creates a separate generator. The generator needs to use the create_dataset method to fill the fields of the dataset. Below the conventions for column names are given.

Column encoding

Convention for column names in the dataset dataframes to achieve distinct column names with certainty:

Dataframe in Dataset	Suffix
pq_df - active power	<original column name>_P
pq_df - reactive power	<original column name>_Q
v_df - voltage	<original column name>_V
pflow_df - active power flow	<original column name>_p
qflow_df - reactive power flow	<original column name>_q

e.g.: UL001_P, UL001_Q, Bus001_V, Bus_001_p, Bus_001_q

The agreed on nomenclature is as follows:
Voltage amplitude and phase - V, phase
Active power flow (busbar) - Pflow
Reactive power flow (busbar) - Qflow
Active power P
Reactive power Q

Graph data

The graph data currently is required as input and needs to be dataframe containing two columns named 'load' and 'bus' filled with matching load and bus names (position-wise).
The graph data is currently generated with PowerFactory. The graph data is used to find observed loads at the known voltage locations. These loads are being assumed to be observed too.

Mind that the namings of columns in the Dataset dataframes and the graph dataframe must match.

Scenario creation or loading - LEDTO

Scenarios - LEDTO

A scenario is defined as a single state estimation / load estimation run with a well defined occurence of known/unknown loads, voltages and flows in the grid.

load_estimation.py
The scenario information is stored in the load estimation data transfer object (LEDTO) during

scenario loading or creation.

self.pq_ind_known -> list of scenarios with a list of column indices (pq_df) of known loads in each
self.v_ind_known -> list of scenarios with a list of column indices (v_df) of known buses in each
self.nndto_list -> list of NNDTO objects, one per scenario; contain neural network training parameters and models
self.last_index -> int, last index in the scenarios list that was done in training (for continuation)
self.metrics_summary_df -> dataframe, summary results of scenarios evaluation

Example for pq_ind_known: [[0,1], [4,5,8,9], …]

Creation / loading

Scenarios are created within the script or loaded from a file (in the mode 'LFfromLE' scenarios are always loaded from a file in a folder of a previous load estimation run; the mode 'LF' doesn't use scenarios).

A **valid scenarios file** has to contain a string with a list of scenarios that contain a list of strings of columns names that are known in that scenario. The columns are the known loads, so columns with suffixes '_P' and '_Q'.
Example string in a file with two scenarios:
[[\"Battery_Storage_P\", \"Battery_Storage_Q\"], [\"Hydrogen_Storage_P\", \"Hydrogen_Storage_Q\", \"load_448_P\", \"load_448_Q\"]]

The function 'load_or_create_scenarios' determines how to proceed in each case:

load_estimation.py
load_or_create_scenarios(run_path,name,dataset,attempts,n_known):
Create random scenarios or load scenarios from a 'scenarios... .json' file at run_path. The run_path is also checked for results from a previous run - if found the corresponding scenarios won't be rerun (continuation).

look_for_valid_scenarios(…,name,…)
Checks if the run_path contains scenarios files of type 'scenarios... .json' for a certain name.
Found scenarios in a file are checked to fit the number of attempts and n_known.
If the scenarios are valid a check is done if training results are present too.

Uses:
load_scenarios():
Load the scenarios info if a scenarios file was found at run_path and encodes the column names therein to dataframe indices with 'scenarios_encoding()'
scenarios_encoding(dataset,scenarios,encode):
Encode / decode the scenarios from a file, encode is to transform string column names to dataframe column indices and sort them ascendingly. Decode is reversed (no sorting).
scenario_validation():
(only if scenarios were found in the run_path)
Assert that the scenario file contents (loaded into pq_ind_known) fits the

run parameters
defined through attempts and n_known. It does not check the contents of the elements though.
If n_known = -1 it will check if the number of scenarios = attempts*(number of loads -1)
If n_known = int it will check if the number of scenarios = attempts
Returns: scenarios_valid (bool)

confirm_models_results_params()
(only if scenarios are valid)
Confirm that a path contains files including the name and of type 'model...h5', 'parameters...json' and 'results_df...csv'. The correct number of files is not investigated.

Returns:
scenarios_found (bool): True if a scenarios json file was found
scenarios_valid (bool): True if scenario file content is validated
models_results_parameters_present (bool): True if training results files are present

Depending on these three returns the function proceeds as:
1. scenario_found = True and scenario_valid = True -> use the found scenarios at run_path and continue at the last index in the model results files found in the path (continuation)
2. scenario_found = False -> create_random_scenarios()
3. scenario_found = True and scenario_valid = False -> raise error that file doesn't fit run parameters

Case 1 and 2 will return an LEDTO object for the 'name', which is then stored in a dictionary with key being 'name'.

Case 2:
create_random_scenarios(pq_df,v_df,attempts,n_known,graph):
Creates scenarios randomly according to attempts, n_known and the number of loads n_loads present in the dataset.

Uses:
random_scenarios(attempts, n_loads, n_known, seed)
Generates scenarios with the amount of known loads varying from 1 to the number of loads minus one (n_known=-1) or with a defined number of known loads (n_known).
Seed is used as seed for the random number generator.

Contains a while loop with at max. 500 attempts of drawing random scenarios:
First, a sample of size of known loads in that scenario is drawn from range(n_loads). Then a check is done if the sample is not already present and if so it is stored in the list known_loads_scenarios, which is returned in the end and looks for example like [[0],[3,4],…].

known_loads_to_indices(known_loads_scenarios):
Takes a list of lists of scenarios containing the known loads in the scenarios and returns a list of scenarios with indices of loads as found in the pq_df dataframe columns.

Args:
known_loads_scenario (list): e.g. [[1],[2],[1,4],...]
Returns:
pq_ind_known (list): e.g. [[2,3],[4,5],[2,3,8,9],...]

create_ledto(pq_ind_known,graph,pq_df,v_df,ledto_prefilled=None):
Takes scenario info from pq_ind_known and matches the known busbars from the graph data with match_loads_buses() to retrieve v_ind_known. Fills the information into a newly created LEDTO object and returns it.

Returns an LEDTO object.

☐★ to do: load_scenarios_LFfromLE

Modes

There are currently 6 modes to choose from that define how the data is split into features and labels.

LE - load estimation

Features	Labels
'known' P, Q, V selected V - at those busbars also Pflow and Qflow	'unknown' P,Q

DLEF - direct load estimation and load flow (renamed from DLF of version 1.2.0)

Features	Labels
'known' P, Q, V selected V - at those busbars also Pflow and Qflow	'unknown' P,Q, V

DLF - direct load flow

Features	Labels
'known' P, Q, V selected V - at those busbars also Pflow and Qflow	'unknown' V

BusLF - busbar load flow

Features	Labels
'known' V selected V - at those busbars also Pflow and Qflow	'unknown' V

LF - load flow

Select all loads as features and all busbars as labels. No scenarios, no known buses.

Features	Labels
all P, Q	all V

LFfromLE - load flow from load estimation (requires a trained model that predicts the loads)

★ **currently not functional**

Features	Labels
'known' P, Q, V predicted P, Q from load estimation selected V - at those busbars also Pflow and Qflow	'unknown' V

Training, data splitting, hyperparameters and CV

First, the known_buses are mapped to indices of the v_df in the dataset (known_buses_ind).

Training is done in two for-loops:
1. over name and corresponding LEDTO
    2. over the scenarios in the LEDTO (here the last_index of the LEDTO is taken as starting point)
        Depending on the mode different functions are used to select features and labels:
        'LF': select_feature_label_scaled_LF()
        'LFfromLE': uses select_feature_label_scaled_scenario and other functions
        'LE', 'DLEF', 'DLF', 'BusLF': use select_feature_label_scaled_scenario

        All of these generate two dataframes, X_le - containing the features and y_le containing the labels. They are then transformed to numpy arrays for the training. The hyperparameter grid is constructed and for each an NNDTO object is created to store the training info and models in. The training is performed in parallel cross-validation runs. Details of all these functions are discussed in the following.

load_estimation.py
select_feature_label_scaled_scenario(pq_ind_known,v_ind_known,dataset,mode,known_buses_ind)
Select data as either features or labels according to the scenario. Scales features and labels by data type (loads and flows - MinMax, voltages - Standard)
Args:
pq_ind_known (list): the column indices of the 'known' loads; it is assumed that even numbers represent P-parts, uneven number Q-parts of the loads.
v_ind_known (list): the column indices of the 'known' buses
mode (string): defines the splitting of features and labels; either 'LE','DLEF','DLF','BusLF'
known_buses_ind: indices of buses that are always known (also used for power flows)

Raises an error if all load and voltage columns are indicated as being known or if none of them are known. Also if after the selection process either y_le or X_le has no column entries.

Returns: X_le, the dataframe with features, y_le, the dataframe with labels and scaler_y, a dataframe with a scaler for each column of y_le.

First, the pq_df and v_df data is scaled and put into a combined dataframe data_le with the combined column names. If the pflow_df is not None and there are known_buses then also the corresponding power flow data from these buses is selected, merged into the dataframe X_le_flow and the data inside is scaled.

Then the selected mode defines the further steps:
'BusLF': errors are raised if the number of known_buses equals the v_df columns, if the number of known buses in the scenario equals the v_df columns or if no busbar is known. Otherwise, the features X_le are selected as the v_df columns with indices v_ind_known of known busbars in the scenario plus the always known busbars. Then if present also the known power flows are added from the known_buses. The labels y_le contain the unknown busbars.

'LE', 'DLEF', 'DLF': features and labels are selected according to the tables of the modes above.

Scalers for labels:

For doing predictions in the original scaling one needs to be able to scale and rescale the input / output of the model - therefore scalers for each column are provided with:

load_flow.py

create_scalers(df, scaler):

Create a DataFrame with fit scalers for each column of df.

Args:

scaler (string): identification of scaler, either 'minmax' or else a StandardScaler is used

Returns: a dataframe with columns as the original dataframe df and the first row containing a scaler for that column.

- [ ] to do: select_feature_label_scaled_LF
- [ ] to do: feature and label selection in mode LFfromLE

For each scenario, after the feature and label selection process the neural network shapes and the hyperparameter grid for training is created. All generated parameter combinations will lead to a trained model from which the best performing one is later determined.

If 'nodes_from_X' is set True, then the input layer size for the neural network is determined from the number of input features (columns in the X-array) - this overwrites the 'inodes' value. The nodes, layer and hyperparameter information is put into the parameters-dictionary. On this the sklearn 'ParameterGrid' function is used to create all possible permutations of the inputs (grid). For the training run of one such grid of parameters a new storage object is instantiated - called NNDTO (neural network data transfer object). It has the following fields that are mostly filled up during the training later on:

scalers		fitted sklearn scalers
gridparams		sklearn parameter grid
results_df	dataframe	NN training results
results_models	list	containing trained NN models
results_models_paths		
Xtest	array	test input dataset
ytest	array	test label dataset
df_avg		
best_model_path		
best_model	model	trained model, best over hyperparameters
model_linreg	model	trained linear regression model
metric_df		
metrics_linreg_df		
fraction_known	float	fraction of loads that were known in the scenario

The actual training of the networks happens in the function:

grid_cv_neural_parallel(grid, X_le, y_le, cv)

Create and train neural nets with parameters according to a ParameterGrid grid and evaluate. Cross-validation is performed.

First, X_le and y_le are split into cross-validation datasets (their number equals cv), the test dataset size is fixed to a factor of 0.15. For this the function 'cross_val_split' is used which itself uses the sklearn KFold method (with data shuffling). That means the neural nets are trained on K-1 folds of data while one fold is left out and used for validation.

Then a for-loop runs over the parameter grid and in each step a number of cv models are being trained in parallel with the joblib function 'Parallel':

each of these parallel executions uses the function 'grid_point_cv_neural' and returns a trained model and the current grid-parameters (dict) expanded with some training results (grid_element_return).

The function returns the training results collected in a dataframe 'grid_elements_df', the trained models and the test datasets.

The return values are then stored in the NNDTO. The NNDTO is appended to the nndto_list of the current LEDTO and the training results and models are stored to disc (results_df <name>_<index>.csv, model <name>_<index>_<model_index>.h5).

Results determination

The metric used for the performance evaluation is the same as the loss function defined in the hyperparameters and can currently either be the mean absolute error or the mean squared error. The result is then determined by calculating the prediction on the test dataset and then the metric of the prediction and the label test data. This is the value stored in the results_df.

Outputs and log

scenarios <name>.json

Contains a string of the list of lists of scenarios that were used in the training run. E.g.: "[[\"PV_495_P\", \"PV_495_Q\"], [\"load_388_P\", \"load_388_Q\", \"load_450_P\", ...]"

Such scenarios files may be used as input for another run and are automatically detected and used (if put into the results directory).

results_df <name>_<index>.csv

Contains a table of the training results in the form (example):

index	cv index	activation	batch_size	layers_nodes	learning_rate	loss	opt	metric	time	epochs
0	1	relu	32	(12,)	0.01	mse	adam	0.016	120.5	24
0	2	relu	32	(12,)	0.01	mse	adam	0.015	130.5	22
0	3	relu	32	(12,)	0.01	mse	adam	0.02	123.9	21

1	1	tanh	64	()	0.1	mse	adam	0.45	240.6	38
…										

The metric value is evaluated from the trained model with the test dataset and the specified loss function. Blocks of rows with the same index are results from training in the cross validation attempts (on the same hyperparameter set). The variation of the corresponding metric values indicates how trustworthy a result is.

A_log.txt

Stores most of the specified arguments for a certain run like directory information, the mode, the data sources, and the attempts and n_known for scenario creation. Furthermore, the input nodes value for the NNs and the number of corss-validation attempts.

It is the last file to appear in the results directory and indicates termination of the training process without exceptions.

Further outputs are the fitted scalers for the input data (scaler_X_<index>.pkl) and the label data (scaler_y_<index>.pkl) for each scenario as well as the names of the feature columns (feature_columns_<index>.json) and the label columns (label_columns_<index>.json)

evaluator.py:

The evaluator script is supposed to be run after a successful training of neural networks. It looks for the file outputs of the training in a spcified directory, loads their content and does evaluations on them as well as visualizations.

Arguments

dir is the name of the directory with the results of the training

exlude_buses_in_evaluation is a list of bus names that should be excluded from the evaluation

pred_plot_buses

save_fig_scen is a boolean value for determining

save_fig_frac

metric_rmse can be set to True to use the root mean square error instead of the mean square error for evaluation

All the other arguments are as in the runner.py by their name.

Data and model loading

The data loading should be done in the same way as in the runner.py.

The models and results from training are being loaded in LEDTO objects, one per name in the

name_list (dataset or grid). The LEDTOs are stored in the results_dict dictionary.

load_ledto(path, name, dataset, load_models_too=False)
Will load the scenarios-json file and a list of NNDTO objects with filled fields of results_df. If load_models_too is True the models will be loaded too into a list into the NNDTOs otherwise only the paths are stored.

Result determination / outputs

In the model evaluation part two similar loop are being executed as in the runner: the outer loop goes over the grids (names, LEDTOs) while the inner goes over the scenarios for each grid. For each scenario depending on the mode the feature and label datasets are determined as in the runner.py. Train and test datasets are generated with the sklearn function train_test_split, again with a test size fraction of 0.15 and shuffling.

Evaluaton starts with the function:

get_best_model(results_df, results_models_path, load_models)
Selects the best performing model from results (averaged over all CV attempts). Returns a dataframe (df_avg) with the averaged values for the metric, time and epochs over the cv attempts and the model path to the best performing model (determined from the averaged metric value). Here, the first model of the best performing cv group is taken.

If load_models is False, the model path is not returned, only the dataframe.

This yields an averaged dataframe (df_avg) from the results_df and a path to the best model which is subsequently being loaded to the nndto.best_model field.

Linear Regression:
As a baseline comparison model a linear regression is performed on the training dataset (which may be larger than the dataset the NNs are trained on depending on the number of cv attempts) and saved to the results directory as lin_reg_model_<index>.pkl and to the nndto field model_linreg.

Generation of the metrics_summary_df:
This part takes up most of the script file. The metrics_summary_df is a dataframe of the shape:

index	scenario	fraction	metric_avg_bus	r2_avg_bus	… further metric columns
0	[8,9]	0.05			
1	[6,7,2,3]	0.1			
…					

The metric columns are named as:

Field name (examples below)	Legend to the field names: avg - averaged over label columns (depending on the 'mode')

	metric - either RMSE or MAE r2 - coefficient of determination sc - metric was derived on the scaled data bus - calculated for buses P - calculated for active powers Q - calculated for reactive powers var - variance of metric or r2 LR - linear regression metrics
metric_avg_bus	
metric_sc_avg_bus	
r2_avg_bus	
r2_sc_avg_bus	
metric_var_avg_bus	
r2_var_avg_bus	
metric_avg_bus_LR	
metric_sc_avg_bus_LR	
r2_avg_bus_LR	
r2_sc_avg_bus_LR	

The metrics_summary_df is generated from individual 'metric_df' dataframes that are retrieved within a loop over the nndtos (scenarios) with two steps, 1)

get_metrics_df(columns, nndto, best_model, metric)
columns are the name of those that are being predicted, the function performs a prediction with the best model and the test dataset stored in the NNDTO and calculates metrics on the scaled and rescaled data and returns them in a dataframe

and 2) with passing the linear regression model to get_metrics(). Both returned dataframes are subsequently merged to the final metric_df.

metric_df has the following schema:

index	ycolumn	metric	r2	metric_sc	r2_sc	metric_var	r2_var	metric_LR	r2_LR	metric_sc_LR	r2_sc_LR
0	'bus00'	0.03	0.99	…	…	…	…	…	…	…	…
…											

In case the RMSE was chosen as desired metric, the square root is or a first order approximation of it depending on the column value type is taken.

Then a new row for the current nndto (scenario) in the loop is added and filled with the scenario information and the fraction of known loads in the scenario.

Depending on the mode, the averages over active powers, reactive powers or

buses in terms of the metrics are calculated and filled into the fields of the current row. Excluded buses are not considered in this evaluation.

The final metrics_summary_df is stored to a csv-file in the results directory 'metric_summary_df <name>.csv'.
Also, all the individual metric_df are stored with names 'metric_df <name>_<index>.csv'.

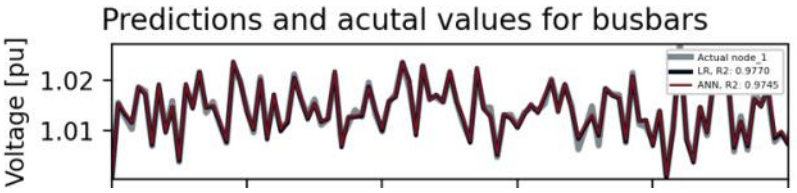
In the last section of the script, plotting options can be chosen.

Visualizations

load_estimation_viz.py

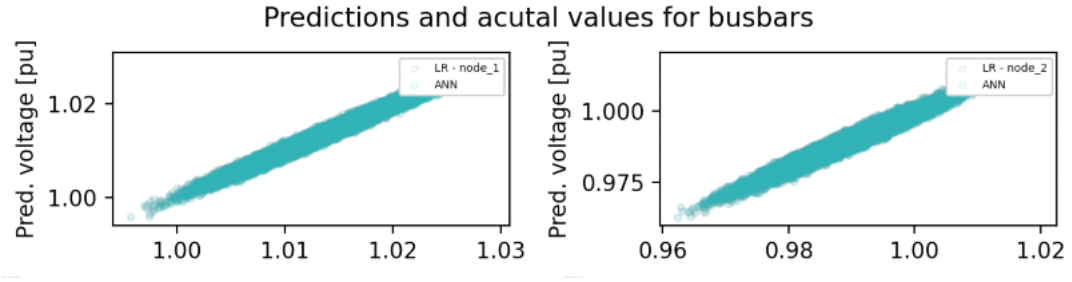
plot_predict_voltage()

Plot the original trace with an overlay of the ANN and the linear regression predictions for a test dataset. Up to 13 traces are plotted. The desired busbars may be specified (pred_plot_buses), otherwise the first up to 13 buses are taken.
Example output:



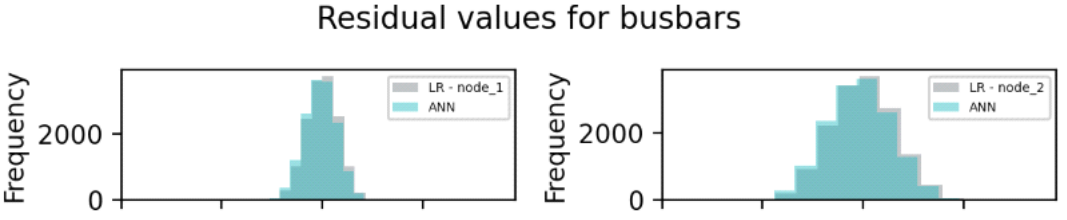
plot_actual_pred_voltage()

Plot scatter of actual vs. predictions - up to 12 plots. Busbars may be specified otherwise the first 12 are taken.
Example output (top part of figure, xscale is actual values):



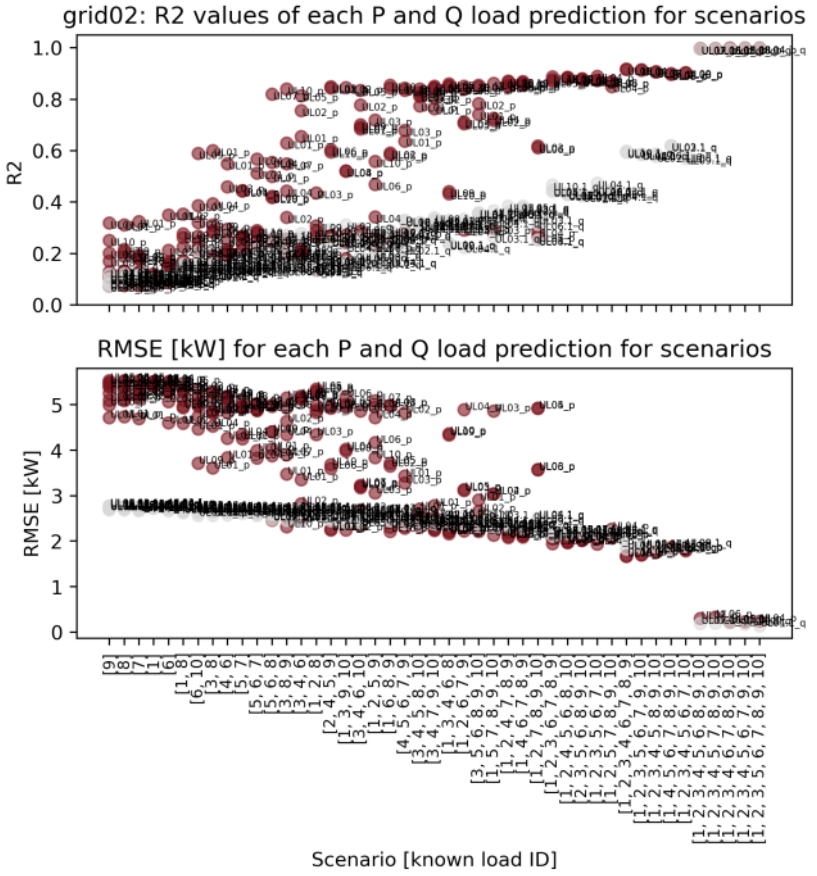
plot_residuals_voltage()

Plot histograms of the residuals of busbars - up to 12 plots. Busbars may be specified otherwise the first 12 are taken.
Example output (top part of figure, xscale is in voltage pu):



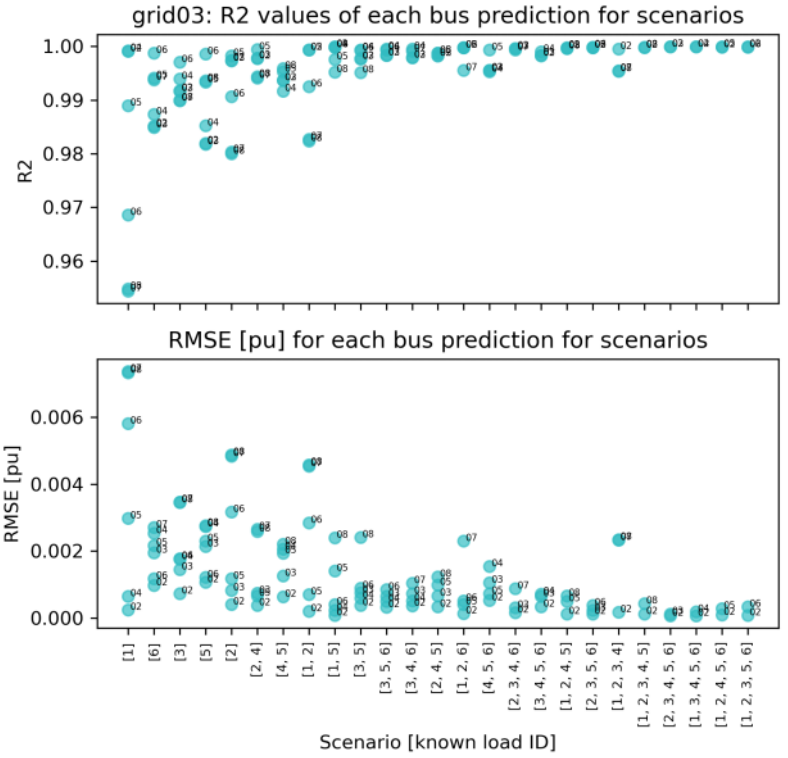
plot_metric_R2_scenarios()

Plot the R2 metric and a chosen metric for each scenario of a grid and for each unknown load. Some transparency is added to the dots as they may sit densly.



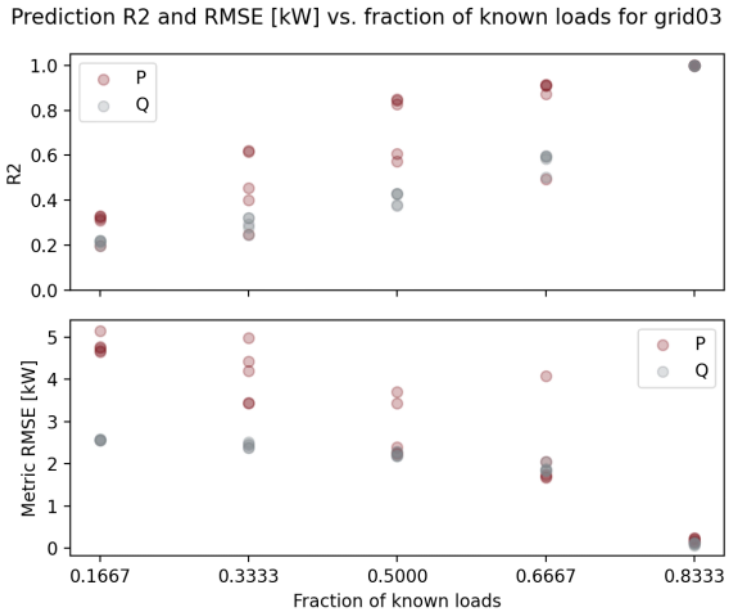
plot_metric_R2_scenarios_buses()

Plot the R2 metric and a chosen metric for each scenario of a grid and for each unknown bus.



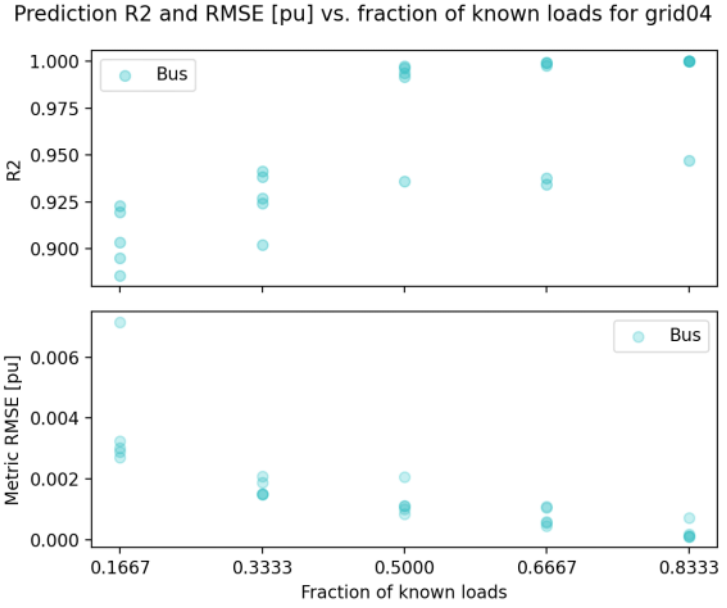
plot_metric_R2_fraction()

Plots averaged R2 / metric for each fraction of unknown loads in the scenarios for a grid.



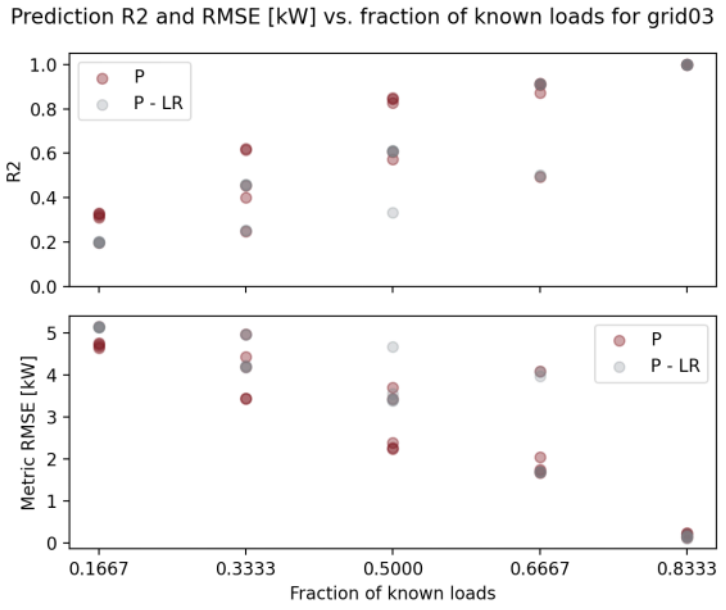
plot_metric_R2_fraction_bus()

Plots averaged R2 / metric for buses for each fraction of unknown loads in the scenarios for a grid.



plot_metric_R2_fraction_LR()

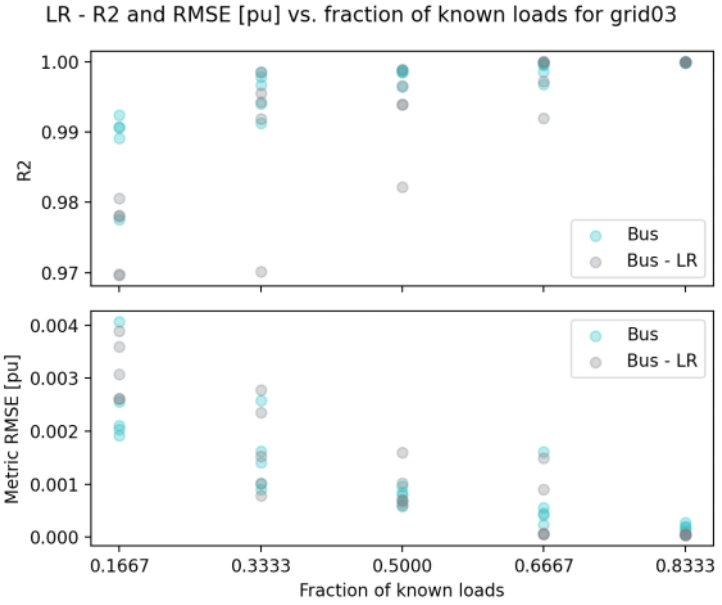
Plots averaged R2 / metric for each fraction of unknown loads in the scenarios for a grid.



plot_metric_R2_fraction_general()

Customizable via arguments - plots averaged R2 / metric for each fraction of unknown loads / buses in the scenarios for a grid.

Example:



1. FAQ